Year : 2023

High-performance computing approaches to solve large-scale dynamic models in economics and finance

Mikushin Dmitry

Mikushin Dmitry, 2023, High-performance computing approaches to solve large-scale dynamic models in economics and finance

Originally published at : Thesis, University of Lausanne

Posted at the University of Lausanne Open Archive http://serval.unil.ch
Document URN :

UNIL | Université de Lausanne

FACULTÉ DES HAUTES ÉTUDES COMMERCIALES

DÉPARTEMENT D'ÉCONOMIE

High-performance computing approaches to solve large-scale dynamic models in economics and finance

THÈSE DE DOCTORAT

présentée à la

Faculté des Hautes Études Commerciales
de l'Université de Lausanne

pour l'obtention du grade de
Docteur en Business Analytics

par

Dmitry MIKUSHIN

Directeur de thèse
Prof. Simon Scheidegger

Co-directeur de thèse
Prof. Philipp Renner

Jury

Prof. Rafael Lalive, Président
Prof. Michalis Vlachos, expert interne
Prof. Florian Oswald, expert externe

LAUSANNE
2023

UNIL | Université de Lausanne

FACULTÉ DES HAUTES ÉTUDES COMMERCIALES

DÉPARTEMENT D'ÉCONOMIE

High-performance computing approaches to solve large-scale dynamic models in economics and finance

THÈSE DE DOCTORAT

présentée à la

Faculté des Hautes Études Commerciales
de l'Université de Lausanne

pour l'obtention du grade de
Docteur en Business Analytics

par

Dmitry MIKUSHIN

Directeur de thèse
Prof. Simon Scheidegger

Co-directeur de thèse
Prof. Philipp Renner

Jury

Prof. Rafael Lalive, Président
Prof. Michalis Vlachos, expert interne
Prof. Florian Oswald, expert externe

LAUSANNE
2023

UNIL | Université de Lausanne
HEC Lausanne
Le Décanat
Bâtiment Internef
CH-1015 Lausanne

I M P R I M A T U R

Sans se prononcer sur les opinions de l'auteur, la Faculté des Hautes Etudes Commerciales de l'Université de Lausanne autorise l'impression de la thèse de Monsieur Dmitry MIKUSHIN, titulaire d'un master en Applied Mathematics and Computer Science de l'Université d'État Lomonossov de Moscou, en vue de l'obtention du grade de docteur en Business Analytics.

La thèse est intitulée :

HIGH-PERFORMANCE COMPUTING APPROACHES TO SOLVE LARGE-SCALE DYNAMIC MODELS IN ECONOMICS AND FINANCE

Lausanne, le 9 février 2023

La Doyenne

Marianne SCHMID MAST

Members of the Thesis Commitee

Prof. Simon Scheidegger
Department of Economics, HEC Lausanne, Switzerland
Directeur de thèse

Prof. Philipp Renner
Department of Economics, Lancaster University
Co-directeur de thèse

Prof. Rafael Lalive
Department of Economics, HEC Lausanne, Switzerland
Président de Jury de thèse

Prof. Michalis Vlachos
Department of Economics, HEC Lausanne, Switzerland
Expert interne de Jury de thèse

Prof. Florian Oswald
Department of Economics, SciencesPo, Paris
Expert interne de Jury de thèse

University of Lausanne
Faculty of Business and Economics

PhD in Business Analytics

I hereby certify that I have examined the doctoral thesis of

Dmitry MIKUSHIN

and have found it to meet the requirements for a doctoral thesis.
All revisions that I or committee members
made during the doctoral colloquium
have been addressed to my entire satisfaction.

Signature: _____ Date: 7.12.22

Prof. Simon SCHEIDEGGER
Thesis supervisor

University of Lausanne
Faculty of Business and Economics

PhD in Business Analytics

I hereby certify that I have examined the doctoral thesis of

Dmitry MIKUSHIN

and have found it to meet the requirements for a doctoral thesis.
All revisions that I or committee members
made during the doctoral colloquium
have been addressed to my entire satisfaction.

Signature:  Date: 6.12.2022

Prof. Philipp RENNER
Thesis co-supervisor

University of Lausanne
Faculty of Business and Economics

PhD in Business Analytics

I hereby certify that I have examined the doctoral thesis of

Dmitry MIKUSHIN

and have found it to meet the requirements for a doctoral thesis.
All revisions that I or committee members
made during the doctoral colloquium
have been addressed to my entire satisfaction.

Signature: _____ Date: 7.12.2022

Prof. Michalis VLACHOS
Internal member of the doctoral committee

University of Lausanne
Faculty of Business and Economics

PhD in Business Analytics

I hereby certify that I have examined the doctoral thesis of

Dmitry MIKUSHIN

and have found it to meet the requirements for a doctoral thesis.
All revisions that I or committee members
made during the doctoral colloquium
have been addressed to my entire satisfaction.

Signature: _____ Date: 7 December 2022

Prof. Florian OSWALD
External member of the doctoral committee

First you jump off the cliff and build
your wings on the way down.

— Ray Bradbury

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor Prof. Simon Scheidegger for his constant support, guidance and infinite patience during my PhD study. His immense knowledge, breakthrough ideas and reviews were vital in inspiring me along the way. Prof. Simon Scheidegger has constantly motivated me to explore alternative ways to view and solve problems and achieve results.
I would also like to thank Prof. Philipp Eisenhauer for his valuable advice and constructive feedback during these years of research. Furthermore, I am very grateful to the team of Faculté des Hautes études commerciales and University of Lausanne for giving me this opportunity.
I would like to thank Prof. Florian Oswald, Dr. Philipp Renner, and Prof. Michalis Vlachos for being part of my thesis committee and for their thoughtful comments.
Last but not least, I would like to thank my family and friends for their unconditional support during these years.

Clarens, January 31, 2023 D. M.

Abstract

This thesis consists of three applications of contemporary high-performance computing to accelerate large-scale dynamic models in economics and finance. The first chapter is entitled “Scalable high-dimensional dynamic stochastic economic modeling” and presents a highly parallelizable and flexible computational method to solve high-dimensional stochastic dynamic economic models. By exploiting the generic iterative structure of this broad class of economic problems, we propose a parallelization scheme that favors hybrid massively parallel computer architectures. Numerical experiments on “Piz Daint” at the Swiss National Supercomputing Centre show that high-dimensional international real business cycle models can be efficiently solved in parallel up to 2,048 compute nodes. The second chapter is called “Rethinking large-scale economic modeling for efficiency: optimizations for GPU and Xeon Phi clusters” and proposes a massively parallelized and optimized framework to solve high-dimensional dynamic stochastic economic models on modern GPU- and KNL-based clusters. Numerical experiments show that our framework scales to at least 4,096 compute nodes. The third chapter, titled “GPU-Accelerated Dynamic Human Capital Models” develops a generic computational method for dynamic discrete-choice models. We align the generic numerical properties of the models under consideration with the recent advancements in GPU computing hardware in order to solve, simulate, and calibrate models of great complexity in relatively short times. Our tests show a speedup of at least three orders of magnitude over the previous state of the art.

Résumé

Cette thèse consiste en trois applications du calcul haute performance contemporain pour accélérer les modèles dynamiques à grande échelle en économie et en finance. Le premier chapitre s'intitule "Scalable high-dimensional dynamic stochastic economic modeling" et présente une méthode de calcul flexible et hautement parallélisable pour résoudre des modèles économiques dynamiques stochastiques à haute dimension. Nous proposons un schéma de parallélisation qui favorise les architectures informatiques hybrides massivement parallèles. Des test numériques au Centre national suisse de supercalcul montrent que des modèles internationaux de cycle économique réel de haute dimension peuvent être résolus efficacement en parallèle jusqu'à 2,048 nœuds de calcul. Le deuxième chapitre s'intitule "Rethinking large-scale economic modeling for efficiency : optimizations for GPU and Xeon Phi clusters" et propose un cadre massivement parallélisé et optimisé pour résoudre des modèles économiques stochastiques dynamiques de haute dimension sur des clusters modernes basés sur GPU et KNL. Notre tests montrent que notre cadre s'adapte à au moins 4,096 nœuds de calcul. Le troisième chapitre, intitulé "GPU-Accelerated Dynamic Human Capital Models", développe une méthode de calcul générique pour les modèles dynamiques à choix discret. Nous alignons les propriétés numériques génériques des modèles considérés avec les progrès récents du matériel de calcul GPU afin de résoudre, simuler et calibrer des modèles de grande complexité dans des délais relativement courts. Nos tests montrent une accélération d'au moins trois ordres de grandeur par rapport à l'état de l'art précédent.

Synthesis

This thesis consists of three applications of contemporary high-performance computing (HPC) to accelerate large-scale dynamic models in economics and finance.

The first chapter is entitled *"Scalable high-dimensional dynamic stochastic economic modeling"* (Brumm et al., 2015), is co-authored with Johannes Brumm (KIT, Germany), Simon Scheidegger (UNIL, Switzerland), and Olaf Schenk (USI, Switzerland). In this work, we present a highly parallelizable and flexible computational method to solve high-dimensional stochastic dynamic economic models. Solving such models often requires the use of iterative methods, like time iteration or dynamic programming. By exploiting the generic iterative structure of this broad class of economic problems, we propose a parallelization scheme that favors hybrid massively parallel computer architectures. Within a parallel nonlinear time iteration framework, we interpolate policy functions partially on GPUs using an adaptive sparse grid algorithm with piecewise linear hierarchical basis functions. GPUs accelerate this part of the computation by one order of magnitude, thus reducing overall computation time by 50%. The developments in this paper include the use of a fully adaptive sparse grid algorithm and the use of a mixed MPI-Intel TBB-CUDA/Thrust implementation to improve the interprocess communication strategy on massively parallel architectures. Numerical experiments on "Piz Daint" (Cray XC30) at the Swiss National Supercomputing Centre show that high-dimensional international real business cycle models can be efficiently solved in parallel. To the best of our knowledge, this performance on a massively parallel petascale architecture for such nonlinear high-dimensional economic models has not been possible prior to the present work.

The second chapter is called *"Rethinking large-scale economic modeling for efficiency: optimizations for GPU and Xeon Phi clusters"* (Scheidegger et al., 2018), is co-authored with Simon Scheidegger (UNIL, Switzerland), Felix Kubler (UZH, Switzerland), Olaf Schenk (USI, Switzerland). In this publication, we propose a massively parallelized and optimized framework to solve high-dimensional dynamic stochastic economic models on modern GPU- and KNL-based clusters. First, we introduce a novel approach for adaptive sparse grid index compression alongside surplus matrix reordering, which

significantly reduces the global memory throughput of the compute kernels and maps randomly accessed data onto cache or fast shared memory. Second, we fully vectorize the compute kernels for AVX, AVX2, and AVX512 CPUs, respectively. Third, we develop a hybrid cluster-oriented work-preempting scheduler based on TBB, which evenly distributes the time iteration workload onto available CPU cores and accelerators. Numerical experiments on Cray XC40 KNL "Grand Tave" and on Cray XC50 "Piz Daint" systems at the Swiss National Supercomputer Centre (CSCS) show that our framework scales nicely to at least 4,096 compute nodes, resulting in an overall speedup of more than four orders of magnitude compared to a single, optimized CPU thread. As an economic application, we compute global solutions to an annually calibrated stochastic public finance model with sixteen discrete, stochastic states with unprecedented performance.

The third chapter, titled *"GPU-Accelerated Dynamic Human Capital Models"* is co-authored with Philipp Eisenhauer (UBonn, Germany and Amazon, USA) and Simon Scheidegger (UNIL, Switzerland). In this paper, we develop a generic computational method for dynamic discrete-choice models. Building on the RESPY project (https://respy.readthedocs.io), an open-source research code for the flexible specification, simulation, and estimation of discrete-choice models, we align the generic numerical properties of the models under consideration with the recent advancements in General-Purpose Graphics Processing Units (GPU) computing hardware in order to solve, simulate, and calibrate models of great complexity in relatively short times. Specifically, our contribution is threefold. First, we propose an optimal memory layout for EKW model data, so that the CPU could perform backward induction with minimum possible pipeline stalls. Second, we derive a partitioning algorithm to distribute the EKW problem to hundred thousand of parallel workers. And third, we develop a generic C++ model core, which is compatible with CUDA and HIP GPU runtimes used by 5 out of 6 currently fastest supercomputers. Our tests show a speedup of at least three orders of magnitude over the previous state of the art, which will allow us to tackle discrete-choice models of unprecedented complexity, which opens the room for novel applications that were previously thought to be intractable.

Contents

Part I

1 Scalable High-Dimensional Dynamic Stochastic Economic Modeling

We present a highly parallelizable and flexible method to compute global solutions of high-dimensional stochastic dynamic economic models. By exploiting the generic structure of such problems, we propose a parallelization scheme that favours CPU-GPU hybrid supercomputing systems. Within a MPI-parallel time-iteration framework, we interpolate policy functions on GPUs using an adaptive sparse grid algorithm with piecewise multi-linear hierarchical basis functions. The defining feature of sparse grids is that they grow considerably slower with increasing dimension than standard tensor product grids. Moreover, in order to capture steep gradients and non-differentiabilities, the grid scheme is automatically refined locally. Due to the high arithmetic density of the interpolation, GPUs can accelerate this part of the computations by more than one order of magnitude, thus reducing the overall computational time by a factor of three. The proposed algorithm enables us to efficiently solve dynamic economic models of a level of complexity and heterogeneity not possible before. To demonstrate the performance of our method, we apply it to high-dimensional international real business cycle models with capital adjustment costs and irreversible investment. Performance tests on the 'Piz Daint Cray CX30' supercomputer indicate that our algorithm scales nicely to at least ten thousand cores for intermediate-sized problems, and even reaches sustained petaflop performance in the case of weak scaling.

1.1 Introduction

Driven by theoretical developments and the availability of "big data," contemporary models in economics and finance have seen tremendous growth in complexity. Heterogeneity between types of agents, such as hand-to-mouth and non hand-to-mouth consumers (see, e.g., Bilbiie (2008); Debortoli and Galí (2017); Kaplan et al. (2018)),

financial frictions (see, e.g., Fernández-Villaverde et al. (2016); Dou et al. (2017), and references therein),such as borrowing constraints, and distributional channels (see, e.g., Krueger et al. (2016)) are widely recognized as key ingredients for modern macroeconomic models.

Clearly, any economy is an extremely complex system. Even when modeling only the most relevant features of a small part of this system, one easily ends up with a large and intricate formal structure. This is often due to the heterogeneity across different consumers, workers, households, firms, sectors, or countries. A further complication stems from the fact that human beings choose their actions based on expectations about an uncertain future. This feedback from the future makes dynamic stochastic economic modeling particularly difficult (see, e.g., Ljungqvist and Sargent (2000); Stokey et al. (1989a)). Model-based economics has for the most part reacted to this challenge in two ways. Either by focusing on qualitative results obtained from extremely simplified models with little heterogeneity, or by only locally solving the equation systems that describe the dynamics around a so-called steady state. In contrast, solving for the global solution of a model with substantial heterogeneity is very costly: The computation time and storage requirements increase dramatically with the amount of heterogeneity, i.e. with the dimensionality of the problem. It is therefore often far beyond the scope of current methods to include as much heterogeneity as a natural modeling choice would suggest. In overlapping generations models researchers often use time-steps that exceed one year by far. For instance, Krueger and Kubler (2004) analyze the welfare implications of social security reform in a model where one period corresponds to six years, thereby reducing the number of adult cohorts and thus the dimensionality of the problem by a factor of six. Similarly, international real business cycle (IRBC) models often include only a very small number of countries or regions. Bengui et al. (2013), for example, analyze cross-country risk-sharing at the business cycle frequency using a two country model — one 'focus' country versus the rest of the world. Reducing the dimensionality of the problem in such ways can deliver valuable qualitative insights. However, to derive solid quantitative results or even to test the robustness of the qualitative results, one often has to look at problems of higher dimension.

Building on Brumm and Scheidegger (2014), this paper shows how we can use modern numerical methods and cutting-edge supercomputing facilities to compute global solutions of high-dimensional dynamic stochastic economic models in a way that fits their generic structure. No matter whether these are solved by iterating on a Bellman equation to update a value function (*parametric dynamic programming*; see, e.g., Judd (1998); Rust (1996)) or by iterating on systems of non-linear equations that represent equilibrium conditions to update functions that represent economic

choices (*time iteration*; see, e.g., Judd (1998)), the computational challenge is similar:

(i) In each iteration step, an economic function needs to be approximated. For this purpose, the function value has to be determined at many points in the high-dimensional state space, and

(ii) each point involves solving a high-dimensional maximization problem (for dynamic programming) or a system of nonlinear equations (for time iteration).

These two important features of the considered problems create difficulties in achieving a fast time-to-solution process. We overcome these difficulties by minimizing both the number of points to be evaluated and the time needed for each evaluation. For the first purpose (i) we use adaptive sparse grids (see, e.g., Bungartz and Griebel (2004); Ma and Zabaras (2009)), while the second task (ii) is accomplished using a hybrid parallelization scheme that minimizes interprocess communication by using Intel Threading Building Blocks (TBB) Reinders (2007) and partially offloads the function evaluations to accelerators using CUDA/Thrust Bell and Hoberock (2011). This scheme enables us to make efficient use of modern hybrid high-performance computing facilities, whose performance nowadays reaches multiple petaflops Dongarra and van der Steen (2012). Their hybrid architecture typically features CPU compute nodes with attached GPUs[1]. We show in this paper that the generic structure of an algorithm that solves dynamic economic models by time iteration or dynamic programming using sparse grids is a natural match for such hybrid systems.

Sparse grid interpolation alleviates the *curse of dimensionality* Bellman (1961) faced by interpolation on standard tensor product grids: Starting with a one-dimensional discretization scheme that employs N grid points, a naive extension to d dimensions using tensor products leads to N^d grid points. In the IRBC model we consider as an application, d depends on the number of countries included in the model. Sparse grids reduce the number of grid points needed from the order $\mathcal{O}\left(N^d\right)$ to $\mathcal{O}\left(N \cdot (\log N)^{d-1}\right)$, while the accuracy of the interpolation only slightly deteriorates in the case of sufficiently smooth functions (Bungartz and Griebel, 2004). Sparse grids go back to Smolyak Smolyak (1963) and have been applied to a whole range of different research fields such as physics, visualization, data mining, Hamilton-Jacobi Bellman (HJB) equations, mathematical finance, insurance, and econometrics (Hegland, 2003; Bokanowski et al., 2013; Bungartz and Griebel, 2004; Garcke and Griebel, 2012; Hager et al., 2010; Heinecke and Pflueger, 2013; Holtz, 2011; Murarasu et al., 2011; Winschel

[1]The Swiss National Supercomputer Centre's "Piz Daint" Cray XC30 that is used in the numerical experiments in Sec. 1.5 consists of Intel Xeon E5 processors with NVIDIA Tesla K20X GPUs attached to it; its peak performance is 7.7 petaflops.

and Kraetzig, 2010). Krueger and Kubler (2004) and Judd et al. (2014) solve dynamic economic models using sparse grids with global polynomials as basis functions. In contrast, we use piecewise-linear local basis functions first introduced by Zenger (1991) in the context of sparse grids. The hierarchical structure of these basis functions lends itself for an adaptive refinement strategy as, e.g., in Ma and Zabaras (Ma and Zabaras, 2009), Bungartz and Dirnstorfer Bungartz and Dirnstorfer (2003), or Pflüger Pflüger (2010). This adaptive grid can better capture the local behavior of functions that have steep gradients or even nondifferentiabilities. The latter feature naturally arise from occasionally binding constraints which are present in many economic models yet have so far been tractable only in low-dimensional cases Brumm and Grill (2014); Christiano and Fisher (2000); Hintermaier and Koeniger (2010).

Parallel computing and sparse grids (Deftu and Murarasu, 2013; Heene et al., 2013; Rabitz and Alis, 1999; Hupp et al., 2013; Murараşu et al., 2012; Murarasu, 2013; Murarasu and Weidendorfe, 2012; Pflüger et al., 2014) enters the picture when we have to solve high-dimensional nonlinear equation systems (or maximization problems) at each point of the sparse grid. Fortunately, within each iteration step, these tasks are independent from each other and can thus be solved in parallel by distributing them via MPI Skjellum et al. (1999) to different processes. When searching for the solution to the equation system at a given point, the algorithm has to frequently interpolate the function computed in the previous iteration step. These interpolations take up 99% of the computation time needed to solve the equation system. As they have a high arithmetic intensity, i.e., many arithmetic operations are performed for each byte of memory transfer and access, they are perfectly suited for GPUs (Heene et al., 2013; Gaikwad and Toke, 2009; Murarasu et al., 2011). We therefore offload parts of the interpolation tasks from the compute nodes to their attached accelerators, which results in a reduction of the overall computation time by roughly 50%. Due to the indicated high intrinsic level of parallelism, the economic modeling code can efficiently use CPU-GPU hybrid supercomputing systems. Our large scale numerical experiments performed on the "Piz Daint" XC30 machine from the Swiss National Supercomputing Centre (CSCS) show that the developments of this paper make it possible to solve realistically sized and thus high-dimensional, heterogeneous economic models in times that are considerably under one hour. To the best of our knowledge, this has not been possible before. We also observe very good strong scaling efficiencies on "Piz Daint".

Summing up, we present a method for solving a large class of generic high-dimensional dynamic stochastic economic models of size and complexity that were not tractable in a reasonable time before. Using adaptive sparse grids, we build a hybrid-parallel iterative procedure which, by construction, can efficiently use modern high-performance

computing architectures. With this work, we hope to help computational research become a strong "third pillar" of economics, alongside theory and experimentation, as it already is in many sciences, like physics or chemistry. Despite the promise of high-performance computing facilities to solve complex economic models, economists have in the past been restrained in doing so. Single GPUs were used in several applications in order to accelerate computations Aldrich (2014), whereas Cai and Judd Cai et al. (2013) used the high latency "Condor" paradigm to solve dynamic programming problems in parallel. However, apart from Brumm and Scheidegger (2014) and Cai et al. (2015), who recently exploited highly parallel low-latency systems, no one in computational economics has so far made the effort to make efficient use of the most advanced contemporary high-performance computing (HPC) systems.

The remainder of the paper is organized as follows. In Sec. 1.2, we describe the structure of the dynamic economic models we solve. In Sec. 2.3, we briefly outline the construction of adaptive sparse grids. In Sec. 2.4, we embed adaptive sparse grid interpolation in a time iteration algorithm. We then discuss in Sec. 1.5 the performance of this algorithm and report how hybrid parallelization can speed up the computations. Section 1.6 concludes.

1.2 High-dimensional dynamic economic models

To capture complex economic phenomena, models often have to include several different economic *agents* (who choose actions that are optimal given their objectives). Depending on the research question, these agents might represent firms, sectors, countries, or certain subgroups of the population, ordered by skills, age, or other characteristics. If the model is dynamic, it is common practice to consider so-called recursive equilibria (Ljungqvist and Sargent, 2000; Stokey et al., 1989b), where the state of the economy can be summarized by a *state variable* and the dynamics of the economy can be captured by a time-invariant function of this state. In most applications, the state variable contains agents' characteristics, for instance their accumulated assets. When multiple agents and/or several of their relevant characteristics are included, the state of the economy can quickly become high-dimensional. As the state influences agents' behavior and thereby the dynamics of the economic model, it is a serious challenge for numerical methods to capture these dynamics if the state space is high-dimensional. To describe this challenge more formally, we first describe the general structure common to many (infinite-horizon) dynamic economic models. In a second step, we describe one concrete example, the IRBC model (see, e.g., Backus et al. (1992); Kollmann et al. (2011)).

1.2.1 Dynamic economic models as functional equations

Let $x_t \in X \subset \mathbb{R}^d$ denote the state of the economy at time $t \in \mathbb{N}$. Then the actions of all agents can be represented by a *policy function* $p : X \to Y$, where Y is the space of possible *policies*. The stochastic transition of the economy from period t to $t+1$ can then be represented by the distribution of next period's state x_{t+1}, which depends on the current state and policies:

$$x_{t+1} \sim F\left(\cdot | x_t, p(x_t)\right). \tag{1.1}$$

While the distribution F as a function of x_t and $p(x_t)$ is implied by the economic assumptions of the model, the policy function p needs to be determined from *equilibrium conditions*. When using time iteration (see Sec. 2.4), these conditions include agents' first-order optimality conditions, next to other conditions like budget constraints or market clearing (Judd, 1998). Taken together, these conditions constitute a functional equation that the policy function p has to satisfy, namely, that for all $x_t \in X$,

$$0 = \mathbb{E}\left\{E\Big(x_t, x_{t+1}, p\left(x_t\right), p\left(x_{t+1}\right)\Big) | x_t, p(x_t)\right\}, \tag{1.2}$$

where the expectation, represented by the operator $\mathbb{E}$, is taken with respect to the distribution $F\left(\cdot | x_t, p(x_t)\right)$ of next period's state x_{t+1}. The function $E : X^2 \times Y^2 \to \mathbb{R}^{2d}$ represents the period-to-period equilibrium conditions of the model. In most economic applications, this function is nonlinear because of concavity assumptions on utility and production functions. As a consequence, the optimal policy p solving (2.3) will also be nonlinear, or even nonsmooth if the model includes so-called (economic) frictions, like borrowing constraints or irreversible investments. Therefore, approximating this function only locally, which is often sufficient if the model exhibits low volatility, might provide misleading results when larger fluctuations are considered. For such applications, we need a global solution, that is, we need to approximate p over the entire state space X or at least on the ergodic distribution of the stochastic state variable (see, e.g., Judd et al. (2014)). We solve for the policy function p using a time iteration procedure (see Sec. 2.4). As a consequence, we need to interpolate many successive approximations of p. To do this efficiently we employ (adaptive) sparse grids (see Sec. 2.3). Next, we introduce the example that we apply our algorithm to. Readers who are more interested in our solution methods can skip the subsequent example.

1.2.2 Example: The international real business cycle model

To demonstrate the capabilities of our method, we choose the IRBC model, which has become a workhorse for studying methods for solving high-dimensional economic models (Den Haan et al., 2011). The IRBC model is relatively simple and easy to explain, whereas the dimensionality of the model can be scaled up in a straightforward and meaningful way as it just depends linearly on the number of countries considered. This feature of the model allows us to focus on the problem of handling high-dimensional state spaces. To show that we can also handle nonsmooth problems, we consider a version of the IRBC model where investment is irreversible.

Stochastic dynamic equilibrium models have been used to study business cycle fluctuations of economic aggregates like output, consumption, and investment since the seminal work of Kydland and Prescott (1982). Initially, the focus was on models of closed economies with the United States being the main application. Later, these real business cycle models were extended to include several countries and were thus called IRBC models (Backus et al., 1992). They can be used to model comovements and spillovers across countries as well as current account deficits and exchange rate movements. However, previous research (see, e.g., Backus et al. (1992); Bengui et al. (2013) analyzed setups with only a very small number of countries/regions (mainly two or three) and mostly considered models without occasionally binding constraints. We solve models with many more countries that nevertheless include occasionally binding constraints, in particular irreversible investment. To focus on the high-dimensionality of the state space, we keep the model simple in other ways. Extending the model in directions that are standard in the literature, however, is not a serious challenge for our solution method. For instance, to discuss exchange rate movements one would have to consider a model with several commodities, differentiated by the country in which they are produced. This extension would not increase the dimension of the state space, just the size of the equations systems that have to be solved.

In the model we are considering, there are M countries, $j = 1,\dots,M$, each using its accumulated capital stock, k_t^j, to produce the output good, which can either be used for investment, χ_t^j, or for consumption, c_t^j, generating utility, $u^j(c_t^j)$, with constant relative risk aversion utility $u^j(c) = c^{-\gamma}/(1-\gamma)$ and risk-aversion parameter γ. Investment is subject to adjustment costs, and it is irreversible in the following sense: The capital stock of a country can neither be consumed nor used for production in another country — an assumption that seems more realistic than perfect reversibility, which is normally assumed to keep the model tractable. However, capital depreciates at a rate $\delta > 0$ and can thus nevertheless shrink over time if there is not enough new

investment. Thus, the law of motion of capital is

$$k_{t+1}^j = k_t^j \cdot (1-\delta) + \chi_t^j. \tag{1.3}$$

The amount produced by each country is given by

$$a_t^j \cdot A \cdot (k_t^j)^\kappa.$$

It thus depends on the size of the capital stock employed, k_t^j, on the overall productivity level, A, as well as on the country specific productivity level, a_t^j, which has the following law of motion:

$$\ln a_t^j = \rho \cdot \ln a_{t-1}^j + \sigma \left(e_t^j + e_t^{M+1} \right). \tag{1.4}$$

The parameters ρ and σ determine persistence and volatility in productivity. The country specific shocks, $e_t^j \sim \mathcal{N}(0,1)$, as well as the global shock, $e_t^{M+1} \sim \mathcal{N}(0,1)$, are assumed to be independent from each other and across time. However, we assume that countries can insure themselves against these shocks by trading assets with payoffs that are contingent on the realization of these shocks. This complete markets assumption[2] implies that the market allocation of capital and consumption (the so-called decentralized competitive equilibrium) can be obtained as the solution to a social planner's problem: maximize the weighted sum of all countries utility from consumption, weighted by welfare weights, τ^j, which depend on the initial capital stocks of the countries. Thus, the social planner solves the following infinite-horizon problem, where the future is discounted by the discount factor, β:

$$\max_{\{c_t^j, k_t^j\}} \mathbb{E}_0 \sum_{j=1}^{M} \tau^j \cdot \left(\sum_{t=1}^{\infty} \beta^t \cdot u^j(c_t^j) \right), \tag{1.5}$$

subject to the aggregate resource constraint

$$\sum_{j=1}^{M} \left(a_t^j \cdot A \cdot (k_t^j)^\kappa - k_t^j \cdot \frac{\phi}{2} \cdot (g_{t+1}^j)^2 - \chi_t^j - c_t^j \right) = 0, \tag{1.6}$$

and the constraint that investment in each country j, χ_t^j, is irreversible,

$$\chi_t^j \geq 0. \tag{1.7}$$

[2] We follow the comparison study by Kollmann et al. (2011) in assuming complete markets. However, we are not directly solving the social planner problem, but iterate on the period-to-period equilibrium conditions given in (1.8)–(1.10) below. With incomplete markets, these conditions have a similar structure and our method can thus be applied as well.

In (1.6), the first term is the amount produced by each country, while the second term represents the convex adjustment costs, where ϕ parametrizes the intensity of capital adjustment costs, and $g_{t+1}^j \equiv k_{t+1}^j / k_t^j - 1$ is the growth rate of capital in country j.

So far, we are considering an infinite horizon problem. However, as mentioned above, it is common practice in economics to focus on recursive equilibria (Stokey et al., 1989a; Ljungqvist and Sargent, 2000), where the state of the economy is summarized by a state variable and the dynamics of the economy is capture by a time-invariant function of this state. We now briefly present the recursive structure of the above IRBC model, while we refer the reader to Brumm and Scheidegger (2014) for the derivation of the equilibrium conditions. The state variables of the IRBC model with M countries consist of

$$x_t = \left(a_t^1, \ldots, a_t^M, k_t^1, \ldots, k_t^M\right) \in X \subset \mathbb{R}^{2M},$$

where a_t^j and k_t^j are the productivity and capital stock of country j, respectively. The policy function $p : \mathbb{R}^{2M} \to \mathbb{R}^{2M+1}$ maps the current state into investment choices, χ_t^j, the multipliers for the irreversibility constraints, μ_t^j, and the multiplier of the aggregate resource constraint, λ_t:

$$p(x_t) = \left(\chi_t^1, \ldots, \chi_t^M, \mu_t^1, \ldots, \mu_t^M, \lambda_t\right).$$

The investment choices determine next period's capital stock in a deterministic way through (1.3). In contrast, the law of motion of productivity, (1.4), is stochastic. Taken together, (1.3) and (1.4) specify the distribution of x_{t+1} (corresponding to (1.1) in the general problem above). The period-to-period equilibrium conditions of this model (corresponding to (2.3)) consist of three types of equations. First are the optimality conditions for investment in capital in each country j:[3]

$$\begin{aligned} &\lambda_t \cdot \left[1 + \phi \cdot g_{t+1}^j\right] - \mu_t^j \\ &-\beta \mathbb{E}_t \left\{ \lambda_{t+1} \left[a_{t+1}^j A \kappa (k_{t+1}^j)^{\kappa-1} + 1 - \delta + \frac{\phi}{2} g_{t+2}^j \left(g_{t+2}^j + 2\right) \right] - \mu_{t+1}^j (1-\delta) \right\} = 0. \end{aligned} \tag{1.8}$$

Second is the irreversibility assumption for investment in each country j, and the associated complementarity conditions,

$$\chi_t^j \geq 0, \mu_t^j \geq 0, \chi_t^j \cdot \mu_t^j = 0. \tag{1.9}$$

[3]The expectation in Eq. 1.8 below is given by the following integral: $(2\pi)^{-\frac{M+1}{2}} \int \Omega(x_t, e_t) \cdot \exp\left(-e_t' \cdot e_t / 2\right) \cdot de_t$, where $\Omega(x_t, e_t)$ is defined as the term whithin the expectation, and $e_t = \left(e_t^1, ..., e_t^{M+1}\right)$.

Table 1.1: Choice of parameters for the IRBC model

Parameter	Symbol	Value
Discount factor	β	0.99
IES of country j	γ^j	a+(j-1)(b-a)/(M-1) with a=0.25, b=1
Capital share	κ	0.36
Depreciation	δ	0.01
Standard deviation of log-productivity shocks	σ	0.01
Autocorrelation of log-productivity	ρ	0.95
Intensity of capital adjustment costs	ϕ	0.50
Aggregate productivity	A	$\big(1-\beta(1-\delta)\big)/(\alpha\cdot\beta)$
Welfare weights	τ^j	A^{1/γ^j}
Number of countries	M	2,3,4

Finally is the aggregate resource constraint

$$\sum_{j=1}^{M}\left(a_t^j\cdot A\cdot(k_t^j)^\kappa - k_t^j\cdot\frac{\phi}{2}\cdot(g_{t+1}^j)^2 - \chi_t^j - c_t\right)=0, \tag{1.10}$$

where we can use the fact that $c_t=(\lambda_t/\tau_j)^{-\gamma^j}$ at an optimal choice.

For all the parameters of the economic model, we make standard assumptions, as in Brumm and Scheidegger (2014) and Den Haan et al. (2011). Nevertheless, we report them here and in Tab. 1.1 for completeness. For our computations, we choose an asymmetric specification where preferences are heterogeneous across countries. In particular, the intertemporal elasticity of substitution (IES) of the M countries is evenly spread over the interval $[0.25,1]$. The welfare weights τ^j need not be specified, as they do not matter for the capital allocation, but only for the consumption allocation which we do not consider. Finally, the parameter A is chosen such that the capital of each country is equal to 1 in the deterministic steady state.

1.3 Sparse grid interpolation

For the time iteration algorithm we propose in Sec. 2.4, we need to repeatedly evaluate (policy) functions at arbitrary coordinates within the domain of interest. As a single function evaluation can be very expensive — it involves solving a system of nonlinear equations (cf. (1.8) and (1.6)) — we need an efficient interpolation scheme. Our method of choice is adaptive sparse grid interpolation, which we now explain.

Generally speaking, we aim to approximate each individual variable of the policy or

value function by a function $f : \Omega \to \mathbb{R}$ as

$$f(\vec{x}) \approx u(\vec{x}) := \sum_i \alpha_i \phi_i(\vec{x}) \tag{1.11}$$

with coefficients α_i and a set of appropriate (piecewise linear) basis functions $\phi_i(\vec{x})$. Employing standard discretization methods for the high-dimensional domain Ω is out of the question, as ordinary discretization approaches yield too many grid points where the functions have to be evaluated. Starting with a one-dimensional discretization scheme that employs N grid points, a straightforward extension to d dimensions by a tensor product construction would lead to N^d grid points, encountering the so-called *curse of dimensionality* Bellman (1961). The exponential dependence of the overall computational effort on the number of dimensions is a prohibitive obstacle for the numerical treatment of high-dimensional problems. Sparse grids, on the other hand, are able to alleviate this *curse of dimensionality* by reducing the number of grid points by orders of magnitude, yet with only slightly deteriorated accuracy if the underlying function is sufficiently smooth (Bungartz and Griebel, 2004).

In this section, we therefore first provide a brief introduction to classical, i.e., non-adaptive sparse grid interpolation. Subsequently, we also show how the hierarchical structure of the basis functions and the associated sparse grid can be used to refine the grid such that it can better capture the local behavior of the functions to be interpolated. In Sec. 2.4, we will see in the case of an economic model that adaptive sparse grids outperform classical sparse grids by far when it comes to interpolating functions that exhibit steep gradients or nondifferentiabilities.

1.3.1 Notation

Following Bungartz and Griebel (2004) and Garcke and Griebel (2012), we first introduce some notation and definitions that we will require later on. For all our considerations, we will focus on the domain $\Omega = [0,1]^d$, where d is the dimensionality of the economic problem. This situation can be achieved for other domains by a proper rescaling. Moreover, let $\vec{l} = (l_1, ..., l_d) \in \mathbb{N}^d$ and $\vec{i} = (i_1, ..., i_d) \in \mathbb{N}^d$ denote multi-indices, and define $|\vec{l}|_1 := \sum_{t=1}^d l_t$ and $|\vec{l}|_\infty := \max_{1 \le t \le d} l_t$.

1.3.2 Hierarchical basis functions

We use a sparse grid interpolation method that is based on a hierarchical decomposition of the underlying approximation space. Such a hierarchical structure is convenient both for local adaptivity (see Sec. 1.3.4) and for the use of massively par-

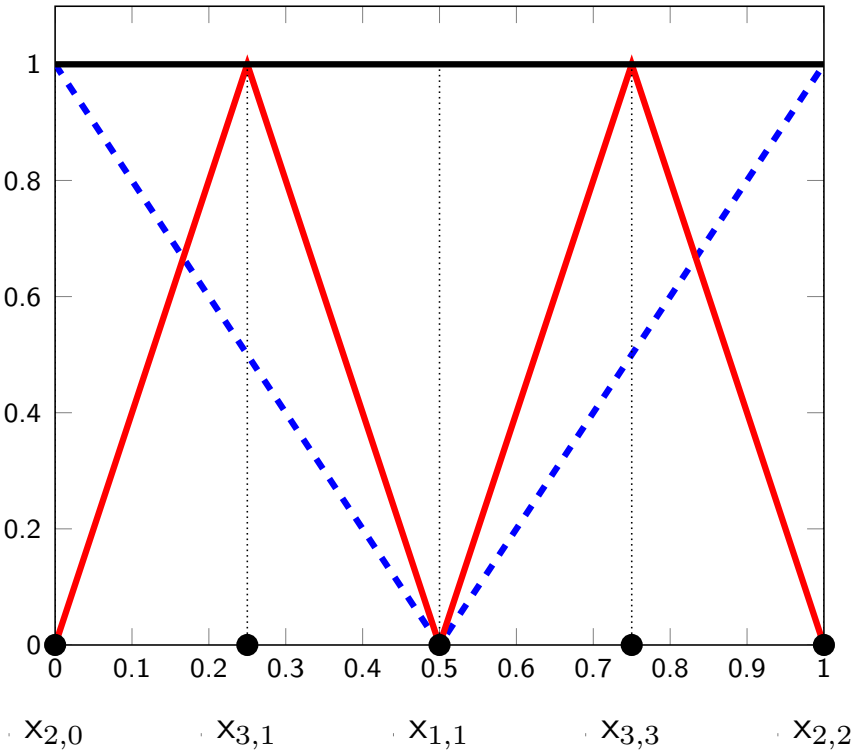


Figure 1.1: Hierarchical basis functions of V_3 in (2.11) in one dimension. Level $l = 1$ (solid black), $l = 2$ (dashed blue), and $l = 3$ (solid red).

allel architectures (see Sec. 2.4). We now explain this hierarchical structure, starting with the one-dimensional case, i.e., $\Omega = [0,1]$. Afterwards, we will extend it to the multivariate case using tensor products. The equidistant sparse grid interpolant we use below (Bungartz and Griebel, 2004; Garcke and Griebel, 2012) consists of a combination of nested one-dimensional grids of different refinement levels. For a given level $l \in \mathbb{N}$, the grid points on $[0,1]$ are distributed as

$$x_{l,i} = \begin{cases} 0.5, & l = i = 1, \\ i \cdot 2^{1-l}, & i = 0,...,2^{l-1},\ l > 1. \end{cases} \tag{1.12}$$

The corresponding piecewise linear basis functions for $x \in [0,1]$ are given by

$$\begin{aligned} &\phi_{l,i}(x) \\ &= \begin{cases} 1, & l = i = 1, \\ \max(1 - 2^{l-1} \cdot |x - x_{l,i}|, 0) & i = 0,...,2^{l-1},\ l > 1. \end{cases} \end{aligned} \tag{1.13}$$

Note that the basis function of level 1 is a constant rather than a hat function, which is different from many other sparse grid constructions (see, e.g., Bungartz and Griebel

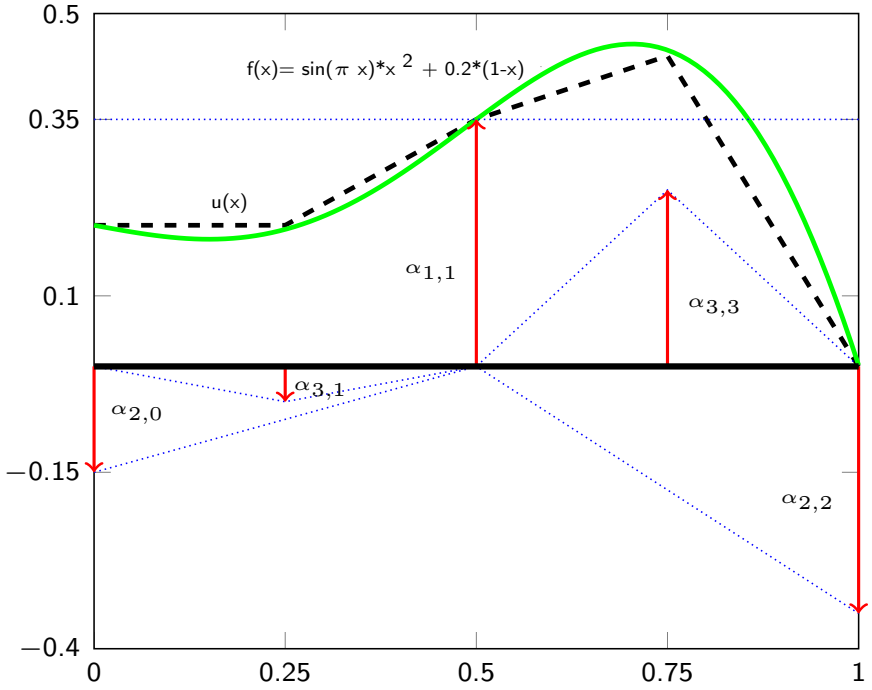


Figure 1.2: Construction of $u(x)$ interpolating $f(x) = \sin(\pi \cdot x) \cdot x^2 + 0.2 \cdot (1-x)$ with hierarchical linear basis functions of levels 1, 2, and 3. The hierarchical surpluses $\alpha_{l,i}$ that belong to the respective basis functions are indicated by arrows. They are simply the difference between the function values at the current and the previous interpolation levels.

(2004); Garcke and Griebel (2012), and references therein). The one-dimensional basis functions can be extended to d-dimensional ones on the unit cube $\Omega = [0,1]^d$ by a tensor product construction. For each grid point $\vec{x}_{\vec{l},\vec{i}} = (x_{l_1,i_1},...,x_{l_t,i_t})$, an associated piecewise d-linear basis function $\phi_{\vec{l},\vec{i}}(\vec{x})$ is defined as the product of the one-dimensional basis functions (cf. (2.6))

$$\phi_{\vec{l},\vec{i}}(\vec{x}) := \prod_{t=1}^{d} \phi_{l_t,i_t}(x_t). \tag{1.14}$$

Next, the hierarchical increment spaces $W_{\vec{l}}$ are defined by

$$W_{\vec{l}} := \operatorname{span}\{\phi_{\vec{l},\vec{i}} : \vec{i} \in I_{\vec{l}}\} \tag{1.15}$$

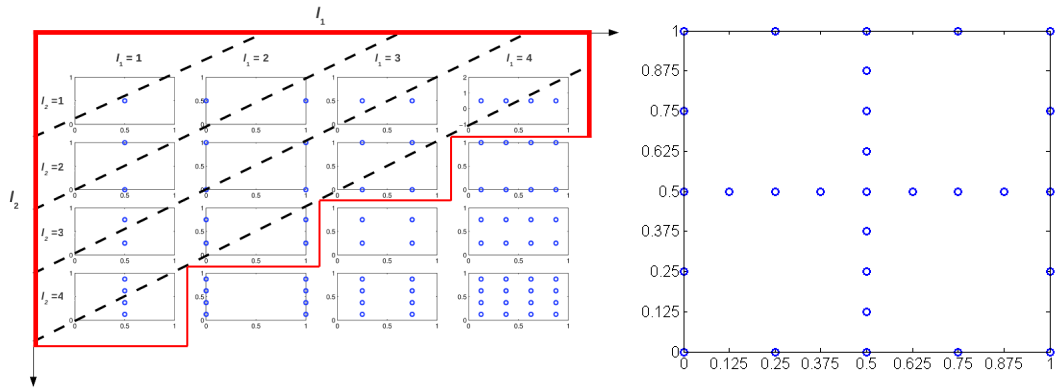


Figure 1.3: Left: schematic construction of a level 4 sparse grid V_4^S Klimke and Wohlmuth (2005) in two dimensions. Right: sparse grid space V_4^S in 2 dimensions, constructed according to (2.13) from the increments displayed in the left graphic.

with the index set $I_{\vec{l}}$ given as

$$I_{\vec{l}} := \begin{cases} \{\vec{i} : i_t = 1, 1 \le t \le d\} & \text{if } l = 1, \\ \{\vec{i} : 0 \le i_t \le 2,\ i_t \text{ even}, 1 \le t \le d\} & \text{if } l = 2, \\ \{\vec{i} : 0 \le i_t \le 2^{l_t - 1},\ i_t \text{ odd}, 1 \le t \le d\} & \text{else.} \end{cases} \tag{1.16}$$

Fig. 1.1 depicts the first three levels of the associated $1d$ hierarchical, piecewise linear basis functions. Consequently, the hierarchical increment spaces $W_{\vec{l}}$ are related to the space V_n of $d-linear$ functions with mesh size $h_n = 2^{1-n}$ in each dimension by

$$V_n := \bigoplus_{l_1=1}^{n} \cdots \bigoplus_{l_d=1}^{n} W_{\vec{l}} = \bigoplus_{|l|_\infty \le n} W_{\vec{l}}, \tag{1.17}$$

leading to a full grid with $\mathcal{O}(2^{nd})$ grid points. The interpolant of f, namely, $u(\vec{x}) \in V_n$, can uniquely be represented by

$$f(\vec{x}) \approx u(\vec{x}) = \sum_{|l|_\infty \le n} \sum_{\vec{i} \in I_{\vec{l}}} \alpha_{\vec{l},\vec{i}} \cdot \phi_{\vec{l},\vec{i}}(\vec{x}) \tag{1.18}$$

with $\alpha_{\vec{l},\vec{i}} \in \mathbb{R}$. Note that the coefficients $\alpha_{\vec{l},\vec{i}} \in \mathbb{R}$ are commonly termed the hierarchical surpluses Zenger (1991); Bungartz and Griebel (2004). They are simply the difference between the function values at the current and the previous interpolation levels (see Fig. 1.2). As we have chosen our set of grid points to be nested, i.e., such that the set of points X^{l-1} at level $l-1$ with support nodes $\vec{x}_{\vec{l},\vec{i}}$ is contained in X^l, namely, $X^{l-1} \subset X^l$, the extension of the interpolation level from level $l-1$ to l only requires us to evaluate the function at grid points that are unique to X^l, that is, at $X_\Delta^l = X^l \setminus X^{l-1}$.

For a sufficiently smooth function f (which we will make precise in the next section) and its interpolant $u \in V_n$ (Bungartz and Griebel, 2004), we obtain an asymptotic error decay of

$$\| f(\vec{x}) - u(\vec{x}) \|_{L_2} \in \mathcal{O}\left(h_n^2\right), \tag{1.19}$$

but at the cost of

$$\mathcal{O}\left(h_n^{-d}\right) = \mathcal{O}\left(2^{nd}\right) \tag{1.20}$$

function evaluations, encountering the *curse of dimensionality*.

1.3.3 Ordinary sparse grids

As a consequence of the *curse of dimensionality*, the question that needs to be answered is how we can construct discrete approximation spaces that are better than V_n in the sense that the same number of invested grid points leads to a higher order of accuracy. The classical sparse grid construction arises from a cost-to-benefit analysis (see, e.g., Bungartz and Griebel (2004); Garcke and Griebel (2012); Zenger (1991), and references therein) in function approximation. Thereby, functions $f(\vec{x}) : \Omega \to \mathbb{R}$ which have bounded second mixed derivatives are considered. For such functions, the hierarchical coefficients $\alpha_{\vec{l},\vec{i}}$ (see (2.12) and Bungartz and Griebel (2004)) rapidly decay, namely,

$$|\alpha_{\vec{l},\vec{i}}| = \mathcal{O}\left(2^{-2|\vec{l}|_1}\right). \tag{1.21}$$

The strategy for constructing a sparse grid thus is to leave out those subspaces among the full grid space V_n that only contribute little to the interpolant (Bungartz and Griebel, 2004). An optimization with respect to the number of degrees of freedom, i.e., the grid points, and the resulting approximation accuracy directly lead to the sparse grid space V_n^S of level n, defined by

$$V_n^S := \bigoplus_{|\vec{l}|_1 \leq n+d-1} W_{\vec{l}}. \tag{1.22}$$

In Fig. 1.3, we depict its construction for $n = 4$ in two dimensions. V_4^S consists of the hierarchical increment spaces $W_{(l_1,l_2)}$ for $1 \leq l_1, l_2 \leq n = 4$. The area enclosed by the red bold lines marks the region where $|\vec{l}| \leq n + d - 1$, fulfilling (2.10). The blue dots represent the grid points of the respective subspaces. Finally, the dashed black lines indicate the hierarchical increment spaces for constant $|\vec{l}|$. Note that the sparse grid contains only 29 support nodes, whereas a full grid would consist of 81 points.

The concrete choice of subspaces depends on the norm in which we measure the error. The result obtained in (2.13) is optimal for the L_2-norm and the L_∞-norm Bungartz

and Griebel (2004). The number of grid points required by the space V_n^S is now given by Bungartz and Griebel (2004); Garcke and Griebel (2012)

$$|V_n^S| = \mathcal{O}\left(h_n^{-1} \cdot \left(\log(h_n^{-1})\right)^{d-1}\right). \tag{1.23}$$

This is of order $\mathcal{O}\left(2^n \cdot n^{d-1}\right)$, which is a significant reduction of the number of grid points, and thus of the computational and storage requirements compared to $\mathcal{O}\left(2^{nd}\right)$ of the full grid space $|V_n|$ (see Fig. 1.3). In analogy to (2.12), a function $f \in V_n^S \subset V_n$ can now be expanded by

$$f_n^S(\vec{x}) \approx u(\vec{x}) = \sum_{|l|_1 \leq n+d-1} \sum_{\vec{i} \in I_{\vec{l}}} \alpha_{\vec{l},\vec{i}} \cdot \phi_{\vec{l},\vec{i}}(\vec{x}). \tag{1.24}$$

The asymptotic accuracy of the interpolant deteriorates only slightly from $\mathcal{O}\left(h_n^2\right)$ in the case of the full grid (cf. (1.19)) down to

$$\mathcal{O}\left(h_n^2 \cdot \log(h_n^{-1})^{d-1}\right), \tag{1.25}$$

as shown e.g. in Bungartz and Griebel (2004); Garcke and Griebel (2012). Taken together, (1.23) and (1.25) demonstrate why sparse grids are so well suited for high-dimensional problems. In contrast to full grids, their size increases only moderately with dimension, while the accuracy they provide is nearly as good as the one of full grids.

1.3.4 Adaptive sparse grids

In many economic applications (Brumm and Scheidegger, 2014), the functions to be interpolated do not meet the regularity conditions assumed above, but instead have steep gradients, nondifferentiabilities, or even finite discontinuities. In such cases, the classical sparse grid methods outlined so far may fail to provide a good approximation. One effective way to overcome this problem is to adaptively refine the sparse grid in regions with high function variation and spend fewer points in regions of low variation (see, e.g., Bungartz and Griebel (2004); Ma and Zabaras (2009); Pflüger (2010), and references therein). The working principle of the refinement strategy we use is to monitor the size of the hierarchical surpluses (see (1.21)), which reflect the local irregularity of the function. For functions with small mixed-derivatives the hierarchical surpluses rapidly converge to zero as the level l tends to infinity (cf. (1.21)). On the other hand, a nondifferentiability or discontinuity can often be identified by large and slowly decaying hierarchical surpluses. Therefore, we use the hierarchical surpluses as an error indicator and refine the grid around a grid point if its surplus

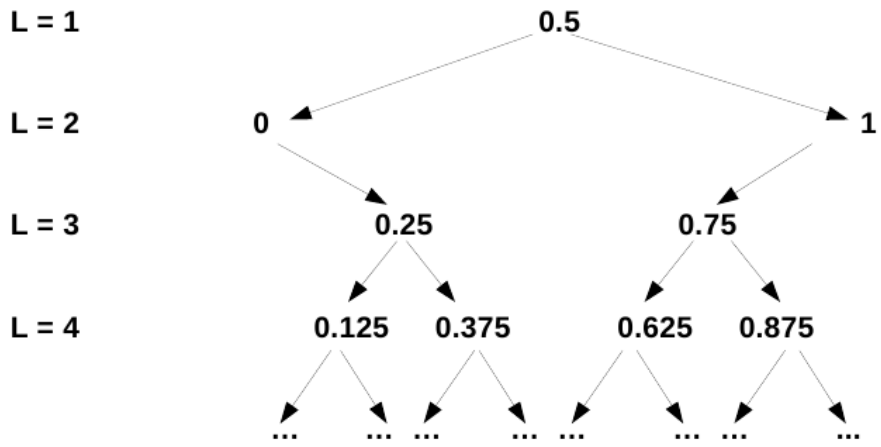


Figure 1.4: One-dimensional tree-like structure of a classical sparse grid (cf. Sec. 1.3.3) for the first three hierarchical levels.

$\alpha_{\vec{l},\vec{i}}$ satisfies

$$|\alpha_{\vec{l},\vec{i}}| \geq \epsilon, \tag{1.26}$$

for a so-called refinement threshold $\epsilon \geq 0$. Technically, the adaptive grid refinement can be built on top of the hierarchical grid structure. The points of the classical sparse grid form a tree-like data structure, as displayed in Fig. 1.4 for the one-dimensional case.[4] Going from one level to the next, we see that there are two *sons* for each grid point (if $l \neq 2$). For example, the point 0.5 from level $l = 1$ is the *father* of the points 0 and 1 from level $l = 2$. In the d-dimensional case, there are consequently two *sons* per dimension for each grid point, i.e., $2d$ *sons* in total. Whenever the criterion given by (1.26) is satisfied, these $2d$ neighbor points of the current point are added to the sparse grid.[5] In this way, we can adapt to nondifferentiabilities induced by occasionally binding constraints that are common in economic models. While existing methods can adapt very precisely to these nondifferentiabilities (Barillas and Fernández-Villaverde, 2007; Brumm and Grill, 2014), adaptive sparse grids work in much higher dimensions.

[4] For more details on the data structure employed to store sparse grids, see Sec. 2.4.2.

[5] We point out that in our application in Sec. 2.4 we interpolate several policies on one grid, i.e., we interpolate a function $f : \Omega \to \mathbb{R}^m$. Therefore, we get m surpluses at each grid point and we thus have to replace the refinement criterion in (1.26) by $g\left(\alpha^1_{\vec{l},\vec{i}}, \ldots, \alpha^m_{\vec{l},\vec{i}}\right) \geq \epsilon$, where the refinement choice is governed by a function $g : \mathbb{R}^m \to \mathbb{R}$. A natural choice for g is the maximum function, which we will use in Sec. 2.4.

1.4 Scalable sparse grid time iteration algorithm

We now describe how to solve the IRBC model introduced in Sec. 1.2.2 using adaptive sparse grids as presented in Sec. 2.3. For this purpose, we build a time iteration algorithm (see, e.g., Judd (1998)) that uses adaptive sparse grid interpolation in each iteration step (cf. Sec. 1.4.1). We parallelize this algorithm by a hybrid parallelization scheme using MPI (Skjellum et al., 1999), Thread Building Blocks (Reinders, 2007), and CUDA/Thrust (Bell and Hoberock, 2011), as outlined in Sec. 1.4.2

1.4.1 Time iteration

The time iteration algorithm that we use to compute a policy function satisfying (2.3) is based on the following heuristic: Solve the equilibrium conditions of the model for today's policy $p : X \to Y$ taking as given an initial guess for the function that represents next period's policy, p_{next}; then, use p to update the guess for p_{next} and iterate the procedure until convergence. Note that in the case of Pareto optimal problems, as the one solved in this paper, convergence of time iteration can be derived from convergence of value function iteration (see Stokey et al. (1989a) for a comprehensive study of value function iteration and its convergence properties) and even explicit error bounds for the approximate policy functions can be obtained under strong concavity (see Maldonado and Svaiter (2007)). For non-optimal economies, results about convergence of time iteration and also the existence of recursive equilibria are harder to obtain, yet are available for large classes of models with heterogeneous agents, incomplete markets, externalities, discretionary taxation and other salient features of applied models (see, e.g., Morand and Reffett (2003); Datta et al. (2005)).

The structure of our time iteration algorithm is given in Algorithm 1. [6] Two remarks about the maximal refinement level, L_{max}, are in place. First, the classical sparse grid of level L is obtained as a special case of this algorithm by setting $L_{max} = L_0 = L$. Second, for the adaptive sparse grid, one could in principle set the maximum refinement level L_{max} to a very large value such that it is never reached for a given refinement threshold. However, this can create practical problems: in case of high curvature or non-differentiabilities, the hierarchical surpluses may decrease very

[6] Note that the formal iterative structure of Algorithm 1 is similar to the approach taken by Bokanowski et al. (2013) to solve the HJB equation. However, there are a couple of differences worth mentioning. First, since we aim to iteratively solve for a multivariate policy function, our adaptive refinement criterion had, as pointed out earlier, to be extended to the multivariate case, whereas Bokanowski et al. (2013) consider a single-valued function approximation. On the other hand, Bokanowski et al. (2013) allow for adaptive coarsening when iterating from one time step to the next, while we only allow for adaptive refinement.

Data: Initial guess p_{next} for next period's policy function. Approximation accuracy $\bar{\eta}$. Maximal refinement level L_{max}. Starting refinement level $L_0 \leq L_{max}$. Refinement threshold ϵ.
Result: The (approximate) equilibrium policy function p.
while $\eta > \bar{\eta}$ **do**
  Set $l = 1$, set $G \subset X$ to be the level 1 grid on X, and set $G_{old} = \emptyset, G_{new} = \emptyset$.
  while $G \neq G_{old}$ **do**
    for $g \in G \setminus G_{old}$ **do**
      Compute the optimal policies $p(g)$ by solving the system of equilibrium conditions

$$0 = \mathbb{E}\left\{E\Big(g, x_{t+1}, p\left(g\right), p_{next}\left(x_{t+1}\right)\Big) | g, p(g)\right\},$$
$$x_{t+1} \sim F\left(\cdot | g, p(g)\right),$$

      given next period's policy p_{next}.
      Define the policy $\tilde{p}(g)$ by interpolating $\{p(g)\}_{g \in G_{old}}$.
      if *($l < L_{max}$ and $\|p(g) - \tilde{p}(g)\|_\infty > \epsilon$) or $l < L_0$,* **then**
        Add the neighboring points (*sons*) of g to G_{new}.
      end
    end
    Set $G_{old} = G$, set $G = G_{old} \cup G_{new}$, set $G_{new} = \emptyset$, and set $l = l + 1$.
  end
  Define the policy function p as the sparse grid interpolation of $\{p(g)\}_{g \in G}$. Calculate (an approximation for) the error: $\eta = \|p - p_{next}\|_\infty$. Set $p_{next} = p$.
end

Algorithm 1: Overview of the crucial steps of the time iteration algorithm.

slowly and the algorithm may not stop to refine until a very high interpolation level. Thus, as one has no reasonable upper bound for the number of grid points created by the refinement procedure, we have to set a maximum refinement level.

An important detail of the implementation is the integration procedure used to evaluate the expectations operator. In case of the IRBC application, the expectation term in Equ. (1.8) has to be evaluated by integrating over the normally distributed productivity shocks. As we want to focus on the grid structure, we chose an integration rule that is simple and fast. In particular, we use a simple monomial rule that exploits the normality assumption and uses just two evaluation points per shock, thus $2(M+1)$ points in total (see, Judd (1998), with references therein). As we apply the same rule along the time iteration algorithm as well as for the error evaluation, this choice factors out the question of finding integration procedures that are both accurate and

efficient. In principle integration could also be carried out using an (adaptive) sparse grid (Bungartz and Dirnstorfer, 2003; Ma and Zabaras, 2009), yet not over the same space that the policy functions are interpolated on. Therefore, we view integration as a problem that is orthogonal to the choice of the grid structure, and thus do not focus on it.

1.4.2 Hybrid parallelization scheme

In each step of the above time iteration procedure the updated policy function is determined using a hybrid-parallel algorithm, as shown in Fig. 1.5. We construct adaptive sparse grids by distributing the newly generated grid points via MPI within a refinement step among multiple, multithreaded processes. The points that are send to one particular compute node are then further distributed among different threads. Each thread then solves a set of nonlinear equations for every single grid point assigned to it. The set of nonlinear equations—given by (1.8)–(1.10) in our application—is solved with `Ipopt` (Waechter and Biegler, 2006), which is a high-quality open-source software for solving nonlinear programs (`http://www.coin-or.org/Ipopt/`). On top of this, we add an additional level of parallelism. When searching for the solution to the equation system at a given point, the algorithm has to frequently interpolate the function computed in the previous iteration step. These interpolations take up 99% of the computation time needed to solve the equation system. As they have a high arithmetic intensity—that is to say, many arithmetic operations are performed for each byte of memory transfer and access—they are perfectly suited for GPUs. We therefore offload parts of the interpolation from the compute nodes to their attached accelerators (cf. Sec. 1.4.3 for more details). Hence, CPU cores and the GPU device of a single node are utilized through multiple threads, and MPI is used for internode communication only.

Given the limited availability of unified multicore CPU/GPU programming models, such as OpenMP 4, and our aim to perform more aggressive manual optimizations, we decided to develop two separate code paths: GPU kernels are implemented with Thrust (Bell and Hoberock, 2011), while CPU multithreading and CPU/GPU workload partitioning is organized with Thread Building Blocks (Reinders, 2007).

1.4.3 Single node optimization and parallelization

In order to solve the IRBC model in minimal time, we aim to utilize the computational resources available on each compute node in an efficient manner. Targeting primarily hybrid CPU+GPU compute nodes, our general strategy is to map the homogeneous

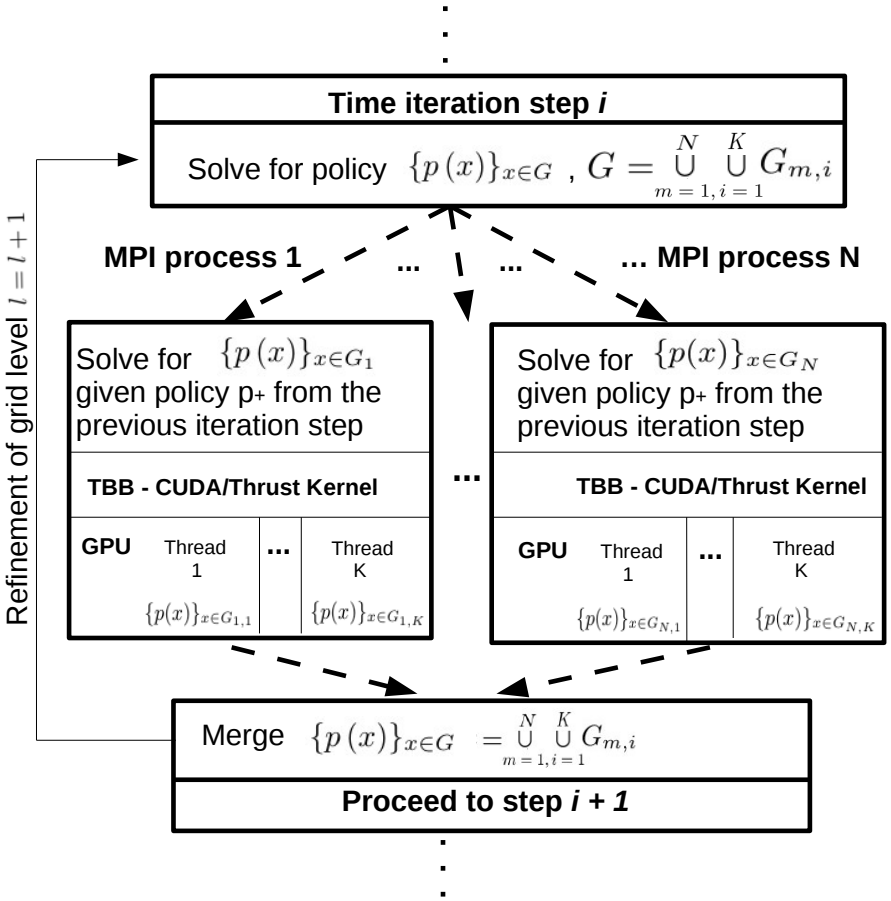


Figure 1.5: Schematic representation of the hybrid parallelization of a time iteration step presented in Algorithm 1 and Sec. 1.4.2. Each MPI process is using TBB and a CUDA/Thrust kernel for the function evaluation.

workload onto a combination of CPU threads and GPU kernels. In this section, we explain the steps taken during IRBC code optimization. The resulting gains are then reported in Sec. 1.5.1.

Explicit programming of mathematical notations “as is” is an essential starting point for any scientific application. The optimizations the compiler is able to perform are, however, not always of the same quality as manual math expressions folding. The following basic source transformations increased the odds of getting a reasonably efficient binary code:

- eliminate floating-point divisions
- eliminate redundant branching
- eliminate redundant computations, conserving the memory throughput.

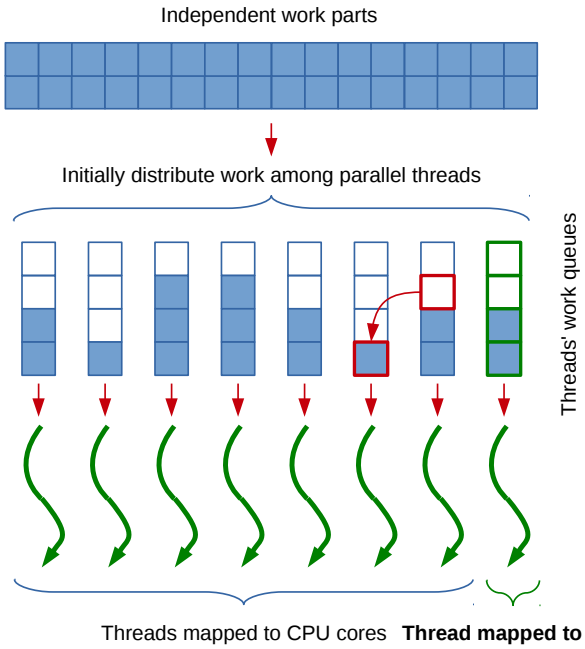


Figure 1.6: Hybrid multithreading with Intel TBB: $(N-1)$ threads on CPU, 1 thread – for GPU; TBB balances workloads automatically using "work stealing"

Above's code changes often work in combination. For instance, precomputing directly usable index arrays made it possible to eliminate branching in the section of the code that computes the linear basis functions. The GPU version of the policy function evaluation is implemented with Thrust's `transform_reduce`. Arrays of read-only indices are transposed to fitful coalescing requirements. One GPU thread handles 4 consecutive indices that are loaded with a single `int4` (`LD.128`) memory transaction.

Multithreading on a single node CPU is implemented with Thread Building Blocks (TBB). Moreover, one of the TBB-managed threads is exclusively used for the GPU kernels dispatch. CPU and GPU threads leverage TBB's automatic workload balancing based on stealing tasks from slower workers (see Fig. 1.6). Our code performs floating-point computations in double precision. Modern SIMD CPUs are able to handle 4 double values in a single instruction using 32-byte AVX vector registers (see Fig. 1.7). The use of AVX not only increases effective arithmetic throughput in compute-bound applications, but also results in a better register allocation and reduced cache pressure in memory-bound applications. Therefore, vectorization is preferred, regardless the class of application. Scalar-vector code transition is mostly straight-forward; however, special attention should be paid to element-wise accesses within AVX vectors: they are expensive and should be avoided. In other words, the most efficient vectorization could be achieved if the code is fully vectorized from loading inputs to storing outputs,

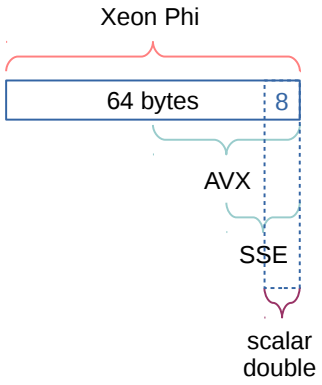


Figure 1.7: Vector registers on modern CPUs: a scalar program can utilize only 1/4 of computational parallelism on AVX-enabled CPUs, e.g. the SandyBridge.

without being interrupted by scalar regions. Specifically for the AVX version of the policy function evaluation, we had to adapt arithmetics, 0-comparison branch and `abs` function all to handle 4 grid points at once, which resulted into 2.1× overall performance improvement.

Thrust's `transform_reduce` implementation allocates the GPU memory buffer to keep partial sums, which is not exposed to the user. The reuse of this buffer across subsequent reductions with equal parameters is not supported. As result, Thrust accompanies each reduction with an allocation and deallocation of a small memory region. We eliminated the overhead of redundant allocations/deallocations by providing alternative Thrust-aware `cudaMalloc`/`cudaFree` implementations that allocate the requested buffer one time, and then pass the existing allocation to all subsequent requests, without freeing it. This modification reduced the total execution time by approximately 7%, as shown in Fig. 1.8.

Most of the read-only data used by the policy function evaluation is shared across all invocations. The only exception is the x coordinate vector, whose size equals to the dimensionality of economic problem, typically a small value. Given that PCI-E data transfer reaches optimal bandwidth for vector sizes of at least several megabytes, x-vector copying always has low efficiency. One simple method to eliminate this small inefficient vector `cudaMemcpy` is to append the vector elements directly to the kernel argument list (as scalars):

```
kernel<<<grid,block>>>(..., x[0], x[1], ..., x[DIM - 1]);
```

Scalar elements are then assembled back into the local array in order to keep simple indexing. This technique requires that we hard-coded the dimensionality of the economic problem into the kernel source. Knowing its value, the compiler will very

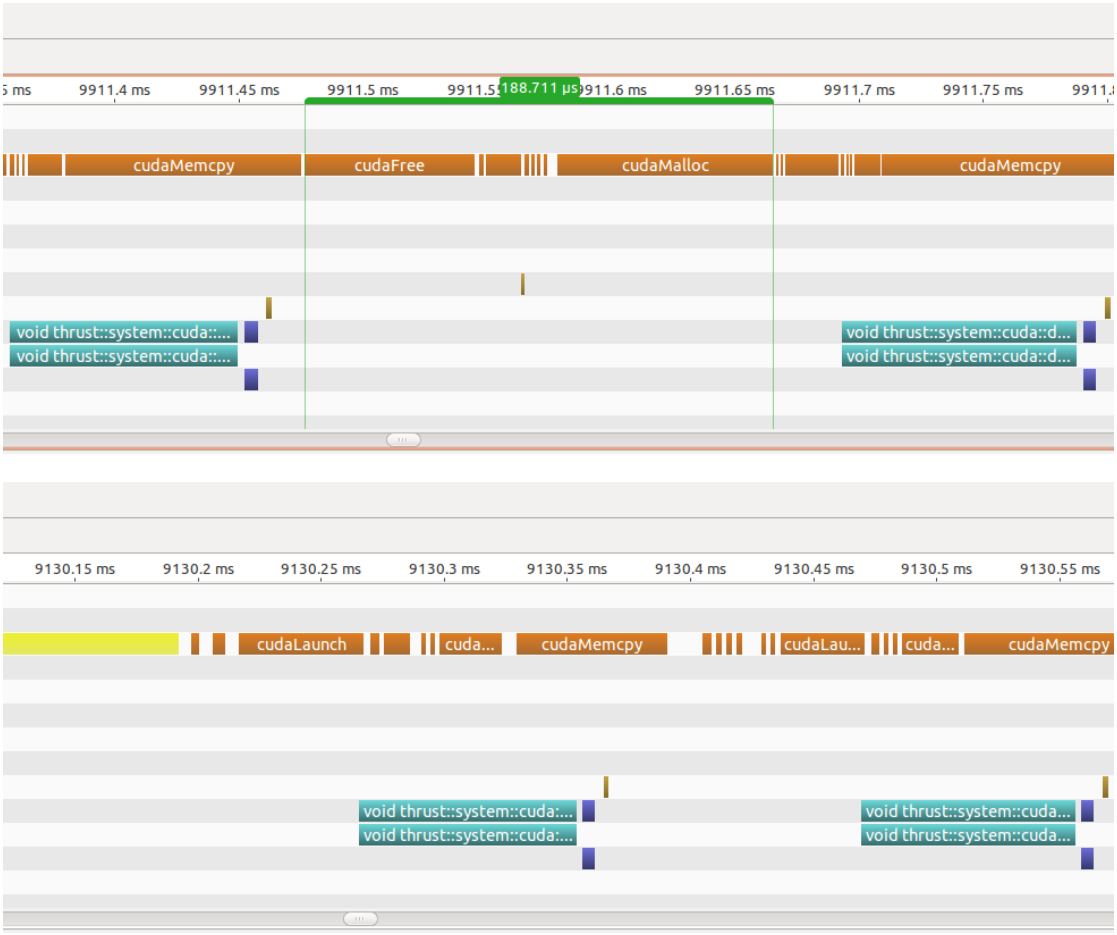


Figure 1.8: Eliminating Thurst's GPU scratch space memory allocation overhead: original implementation profile (top), reusing single-time allocation across all Thrust kernel calls (bottom). The gap between kernel launches is approximately 2 times smaller, thanks to elimination of `cudaMalloc`/`cudaFree` (green range).

likely perform complete unrolling of the corresponding loop both in the CPU and GPU versions, resulting in less branching. The local array will be mapped onto registers. Our implementation deploys JIT-compilation to dynamically compile CPU/GPU kernels, hard-coding the required dimension value, which could be scripted in a number of ways. Our implementation uses C macros for the x-vector manipulations, invokes the compiler from the running program and loads the compiled object as a shared library (`dlopen`/`dlsym`). This saves on the time of separate host-device memory transfers and leads to a speedup of about 15%. On the downside, the hard-coded kernels must be generated during program runtime, inducing some overhead from the compilation and disk I/O. The quantitative impact of all optimizations discussed here are summarized in Sec. 1.5.1 and Figure 1.9.

1.5 Numerical experiments

For our scaling experiments, we consider an 8-dimensional economic problem with four countries in Sec. 1.5.1 and 1.5.2, and four to 8-dimensional models in Sec. 2.5.4. [7] In this section, we report on the single node performance achieved by the various code optimizations described in Sec. 1.4.3. Moreover, we evaluate the strong scaling efficiencies of the IRBC model on the new Cray XC30 "Piz Daint" system. Finally, we discuss solutions to the IRBC models and show how adaptive sparse grids can speed up the computations.

We deploy the IRBC model on the 5,272-node Cray XC30 "Piz Daint" system installed at Swiss National Supercomputing Centre (CSCS). Cray XC30 compute nodes combine 8-core Intel Xeon E5-2670 (SandyBridge) CPUs with 1× NVIDIA Tesla K20X GPU. The IRBC model is compiled with GNU compilers and CUDA Toolkit 5.0.

1.5.1 Single node performance

To give a measure of how the various optimizations discussed in Sec. 1.4.3 impact the performance of the time iteration code on a single CPU thread, GPU and entire node, we evaluated the second refinement levels of a single time step from the IRBC model outlined before. This instance consists of 128 grid points, 1,152 variables and constraints. The results are summarized in Fig. 1.9. and indicate a total speedup of about 30× when going from the naive single CPU thread implementation to a more efficient version utilizing both CPU and GPU resources of the entire node. Most notably, we can see that a single-threaded GPU is about 12× faster than a single-threaded CPU, leading to an overall speedup of ~50% when the entire node is utilized in a multi-threaded mode[8] (see Fig. 1.9).

[7] Note that in computational economics, there are no standard baseline tests such as, for instance, the Sedov-–Taylor blast wave test in physics (Landau et al., 2007). What comes closest to being a standard high-dimensional test case is the the IRBC model used in Den Haan et al. (2011). However, to demonstrate the potential of adaptive grids, we include irreversibility constraints that make the model much harder to solve (cf., Sec. 1.2.2). Solving dynamic economic models with such constraints in high dimensions was not possible before and there is thus no baseline to compare to.

[8] Note that Rabitz and Alis (1999) observed in their examples speedups one to two orders of magnitude larger than the one observed here when invoking GPUs for function evaluations on sparse grids. However, their results are not directly comparable to ours due to four reasons: first they compare a so-called "iterative" sparse grid evaluation to a "recursive" one that serves as baseline. We, on the other hand, compare a multi-threaded algorithm to a single-threaded one, where the function evaluation is performed in an "iterative" fashion, similar to Rabitz and Alis (1999). Secondly, their scaling experiments are based on a test case where tens of thousands of function evaluations can be performed at once, whereas we can only perform a couple of hundred function evaluations per GPU invocation. Third, their most significant speedups were observed in single presicion computations, whereas we have to run in double-precision mode. Finally, Rabitz and Alis (1999) were using different hardware

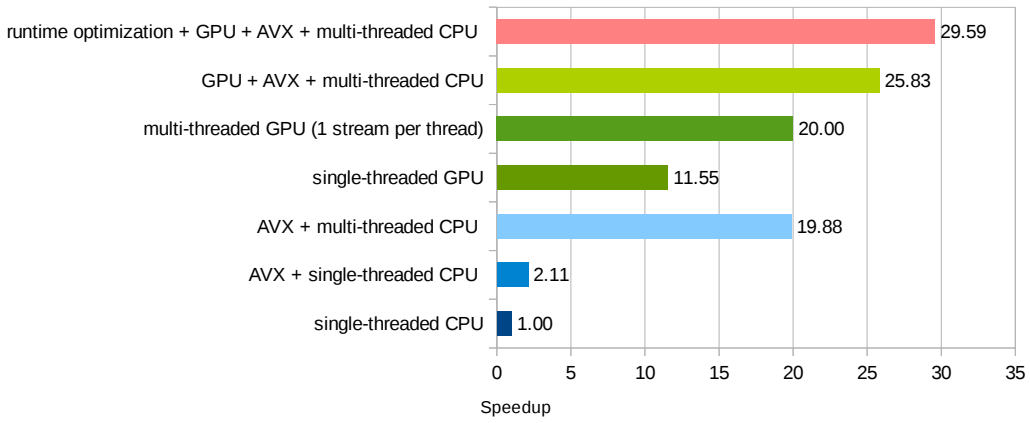


Figure 1.9: Comparison of walltimes for different IRBC model code variants on a single node of Piz Daint (speedup against scalar serial version). Benchmarked configuration: 8-dimensional model with 128 grid points and 1,152 variables and constraints.

1.5.2 Strong Scaling

We now report strong scaling efficiency of our code. The test problem is again a single timestep of an 8-dimensional economic problem with 4 countries. In order to provide a consistent benchmark, we used a nonadaptive classical sparse grid of refinement level 6. This instance has a total of 510,633 variables and constraints per time step [9]. The economic test case was solved with increasingly larger numbers of nodes (from 1 to 2,048 nodes). Fig. 1.10 shows the execution time and scaling on different levels and their ideal speedups. We used 1 MPI processes per multi-threaded Intel SandyBridge node, of which each offloads part of the function evaluation to the K20x GPU (cf. Sec. 1.4.2 and Fig.1.5). For this benchmark, the code scales nicely up to the order of 256 to 512 nodes. Thus, combined with the speedup gains due to TBB, the GPU and the code optimizations reported in Secs. 1.4.3 and 1.5.1, we attain an overall speedup of more than three orders of magnitude for our benchmark. The dominant limitation to the strong scaling stems from the fact that within the first few refinement levels, the ratio of "points to be evaluated to MPI processes" is often smaller than one with increasing node numbers, i.e., there are MPI processes idling, as can be seen in Fig. 1.10. Moreover, the workload sometimes may be unbalanced in the case of large node numbers in a sense that, e.g. one MPI process gets 2 points to work on, while a second one obtains only 1 point to work on. The better parallel efficiency on the higher refinement levels is due to the fact we have many more points

where for instance GPUs were attached to one compute node.

[9] The sparse grid under considerartion consists of 56,737 points that each hold 9 variables.

available on this refinement level, so the workload is somewhat fairer distributed among the different MPI processes. Thus, strong scaling efficiencies will be much better for higher-dimensional models ($d > 8$), as the number of newly generated grid points grows faster with increasing refinement levels. In a 24-dimensional model, for example, the points of a fixed sparse grid grow with the refinement level by 48, 1,152, 18,496, and 224,304 points, i.e. the ratio of grid points to be evaluated in the individual refinement levels per MPI process is considerably larger compared to our example.

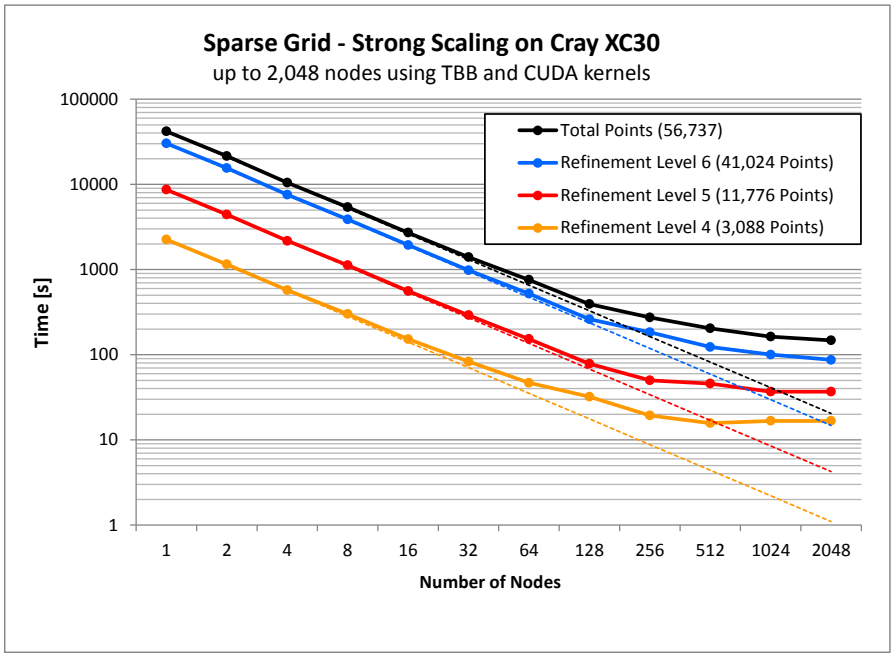


Figure 1.10: Strong scaling plots on Piz Daint for an IRBC model using 6 levels of grid refinements and in total 56,737 points. "Total" shows the entire simulation time up 2,048 nodes. We also show execution times for the computational subcomponents on different refinement levels, e.g, for refinement level 6 using 41,024 points, refinement level 5 using 11,776 points, and refinement level 4 using 3,088 points. Dotted lines show ideal speedups.

1.5.3 Convergence of time iteration

In order to gain an understanding of how the adaptivity in our algorithm can speed up computations in nonsmooth economic problems, we compare adaptive and non-adaptive solutions of the IRBC model with binding constraints outlined in Sec. 1.2.

For this purpose, we define the L_∞- and the L_2-error as (cf. Algorithm 1)

$$L_\infty = \max_{i=1,\dots,\Theta} |p(x_i) - p_{next}(x_i)|, \tag{1.27}$$

and

$$L_2 = \frac{1}{\Theta}\left(\sum_{i=1}^{\Theta}\left(p(x_i) - p_{next}(x_i)\right)^2\right)^{\frac{1}{2}}, \tag{1.28}$$

i.e., we interpolate the two consecutive policy functions p and p_{next} at $\Theta = 10,000$ test points that were randomly generated from a uniform distribution over the state space. In Fig. 1.11, we compare the decaying L_2 and L_∞- error for a complete simulation of an 8-dimensional model, once run with a fixed sparse grid of level 7 (refinement level $L_{max} = 6$), and once run with an adaptive sparse grid of a refinement threshold $\epsilon = 0.01$ and a maximum refinement level of six. [10] It is apparent from Fig. 1.11 that convergence of the time iteration algorithm is rather slow. This is to be expected, as time iteration has, at best, a linear convergence rate.[11] Fig. 1.11 also shows that adaptive sparse grids are much more efficient in reducing the approximation errors in our model, as they put additional resolution where needed, while not wasting resources in areas of smooth variation. The adaptivity in this particular benchmark reduces the size of the grid by more than one order of magnitude compared to a classical sparse grid (see Tab. 1.2). Since the interpolation time on sparse grids grows faster than linearly with the number of points (see, e.g., Murarasu et al. (2011)), the walltime is in this experiment reduced by approximately two orders of magnitude. Hence, adaptive sparse grids introduce an additional layer of sparsity on top of the a priori sparse grid structure of the classical sparse grid. In Fig. 1.12, we illustrate this by displaying 2-dimensional projections of a fixed and an adaptive sparse grid. Thus, we are able to locally mimic an interpolant that is of very high order where needed, while in other regions, only a few points are invested. This is contrasted by non-adaptive methods which can only provide one resolution over the whole domain. This feature is illustrated in Tab. 1.2, where we compare the number of grid points for different grid types and dimensions. With adaptive sparse grids, we spend at least one order of magnitude fewer points compared to ordinary sparse grids in order to reach the same accuracy of the interpolant.

Let us now turn our attention to the economic interpretation of our global solutions to the nonsmooth IRBC models. While the L_∞ and L_2 errors displayed in Fig. 1.11

[10] Note that $\epsilon = 0.01$ and $L_{max} = 6$ in this example were chosen such that the simulations satisfy the order of accuracy desired.

[11] Assuming strong concavity of the return function (in either the state or the choice), Maldonado and Svaiter (2007) show that the policy function of a stochastic dynamic programming problem is Hoelder continuous in the value function and that its convergence rate is the square root of the discount factor.

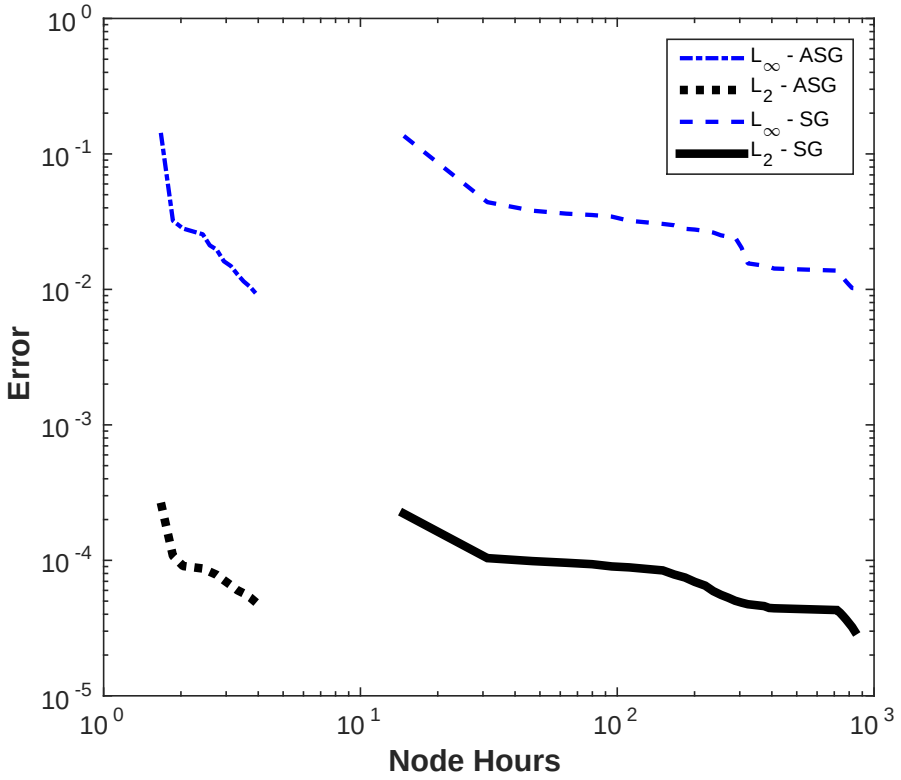


Figure 1.11: Comparison of the decreasing maximum (L_∞) and L_2-error for conventional (SG) and adaptive sparse grid (ASG) solutions to the $2N = 8$-dimensional nonsmooth IRBC model as a function of node hours on a Cray XC30. We compute these errors for ten thousand points drawn from a uniform distribution over the state space.

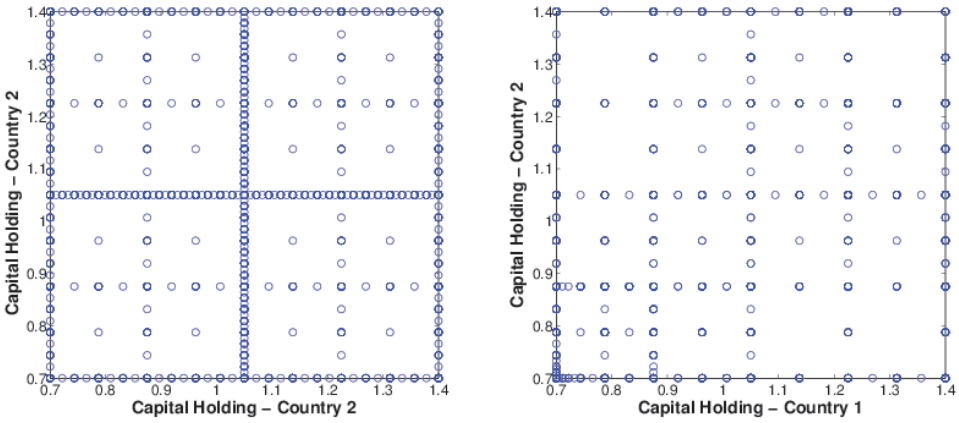


Figure 1.12: This figure displays 2-dimensional projections of two different grids. The left one is from a classical sparse grid, the right one from an adaptive grid of comparable accuracy. Both grids were generated in the course of running a $2M = 4$-dimensional simulation. The x-axis shows capital holding of country 1, the y-axis shows capital holding of country 2, while the two other axis of the four dimensional grid (productivities of the two countries) are kept fixed at their mean values.

give an indication about the rate of convergence, we are still lacking a measure of how accurate these solutions are.

To give an economic interpretation to the accuracy of the computed solutions, recall that the policy functions have to satisfy a set of equilibrium conditions. Therefore, it is common practice in economics (see, e.g., Kollmann et al. (2011)) to compute (unit-free) errors in the $M+1$ equilibrium conditions. As in our model, these conditions often mainly consist of *Euler equations*[12], the respective errors are therefore called

[12]In economics, the term *Euler equation* has a specific meaning different from its meaning in fluid dynamics. Here, Euler equations are first-order optimality conditions in dynamic equilibrium models. These (difference or differential) equations thus characterize the evolution of economic variables along

Table 1.2: Average Euler errors for an adaptive sparse grid solution of the nonsmooth IRBC model with increasing dimension. For all dimensions, we use a refinement threshold $\epsilon = 0.01$ to further refine the grid up to a maximum refinement level of six. The number of required grid points of the adaptive sparse grid solution ($|V_7^{AS}|$) is contrasted by the corresponding grid size of the classical sparse grid ($|V_7^S|$) and a full tensor product grid ($|V_7|$). All Euler errors are reported in $\log_{10}$-scale.

Dimension d	Full grid $\lvert V_7\rvert$	$\lvert V_7^S\rvert$	$\lvert V_7^{AS}\rvert$	Euler error
4	17,850,625	2,929	512	-2.88
6	75,418,890,625	15,121	1,679	-2.73
8	318,644,812,890,625	56,737	4,747	-2.66

Euler errors. In our IRBC model, there is one Euler equation error for each country $j \in \{1, ..., M\}$:

$$\left[\beta \mathbb{E}_t \left\{\lambda_{t+1}\left[a^j_{t+1} A\kappa (k^j_{t+1})^{\kappa-1} + (1-\delta) + \frac{\phi}{2} g^j_{t+2}\left(g^j_{t+2}+2\right)\right] - \mu^j_{t+1}(1-\delta)\right\}\right] \cdot \left[\lambda_t\left(1+\phi g^j_{t+1}\right)\right]^{-1} - 1. \quad (1.29)$$

In addition, there is one additional error from the aggregate resource constraint:

$$\sum_{j=1}^{M}\left(a^j_t \cdot A \cdot (k^j_t)^\kappa + k^j_t \cdot \left((1-\delta) - \frac{\phi}{2}\cdot (g^j_{t+1})^2\right) - k^j_{t+1} - \left(\frac{\lambda_t}{\tau_j}\right)^{-\gamma^j}\right) \cdot \left(\sum_{j=1}^{M}\left(a^j_t \cdot A \cdot (k^j_t)^\kappa + k^j_t \cdot \left(-\frac{\phi}{2}\cdot (g^j_{t+1})^2\right)\right)\right)^{-1}. \quad (1.30)$$

In case of the IRBC model with irreversible investment there is one additional complication. Denoting the error defined in Eq. 1.29 by EE^j and defining the percentage violation of the irreversibility constraint by

$$IC^j \equiv 1 - \frac{k^j_{t+1}}{k^j_t \cdot (1-\delta)} \quad (1.31)$$

the error is now given by

$$\max\left(EE^j, IC^j, \min\left(-EE^j, -IC^j\right)\right). \quad (1.32)$$

The economic reason for this functional form of the error is that the optimal level of investment might be negative and thus not feasible due to the irreversibility constraint. In this case, the violation or slackness in the constraint (that is, IC^j or $-IC^j$) has to be taken into account when calculating the economic error (see Brumm and Scheidegger (2014) for a more detailed explanation of (1.32)). To calculate the $M+1$ errors at a given point in the state space, we evaluate the terms in (1.29) to (1.32) using the computed equilibrium policy function for calculating both today's policy and next period's policy. To generate the statistics on Euler errors reported below we then proceed as follows. We compute the $M+1$ errors for all points in the state space that are visited for ten thousand points drawn from a uniform distribution over the state space. We then take the maximum over the absolute value of these errors, which results in one error for each point. Finally, we compute the average over all points and report the result in $\log_{10}$-scale.

an equilibrium path.

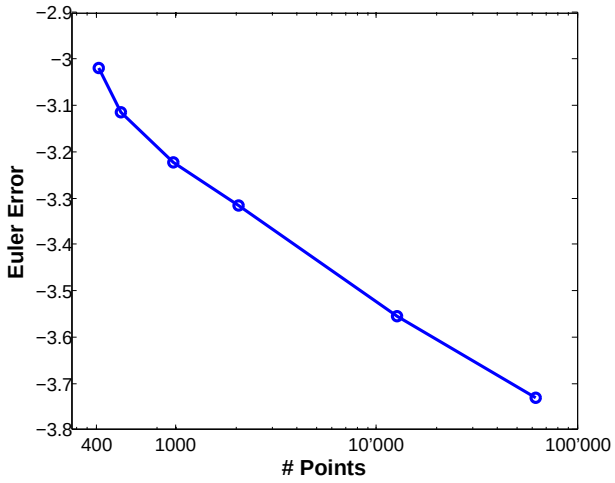


Figure 1.13: Comparison of the average Euler error (in $\log_{10}$-scale) for adaptive sparse grid solutions to the $2M = 4$-dimensional nonsmooth IRBC model as a function of the number of gridpoints resulting from varying the refinement threshold $\epsilon = \{0.02, 0.01, 0.005, 0.0025, 0.001, 0.0005\}$.

Tab. 1.2 reports the average Euler errors for adaptive sparse grids of a fixed refinement threshold $\epsilon = 0.01$, a maximum refinement level $L_{max} = 6$, and increasing dimensionality. We find that the accuracy moderately depends on the dimension of the model. There seems to be a downward trend in the average Euler error. However, this behavior is not surprising. One has to keep in mind that kinks that appear in our $2M$-dimensional models are in fact $(2M-1)$-dimensional hypersurfaces. Thus, such objects become much harder to approximate as M increases. Moreover, the maximum refinement $L_{max} = 6$ is binding for dimensions six and eight, while it is not reached for dimension four. Therefore, with a larger L_{max} the errors for higher dimensions would slightly improve relative to the four-dimensional case. More importantly, the Euler errors can be improved substantially by lowering the refinement threshold ϵ (and increasing the maximum refinement level $L_{max} = 6$)

To show how powerful the adaptive grids are in reducing Euler errors we focus on the four-dimensional case and set L_{max} such that it is never reached. In Figs 1.13 and 1.14, the average Euler errors for adaptive sparse grid solutions of the 4-dimensional nonsmooth IRBC model of different refinement thresholds ϵ are reported. These figures show that the error converges roughly linearly with respect to $1/\epsilon$ and the number of points. The smaller the chosen refinement threshold, the larger the maximum refinement level reached and the larger the number of gridpoints.

To sum up, the hybrid parallel time iteration algorithm presented in this paper can successfully compute global solutions of high-dimensional, (nonsmooth) dynamic

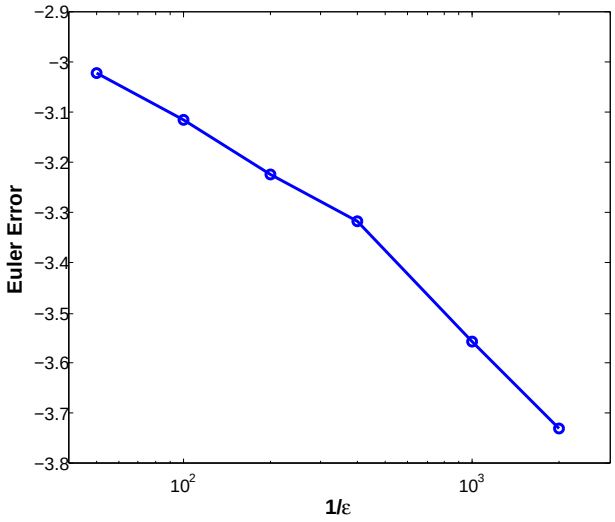


Figure 1.14: Average Euler error (in $\log_{10}$-scale) for adaptive sparse grid solutions to the $2M = 4$-dimensional nonsmooth IRBC model as a function of the inverse of the refinement threshold $1/\epsilon$.

stochastic economic models of a level of complexity not possible before—models with occasionally binding constraints have so far been tractable only in low-dimensional cases (Brumm and Grill, 2014; Christiano and Fisher, 2000; Hintermaier and Koeniger, 2010).

1.6 Conclusion

Solving complex high-dimensional dynamic stochastic economic models numerically in reasonable time—i.e., in hours or days—imposes a variety of problems. In this work, we developed an effective strategy to address these challenges by combining adaptive sparse grids, time iteration methods, and high-performance computing in a powerful toolkit that can handle a broad class of models up to a level of heterogeneity not seen before.

First, using (adaptive) sparse grids alleviates the *curse of dimensionality* imposed by the heterogeneity of the economic models. Second, they can successfully resolve non-smooth policy functions, as they put additional resolution where needed, while not wasting resources in areas of smooth variation. High-performance computing on the other hand enters the picture when we aim to minimize the time-to-solution. By exploiting the generic structure common to many dynamic economic models, we implemented a hybrid parallelization scheme that uses state-of-the-art parallel computing paradigms. It minimizes MPI interprocess communication by using TBB

and partially offloads the function evaluations to GPUs.

Numerical experiments on Piz Daint (Cray XC30) at CSCS show that our code is very scalable and flexible. In the case of our intermediate-sized, 8-dimensional IRBC benchmark, we found very good strong scaling properties up to the order of 256 to 512 nodes. The dominant limitation to the strong scaling stems from the fact that within the first few refinement levels, the ratio of "points to be evaluated to MPI processes" is often smaller than 1 with increasing node numbers, i.e. there are MPI processes idling for some time. This all suggests that our framework is very well suited for large-scale economic simulations on massively parallel high-performance computing architectures.

Acknowledgments

The authors would like to thank Felix Kübler (University of Zurich) for helpful discussions and support. Computing time on "Piz Daint" at the Swiss National Supercomputing Center was provided by a USI-CSCS allocation contract. Additionally, this work was supported by a grant from the Swiss National Supercomputing Centre (CSCS) under project ID s555.

Part II

2 Rethinking large-scale economic modeling for efficiency

We propose a massively parallelized and optimized framework to solve high-dimensional dynamic stochastic economic models on modern GPU- and KNL-based clusters. First, we introduce a novel approach for adaptive sparse grid index compression alongside a surplus matrix reordering, which significantly reduces the global memory throughput of the compute kernels and maps randomly accessed data onto cache or fast shared memory. Second, we fully vectorize the compute kernels for AVX, AVX2 and AVX512 CPUs, respectively. Third, we develop a hybrid cluster oriented work-preempting scheduler based on TBB, which evenly distributes the time iteration workload onto available CPU cores and accelerators. Numerical experiments on Cray XC40 KNL "Grand Tave" and on Cray XC50 "Piz Daint" systems at the Swiss National Supercomputer Centre (CSCS) show that our framework scales nicely to at least 4,096 compute nodes, resulting in an overall speedup of more than four orders of magnitude compared to a single, optimized CPU thread. As an economic application, we compute global solutions to an annually calibrated stochastic public finance model with sixteen discrete, stochastic states with unprecedented performance.

2.1 Introduction

Optimal taxation and the optimal design of public pension systems are classic themes in economics with obvious relevance for society. To address these questions quantitatively, dynamic stochastic general equilibrium models with heterogeneous agents are used for counter-factual policy analysis. One particular subclass is called an overlapping generation (OLG) model (Diamond, 1965). These models are essential tools in public finance since they allow for careful modeling of individuals' decisions over the life cycle and their interactions with capital accumulation and economic growth.

There are now several areas where, over the last 10 to 20 years, deterministic OLG

models have been fruitfully applied to the analysis of taxation and fiscal policy. In particular large-scale deterministic versions of the model have been applied to the "fiscal gap" (Evans et al., 2012), to "dynamic scoring of tax policies" (Auerbach and Kotlikoff, 1987), and to the evaluations of social security reforms (see, e.g., Feldstein and Liebman (2001)). It is clear, however, that to be able to address these policy-relevant questions thoroughly, uncertainty needs to be included in the basic model. Both uncertainty about economic fundamentals as in Krueger and Kubler (2006) as well as uncertainty about future policy (David S. Bizer, 1989) crucially affect individuals' savings, consumption, and labor-supply decisions and the uncertainty in the specification of the model can overturn many results obtained in the deterministic model. Moreover, uncertainty about future productivity as well as uncertainty about future taxes have first-order effects on agents' behavior. Unfortunately, when one introduces this form of uncertainty into the model, there does not exist steady-state equilibria, as the stochastic aggregate shocks affect everybody's return to physical and human capital. These effects do not cancel out in the aggregate so that the distribution of wealth across generations changes with the stochastic aggregate shock. This feature makes it difficult to approximate equilibria with many agents of different ages and aggregate uncertainty—realistic calibrations of the model lead to very-high-dimensional problems that were so far thought to be unsolvable. This explains why relatively little policy-work has been carried out using stochastic OLG models. Krueger and Kubler (2004), for example, analyze welfare implications of social security reforms in an OLG model where one period corresponds to six years, thereby reducing the number of adult cohorts and thus the dimensionality of the problem by a factor of six. Hasanhodzic and Kotlikoff (2013), on the other hand, approximate the solution of an OLG model using simulation-based methods and certainty equivalents. Their method only yields acceptable solutions for special cases and cannot be easily extended to tackle general OLG models.

This article shows how we can leverage recent developments in computational mathematics and massively parallel hardware to compute global solutions to general stochastic OLG models in relatively short times. As a test case, we have solved a 59-dimensional model with 16 discrete, stochastic states—much larger than any problem known to be addressed so far in this stream of the literature. Therefore, our methodology opens the room to address economic research questions of unprecedented realism.

In stochastic dynamic models, individuals' optimal policies and prices are unknown functions of the underlying, high dimension states and are solved for by so-called *time iteration* algorithms (see, e.g., Judd (1998)). Two major bottlenecks create difficulties in achieving a fast time-to-solution process when solving large-scale dynamic stochastic

OLG models with this iterative method, namely,

(i) in each iteration step, several economic functions need to be approximated and interpolated. For this purpose, the function values have to be determined at many points in the high-dimensional state space, and

(ii) each point involves solving a system of nonlinear equations (around 60 equations in 60 unknowns).

We overcome these difficulties by massively reducing the number of grid points required to represent the economic functions by using adaptive sparse grids (ASGs; see, e.g., Bungartz and Griebel (2004); Ma and Zabaras (2009); Pflüger (2012); Nobile et al. (2008)) as well as by compressing the ASGs only to visit points with meaningful contribution when interpolating on them. Also, the time spent in each iteration step is substantially reduced by applying massively parallel processing. Using the Message Passing Interface (MPI) (Skjellum et al., 1999), we distribute the workload—that is, the grid points, across compute nodes. The nodes can optionally be equipped with NVIDIA GPUs. Within a single node, the workload is further partitioned among CPU cores and a GPU with Intel Thread Building Blocks (TBB) (Reinders, 2007). The CPU code deploys AVX, AVX2 or AVX-512 vectorization for Sandy/Ivy Bridge, Haswell/Broadwell or Skylake/KNL, respectively, while NVIDIA GPU kernels are written in CUDA (Buck et al., 2004). This scheme enables us to make efficient use of the contemporary HPC facilities that consist of a variety of special purpose as well as general purpose hardware and whose performance nowadays can reach dozens of petaflop/s (https://www.top500.org). To sum up, the main contributions of this paper are as follows:

- Building on Brumm and Scheidegger (2017); Brumm et al. (2015), we propose a generic parallelization scheme for time iteration algorithms that aim to solve mixed high-dimensional continuous/discrete state dynamic stochastic economic models.
- We show that our parallelization approach is ideally suited for heterogeneous CPU/GPU HPC systems as well as for Intel Xeon Phi KNL clusters.
- We introduce an original compression method for ASGs that reduces computations, yet allowing partial vectorization and randomly accessed data fitting into cache or GPU shared memory.
- We present highly efficient and scalable implementations of the time iteration algorithm on the aforementioned hardware platforms.

- As an example application, we compute global solutions to 59-dimensional OLG model with 16 discrete, stochastic states with unprecedented performance.

This paper is organized as follows. In Sec. 2.2, we describe the abstract economic models we aim to solve. In Sec. 2.3, we briefly summarize the theory of ASGs. In Sec. 2.4, we embed ASG interpolation in a time iteration algorithm. Moreover, we also discuss the respective hybrid parallelization scheme as well as a novel compression method for ASGs that substantially reduces computations. In Sec. 2.5, we report on how our implementation performs and scales in solving an annually calibrated, stochastic OLG model.

2.2 Overlapping generation models

To demonstrate the capabilities of our method, we consider an annually calibrated, stochastic OLG model similarly to the one described in Krueger and Kubler (2006)—that is, agents have a model lifetime of 60 periods, each corresponding to one year of life after the age of 20. Moreover, there are $N_s = 16$ discrete states in our model that represent the economy in a variety of situations such as booms, busts as well as different tax regimes. Agents face taxes τ_l on labor income and τ_c on capital income. Tax rates change stochastically over time and are used to fund a pay-as-you-go social security system. We assume that the average retirement age is 65, and that agents receive social security payments, financed by the labor-income tax, starting at age 66. It is clear that the level of complexity listed here is needed to model for example demographic effects that are caused by retirement as well as to mimic the fact that agents choose their actions based on expectations about an uncertain future. However, taken together, this all results in a very intricate formal structure of the model.

This is an example of a broad class of models in macroeconomics and public finance and is typically solved by time iteration algorithms (see, e.g., Judd (1998)). To this end, we outline in Sec. 2.2.1 the general structure that is common to OLG models. Moreover, we briefly describe how we iteratively solve them.

2.2.1 Abstract model formulation and solution method

The formal structure[1] common to stochastic OLG models can be described as follows: The economy is populated with agents that live for A periods. Each of them can uniquely be identified by her age a, where $1 \leq a \leq A$. Let $s_t = (z_t, x_t) \in S \subset \mathbf{Z} \times \mathbf{B} \subset \mathbb{R} \times \mathbb{R}^{\mathbf{d}}$

[1] Note that we omit a detailed discussion of the OLG model, as this is beyond the scope of the paper. For a detailed review of this application, we refer to Krueger and Kubler (2006); Brumm et al. (2017).

denote the state of the economy at time $t \in \mathbb{N}$, where $\mathbf{Z}$ is a finite set of size $N_s \in \mathbb{N}$, $d = A-1$ is the dimensionality of x_t, and $\mathbf{B}$ is a $d-$dimensional rectanglular box. z_t represents a stochastic shock to the economy, e.g., to it's output, and x_t characterizes the economy in z_t. In our OLG model, it is given by

$$x_t = (K, \omega_2, ..., \omega_{A-1}) \subset \mathbb{R}^{A-1}, \tag{2.1}$$

where K is the aggregate capital and ω_i are the wealth levels of generations $i = 2$ to $i = A-1$. The actions of all agents in the economy can be represented by a *policy function* $p : S \to Y$, where Y is the space of possible *policies*. In our OLG model, the optimal policy $p : R^{N_s \cdot d} \to R^{N_s \cdot 2 \cdot d}$ maps the current state s_t into unknown asset demand functions $k_i : \mathbf{Z} \times \mathbf{B} \to \mathbb{R}$ and value functions $v_i : \mathbf{Z} \times \mathbf{B} \to \mathbb{R}$, where $i = 1, ..., A-1$, and $z \in \mathbf{Z}$. Furthermore, the evolution of the current state of the economy s_t from period t to $t+1$ is described by the state transition

$$s_{t+1} \sim \mathscr{P}\left(\cdot | s_t, p(s_t)\right), \tag{2.2}$$

where the distribution $\mathscr{P}(\cdot)$ is pre-defined and model specific. In our case, the stochastic transition of the economy from period t to $t+1$ is given by a Markov chain—that is, z_t follows a first-order Markov process with transition probability $\pi(z'|z)$. The stationary policy function p needs to be determined from *equilibrium conditions*. These conditions constitute a functional equation that the policy function p has to satisfy, namely, that for all $s_t \in S$,

$$0 = \mathbb{E}\left[f\Big(s_t, s_{t+1}, p\left(s_t\right), p\left(s_{t+1}\right)\Big) | s_t, p(s_t) \right], \tag{2.3}$$

where $f : S^2 \times Y^2 \to \mathbb{R}^{N_s \cdot d}$ represents the period-to-period equilibrium conditions of the OLG model, and where the expectation operator is taken on the the disrete shocks. This function is nonlinear because of concavity assumptions on utility and production functions. As a direct consequence, the optimal policy p solving (2.3) will also be nonlinear. Hence, approximating it only locally might provide misleading results. For such applications, we, therefore, need a global solution, that is, we need to approximate p over the entire state space S. In our work, we approximate the unknown equilibrium asset demand and value functions on an individual ASG per discrete state $z \in \mathbf{Z}$ by piecewise multilinear functions $\hat{k}_i(z, \cdot | \alpha^k)$, $\hat{v}_i(z, \cdot | \alpha^v)$ that are uniquely defined by finitely many coefficients α^k, α^v (see Sec. 2.3). In order to solve for the unknown coefficients, we require that the functional equations of the OLG model (see (2.3) and Krueger and Kubler (2006); Brumm et al. (2017)) hold exactly at M grid points $x_{i=1,...,M} \in \mathbf{B}$ per discrete state z.

Our computational strategy to solve the OLG model is to search for a recursive equi-

librium (see, e.g., Stokey et al. (1989a))—that is, a time-invariant policy function p by using a time iteration algorithm (see, e.g., Judd (1998)). The sequential version of this algorithm is summarized in Alg. 2 and is based on the following heuristic: solve the equilibrium conditions of the model for today's policy $p : S \to Y$ taking as given an initial guess for the function that represents next period's policy, p_{next}; then, use p to update the guess for p_{next} and iterate the procedure until numerical convergence is reached. As a practical consequence, we need to compute many successive approxi-

Data: Initial guess for $p = \big(p(z=1), ..., p(z=N_s)\big)$. Convergence tolerance tol.
Result: The time-invariant policy function p.
while $\epsilon > tol$ **do**
  $p_{next} \leftarrow p$.
  for $z = 1; z \leq N_s; z = z+1$ **do**
    approximate $p(z)$ by solving (2.3) at M grid points **given** p_{next}.
  end
  $\epsilon = \|p - p_{next}\|$.
end

Algorithm 2: Time iteration algorithm.

mations of p that rely on interpolating on p_{next}. To do this efficiently, we employ ASGs (see Sec. 2.3) in combination with a hybrid parallelization scheme (see Sec. 2.4.1) as well as a novel ASG compression scheme (see Sec. 2.4.2).

2.3 Basics on adaptive sparse grids

Our method of choice to tackle the numerical issues that arise from the nature of the high-dimensional state space as described in Sec. 2.2, namely, the repeated construction and evaluation of multivariate policy and value functions (see (2.3)) are ASGs. In this section, we summarize its basics. For thorough derivations, we point the reader, e.g., to Bungartz and Griebel (2004); Garcke and Griebel (2012).

We consider the representation of a piecewise d-linear function $f : \Omega \to \mathbb{R}$ for a certain mesh width $h_n = 2^{1-n}$ with some discretization level $n \in \mathbb{N}$. As we aim to discretize Ω, we restrict our domain of interest to the compact sub-volume $\Omega = [0,1]^d$, where d in our case is the dimensionality of the OLG model. This situation can be achieved for most other domains by re-scaling and possibly carefully truncating the original domain. In order to generate an approximation u of f, we construct an expansion

$$f(\vec{x}) \approx u(\vec{x}) := \sum_{j=1}^{N} \alpha_j \phi_j(\vec{x}) \tag{2.4}$$

with N basis functions ϕ_j and coefficients α_j. We use one-dimensional hat functions

$$\begin{aligned} &\phi_{l,i}(x) \\ &= \begin{cases} 1, & l = i = 1, \\ \max(1 - 2^{l-1} \cdot \left| x - x_{l,i} \right|, 0), & i = 0, ..., 2^{l-1},\ l > 1, \end{cases} \end{aligned} \tag{2.5}$$

which depend on a level $l \in \mathbb{N}$ and index $i \in \mathbb{N}$. The corresponding grid points are distributed as

$$x_{l,i} = \begin{cases} 0.5, & l = i = 1, \\ i \cdot 2^{1-l}, & i = 0, ..., 2^{l-1},\ l > 1, \end{cases} \tag{2.6}$$

and are depicted in Fig. 2.1. We use a sparse grid interpolation method that is based on a hierarchical decomposition of the underlying approximation space. Hence, we next introduce, hierarchical index sets I_l:

$$I_l := \begin{cases} \{i = 1\}, & \text{if } l = 1, \\ \{0 \le i \le 2,\ i \text{ even}\} & \text{if } l = 2, \\ \{0 \le i \le 2^{l_t - 1},\ i \text{ odd}\} & \text{else}, \end{cases} \tag{2.7}$$

that lead to hierarchical subspaces W_l spanned by the corresponding basis $\phi_l := \{\phi_{l,i}(x), i \in I_l\}$. Fig. 2.1 shows the basis functions up to level 3. The hierarchical basis functions extend to the multivariate case by using tensor products:

$$\phi_{\vec{l},\vec{i}}(\vec{x}) := \prod_{t=1}^{d} \phi_{l_t,i_t}(x_t), \tag{2.8}$$

where $\vec{l}$ and $\vec{i}$ are multi-indices, uniquely indicating level and index of the underlying one-dimensional hat functions for each dimension. They span the multivariate subspaces by

$$W_{\vec{l}} := \text{span}\{\phi_{\vec{l},\vec{i}} : \vec{i} \in I_{\vec{l}}\} \tag{2.9}$$

with the index set $I_{\vec{l}}$ given by a multidimensional extension to (2.7):

$$I_{\vec{l}} := \begin{cases} \{\vec{i} : i_t = 1, 1 \le t \le d\} & \text{if } l = 1, \\ \{\vec{i} : 0 \le i_t \le 2,\ i_t \text{ even}, 1 \le t \le d\} & \text{if } l = 2, \\ \{\vec{i} : 0 \le i_t \le 2^{l_t - 1},\ i_t \text{ odd}, 1 \le t \le d\} & \text{else}. \end{cases} \tag{2.10}$$

The space of piecewise linear functions V_n on a Cartesian grid with mesh size h_n for a given level n is then defined by the direct sum of the increment spaces (cf. (2.9)):

$$V_n := \bigoplus_{|l|_\infty \le n} W_{\vec{l}}, \quad |l|_\infty := \max_{1 \le t \le d} l_t. \tag{2.11}$$

The interpolant of f, namely, $u(\vec{x}) \in V_n$, can now uniquely be represented by

$$f(\vec{x}) \approx u(\vec{x}) = \sum_{|l|_\infty \leq n} \sum_{\vec{i} \in I_{\vec{l}}} \alpha_{\vec{l},\vec{i}} \cdot \phi_{\vec{l},\vec{i}}(\vec{x}). \tag{2.12}$$

Note that the coefficients $\alpha_{\vec{l},\vec{i}} \in \mathbb{R}$ are commonly termed hierarchical surpluses. They are merely the difference between the function values at the current and the previous interpolation levels. For a sufficiently smooth function f the asymptotic error decays as $\mathcal{O}\left(h_n^2\right)$ but at the cost of spending $\mathcal{O}\left(h_n^{-d}\right) = \mathcal{O}\left(2^{nd}\right)$ grid points, thus suffering the curse of dimensionality (Bellman, 1961). As a consequence, the question that needs to be answered is how we can construct discrete approximation spaces that are better than V_n in the sense that the same number of invested grid points leads to a higher order of accuracy. Luckily, for functions with bounded second mixed derivatives, it can be shown that the hierarchical coefficients rapidly decay, namely, $|\alpha_{\vec{l},\vec{i}}| = \mathcal{O}\left(2^{-2|\vec{l}|_1}\right)$. Hence, the hierarchical subspace splitting allows us to select those $W_{\vec{l}}$ that contribute most to the overall approximation. This can be done by an a priori selection, resulting in the sparse grid space V_n^S of level n, defined by

$$V_n^S := \bigoplus_{|\vec{l}|_1 \leq n+d-1} W_{\vec{l}}, \quad |\vec{l}|_1 = \sum_{i=1}^{d} l_t. \tag{2.13}$$

In Fig. 2.1, we depict its construction for $n = 3$ in two dimensions. V_3^S shown there consists of the hierarchical increment spaces $W_{(l_1,l_2)}$ for $1 \leq l_1, l_2 \leq n = 3$. The number of grid points required by the space V_n^S is now of order $\mathcal{O}\left(2^n \cdot n^{d-1}\right)$, which is a significant reduction of the number of grid points, and thus of the computational and storage requirements compared to the Cartesian grid space. In analogy to (2.12), a function $f \in V_n^S \subset V_n$ can now be expanded by

$$f(\vec{x}) \approx u(\vec{x}) = \sum_{|l|_1 \leq n+d-1} \sum_{\vec{i} \in I_{\vec{l}}} \alpha_{\vec{l},\vec{i}} \cdot \phi_{\vec{l},\vec{i}}(\vec{x}), \tag{2.14}$$

which contains substantially fewer terms. In the case that functions do not meet the smoothness requirements or that show distinct local features as we face in the model described in Sec. 2.2, they can still be tackled efficiently with sparse grids if spatial adaptivity is used. The classical sparse grid construction introduced in (2.13) defines an a priori selection of grid points that are optimal for functions with bounded second-order mixed derivatives. An adaptive (a posteriori) refinement can, additionally, based on a local error estimator, select which grid points in the sparse grid structure should be refined. The most common way of doing so is to add $2d$ children in the hierarchical structure with increasing grid refinement level if the hierarchical surpluses satisfy $g(\alpha) \geq \epsilon$ for a so-called refinement threshold $\epsilon \geq 0$. For more details regarding ASGs,

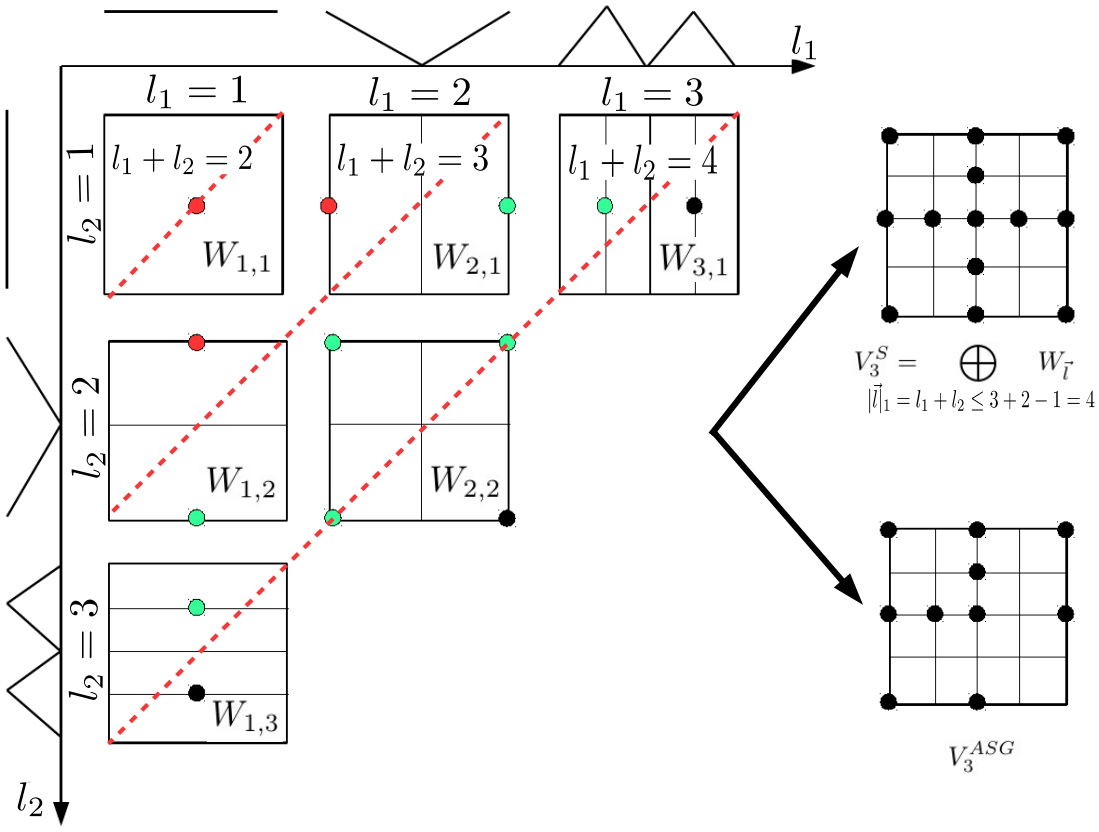


Figure 2.1: Left panel: Hierarchical increment spaces $W_{(l_1,l_2)}$ for $1 \leq l_1, l_2 \leq n = 3$ with their corresponding grid points and one-dimensional piecewise linear basis functions of levels 1, 2, and 3. Top right panel: Construction of a classical sparse grid V_3^S (see (2.13)), consisting of all the points displayed in the left figure. Note that the optimal selection of subspaces for the classical sparse grid is indicated by the dashed lines of constant $l_1 + l_2$. Bottom right panel: Construction of the ASG V_3^{ASG}. Note that the red dots in the left panel symbolically represent points that would be refined, i.e., $g(\alpha) \geq \epsilon$ holds, whereas the green ones indicate points where the grid is not further refined. The black points in the left panel are only contained in V_3^S, and not V_3^{ASG}.

we refer the reader e.g. to Bungartz and Dirnstorfer (2003); Ma and Zabaras (2009); Pflüger (2012). The lower right panel of Fig. 2.1 illustrates a qualitative example of how a sparse grid is refined adaptively, adding a second layer of sparsity to the sparse grid.

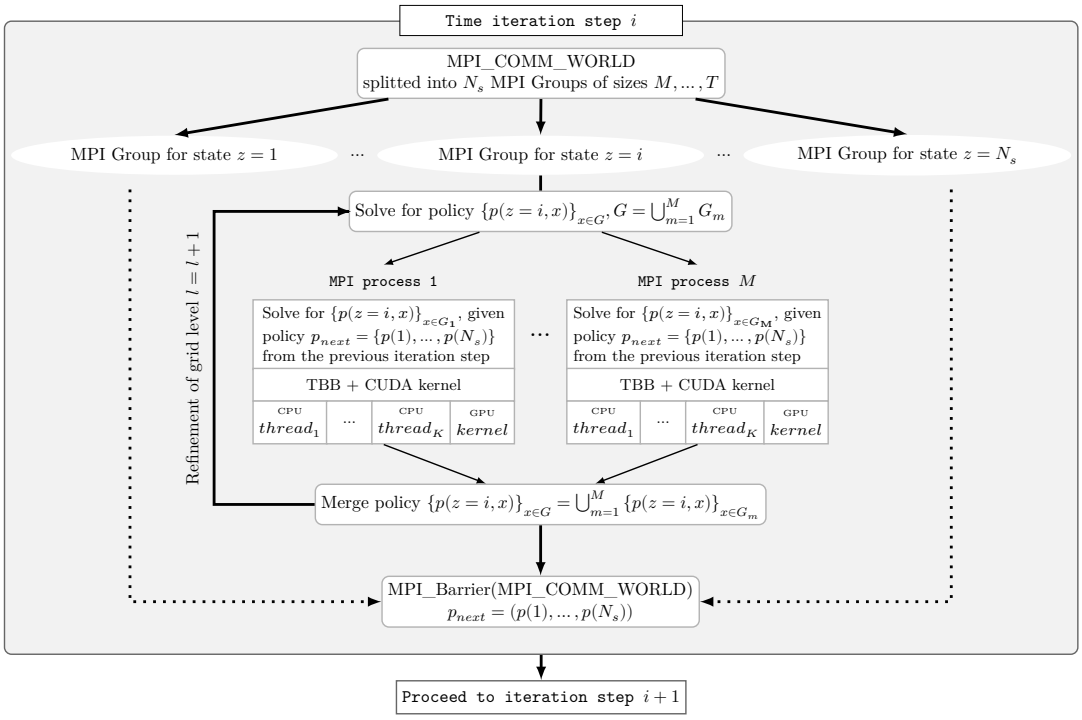


Figure 2.2: Schematic representation of the hybrid parallelization scheme in a single time step. Every `MPI` process within an `MPI_Group` is using TBB. In the case of deploying our software on hybrid CPU/GPU nodes, the interpolation on the next period's policy function p_{next} is partially offloaded to GPUs.

2.4 Parallel time iteration algorithm

We describe now how to solve the stochastic OLG model introduced in Sec. 2.2. For this reason, we implement a massively parallel version of a time iteration algorithm (see code listing 2 and Judd (1998)) for mixed high-dimensional continuous/discrete states that uses one ASG per shock $z \in \mathbf{Z}$ in each iteration step.[2] Building on Brumm et al. (2015), we parallelize this algorithm by a hybrid scheme using MPI (Skjellum et al., 1999), TBB (Reinders, 2007) (and CUDA, if a GPU is present on the compute

[2] In line with Sec. 2.2, the mixed discrete/continuous state variables of the OLG model at time t consist of $s = (z, x)$, where x has 59 dimensions, and the shock z has $N_S = 16$ possible realizations. Moreover, the policy function $p = \big(p(z=1,\cdot), \dots, p(z=16,\cdot)\big) : \mathbb{R}^{16 \cdot 59} \to \mathbb{R}^{16 \cdot 2 \cdot 59}$ maps the current state into asset demand and value functions (see Sec. 2.2). We, therefore, have to approximate 118 coefficients $\alpha = (\alpha^k, \alpha^v)$ per state z and grid point.

node), and deploy compute kernels that leverage AVX, AVX2 or AVX-512 vectorization, depending on the respective hardware platform (see Sec. 2.4.1).[3]

One significant performance bottleneck when solving large-scale economic models always lie on interpolating the previous iteration step's policy functions. When searching for the solution to the equation system at a given point for a given shock z (cf. (2.3)), the algorithm has to frequently interpolate on the policy functions p_{next} of all the $N_s = 16$ states from the previous iteration step at once. These interpolations typically take up to 99% of the computation time needed (see, e.g., Brumm and Scheidegger (2017)) to solve the nonlinear set of equations and therefore need to be carried out as rapidly as possible to guarantee a fast time-to-solution process. In our earlier work (Brumm et al., 2015), we applied a dense matrix data format that is very similar to the one proposed by Heinecke and Pflueger (2013) and for which highly optimized algorithms exist to perform the interpolation task. However, for the applications in scope here, we cannot maintain this data structure, since in contrast to Brumm et al. (2015), where we had to deal with interpolating on one single ASG of intermediate size only (around 8 continuous dimensions), we now have to be able to operate on 16 very large—that is, 59-dimensional ASGs at once (cf. Secs. 2.2 and 2.5). Keeping the aforementioned dense matrix format to store the previous timestep's policy function for the interpolation introduces a memory footprint of a non-trivial size that, in turn, would substantially slow down interpolations and thus the time-to-solution. To this end, we propose a novel, generic data compression method for ASGs (see Sec. 2.4.2).

2.4.1 Hybrid parallelization scheme on heterogeneous HPC systems

In every step i of the time iteration procedure (see Alg. 2), the policy function p is updated by using a hybrid-parallel algorithm (see Fig. 2.2). Conceptually, the top layer of parallelism is the N_s discrete states of the OLG model, which are completely independent of each other within a time step. Hence, the `MPI_COMM_WORLD` communicator is split into $N_s = 16$ sub-communicators, each of them representing an individual discrete state—that is, an independent ASG which updates its share of the total policy $p = \big(p(z = 1), \ldots, p(z = N_s)\big)$. Next, every `MPI_Group` gets a fraction from all the MPI processes available in the `MPI_COMM_WORLD` communicator assigned

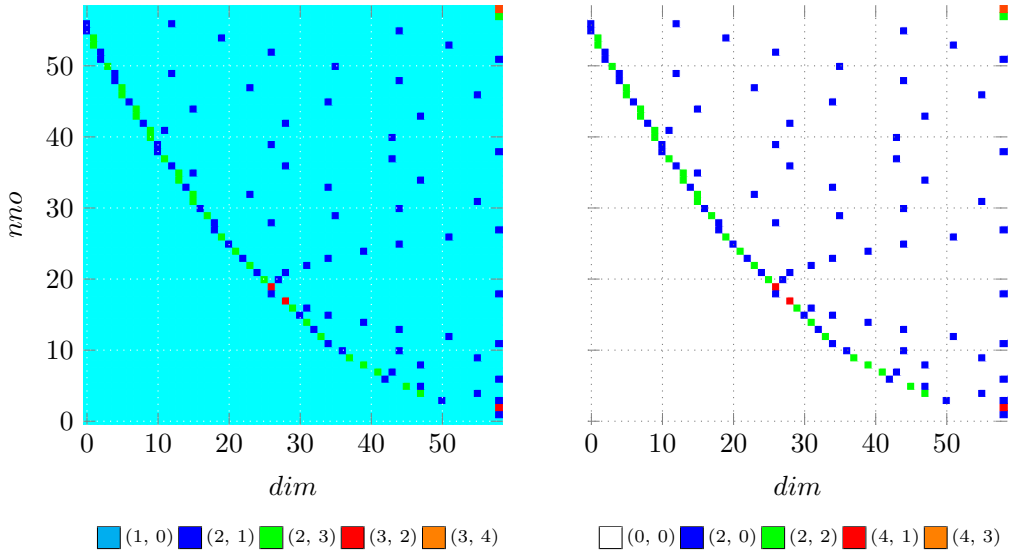
[3] Note that the developments presented in this article substantially improve over our previous work (Brumm et al., 2015). First, we extend our original code base, and it's respective parallelization scheme such that it can handle high-dimensional continuous as well as discrete, stochastic states at the same time. Second, our compute kernels now also deploy AVX2 and AVX-512 vectorization. Third, we introduce a novel data structure for operating on ASGs (see Sec. 2.4.2).

such that an optimal workload balance across different discrete states in guaranteed[4] (see the top part of Fig. 2.2). We achieve this by using the number of grid points M_z contained in $p_{next}(z)$ from the previous iteration step $i-1$ as a proxy for the demand on computational resources necessary in the current time step i. In particular, we assign the fraction $\texttt{MPI\_COMM\_SIZE}(z) = M_z / \left(\sum_{j=1}^{N_s} M_j\right)$ from the total available MPI processes to an individual state z. Inside every `MPI_Group`, an ASG is constructed in a massively parallel fashion. The points that are newly generated within a refinement level (see (2.13)) are distributed via MPI among multiple, multi-threaded processes. The points that are sent to one particular compute node are then further distributed among different threads. Multithreading on compute nodes is implemented with TBB. To guarantee efficient use of any of the compute nodes, the threads leverage TBB's automatic workload balancing based on stealing tasks from the slower workers. In general, each TBB thread has to solve an independent set of nonlinear equations for every single grid point assigned to it. These nonlinear equations (see (2.3)) are solved with `Ipopt` (Waechter and Biegler, 2006), which is high-quality open-source software for solving nonlinear programs (`http://www.coin-or.org/Ipopt/`). On top of this, we add an additional level of parallelism. When searching for the solution to the equation system at a given point for a given shock z, the algorithm has to frequently interpolate on the policy functions of all the $N_s = 16$ states from the previous iteration step at once. As they have a high arithmetic intensity—that is to say, many arithmetic operations are performed for each byte of memory transfer and access—they can leverage on SIMD AVX, AVX2 and AVX-512 instructions as well as on the massive parallelism of GPUs, depending on the hardware we deploy our code framework on. In the case of CPU/GPU nodes, we offload parts of the policy function interpolation from the compute nodes to their attached accelerators. In particular, one of the TBB-managed threads is exclusively used for the GPU kernel dispatch, as indicated in the lower part of Fig. 2.2. This GPU-dedicated thread launches the kernel and polls until it is finished. As the GPU is much more powerful than a CPU thread, the TBB scheduler will observe the "skew" of work balance, and will dynamically re-adjust the amounts of work in favor of the GPU-dedicated thread.

2.4.2 Adaptive sparse grid compression

While the primary arithmetic operations to calculate surpluses and perform interpolations on ASGs are rather simple (see Sec. 2.3), accessing the data requires most of the computing time, which emphasizes the importance of efficient data structures. Depending on the target hardware platform, the most widespread techniques for

[4] Note that the overhead of invoking an `MPI_barrier` after each iteration to synchronize across sub-communicators (see Fig. 2.2) is typically relatively small—that is, less than 1% of the total runtime.



(a) Initial (l,i) data.

(b) Zero-eliminated (l,i) data.

Figure 2.3: First step of the data compression: initial elimination of zeros in $\widetilde{\Xi}$, exemplified by a $(0\ldots58)\times(0\ldots58)$ submatrix for a sparse grid of maximum refinement level 3. In both figures, the x-direction corresponds to the dimension index *dim*, whereas the y-axis corresponds to grid points that are uniquely labeled by *nno*. The color-coded squares indicate scalar combinations of (l,i).

storing ASGs are matrix-kind of structures (see, e.g., Heinecke and Pflueger (2013)) or hash tables (see, e.g., Bungartz and Dirnstorfer (2003), and references therein). However, a direct application of those schemes is suboptimal due to particularities of the target application. To reduce the compute time spent on interpolations when performing time iteration, we, therefore, introduce here a novel data compression scheme for ASGs. Its primary features are that it significantly reduces the global memory throughput of the compute kernels and maps randomly accessed data onto cache or fast shared memory. Conceptually, an ASG is represented by a set of nno points that are all uniquely defined by multi-index pairs $(\vec{l},\vec{i})$ as well as a vector of surpluses $\vec{\alpha}$ (see (2.10) and (2.14)). Let $\widetilde{\Xi}$ be a matrix that is formed of multi-index pairs, as illustrated in Fig. 2.3a by an example for a sparse grid of maximum refinement level 3. Next, we derive a matrix Ξ from $\widetilde{\Xi}$ by pre-processing the scalar entries for every dimension dim of the multi-index pairs as $l \leftarrow 2 \ll (l-2)$ and $i \leftarrow i-1$, which leads up 96.8% of "zeros" content, as shown in Fig. 2.3b. Note that the (l,i) pair in a given dimension dim is considered zero only if both l and i are zero at the same time. Depending on the level l, each d-sized row usually contains only a few non-zero pairs: at most 1 for level 1, and at most 2 for level 2. Moreover, let n_{freq}—that is, *the number of freqencies*, be the maximum number of non-zero values across individ-

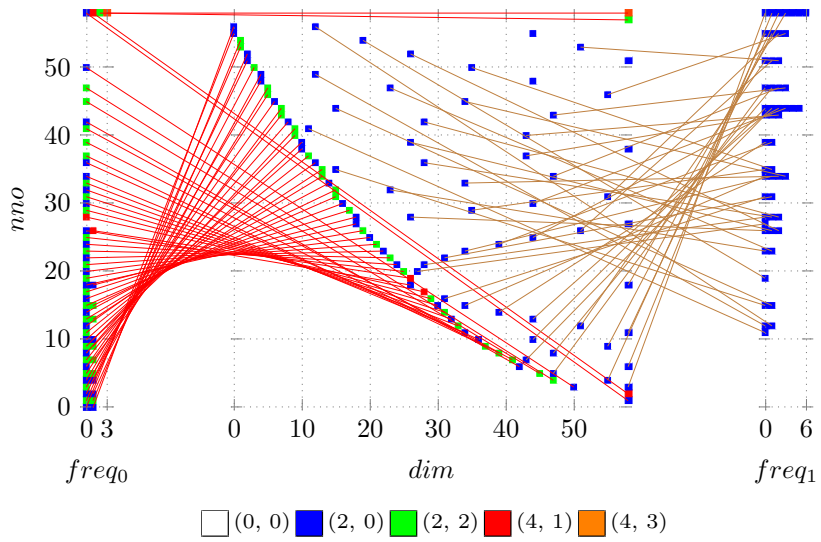


Figure 2.4: Second step of the data compression: distribution of the non-zeros from Ξ across two tables of $\{(l,i), k_{nno}\}$ elements. In addition to (l,i), the pair's Ξ row index is stored in k_{nno}.

ual Ξ rows. We decompose the dense Ξ matrix into a set of matrices $\xi^{?\times d}_{freq}$, where $freq = \overline{1, n_{freq}}$.[5] Each of those matrices shall contain no more than one non-zero element from each Ξ row such that the sequences of elements picked up as one from every ξ_{freq}, could, later on, be built up into *chains*. Therefore, ξ_{freq} rows may still contain some zero elements. The number of rows in $\xi^{?\times d}_{freq}$ is dynamically expanded to fit the actual non-zero population. The set of ξ_{freq} matrices is a sparse representation of Ξ—each ξ_{freq} element, in addition to (l,i), holds the pair's row index in Ξ. Fig. 2.4 illustrates the decomposition of Ξ into two ξ_{freq} matrices. Note that for illustrative purposes, we show here only the first 59 points from the example sparse grid being represented in our novel data structure. After the initial placement of its elements, the Ξ row index components from each ξ_{freq} matrix element are renumbered in a sorted order that ranges from the first to the last row of ξ_{freq}. A set of *transition* matrices T_{freq} holds correspondences between row indices of consecutive ξ_{freq} matrix pairs after the individual renumbering. The original Ξ row indices in the $\xi_{freq}-$elements are omitted after renumbering. The elements of the ξ_{freq} matrices are further iterated to form a global array of unique elements xps, and a linear lookup index vector V_{freq} is defined for each ξ_{freq} matrix. As result, the size of the xps array denotes which of the linear basis calculations have a non-zero contribution in forming the ASG inter-

[5] $\xi^{?\times d}_{freq}$ is a short-hand notation for the fact that we have a dynamically expandable matrix with fixed row size. We start filling it will elements, starting from the first row. If the first row's j-th column is already busy, we append the second row and place another j-th element there, and so on.

polant (see (2.14)) and thus are meaningful to perform. Finally, we use T_{freq}, xps, and the lookup indices V_{freq} to construct the set of contributing linear basis chains, as shown in Alg. 3. Note that the rows from the matrix in which we store the hierarchical

```
for i = 0, i_ch = 0; i < nno; i = i + 1, i_ch = i_ch + n_freq do
    chains(i_ch) = V(0, i);
    for i_freq = 0; i_freq < n_freq; i_freq = i_freq + 1 do
        chains(i_ch) = V(i_freq, T(i_freq, i));
    end
end
```

Algorithm 3: Construction of chains from transition matrices and lookup indices.

surpluses are reordered accordingly.

Our main motivation for the sparse grid index compression introduced above is to eliminate redundant computations when interpolating on the ASGs (see (2.14)). Indeed, as shown in the left panel of Fig. 2.5 by example of a pure x86 (serial) code listing, the complexity of the linear basis computation shrinks from $nno \times d$ iterations in the dense representation (see Brumm et al. (2015)) down to $nno \times n_{freq}$ in the case of our proposed data format. Given that in our practital application (see Sec. 2.5), $d = 59$ and n_{freq} is a small constant ($n_{freq} \leq 7$ in typical cases), the complexity decreases roughly one order of magnitude, yet introducing some memory access penalty due to the additional indexing chains. The number of meaningful basis function factors xps to be calculated is usually relatively small. For instance, $xps = 473$ in the case of a sparse grid that consists of about $nno = 300,000$ points (see Fig. 2.1), which easily fits the cache as well as the GPU shared memory (48 KB). In Sec. 2.5.1, we analyze the overall performance impact of the index compression for different interpolation kernels.

2.5 Performance and Scaling

In this section, we first show in Sec. 2.5.1 how the data structure introduced in Sec. 2.4.2 improves on the performance of the interpolation kernels. Second, we report in Sec. 2.5.2 on the single node performance achieved by the entire time iteration algorithm. Third, we evaluate the strong scaling behavior of our implementation in Sec. 2.5.3. Finally, we discuss solutions to an annually calibrated OLG model in Sec. 2.5.4.

We deploy our code on two different types of hardware. As the first testbed, we use the

```
vector<double> xpv(xps.size(), 1.0);
for (int i = 0, e = xpv.size(); i < e; i++)
{
    const Index<uint16_t>& index = xps[i];
    const uint32_t& j = index.index;
    double xp = LinearBasis(x[j], index.l, index.i);
    xpv[i] = fmax(0.0, xp);
}

for (int i = 0, ichain = 0; i < nno; i++, ichain += nfreqs)
{
    double temp = 1.0;
    for (int ifreq = 0; ifreq < nfreqs; ifreq++)
    {
        const auto& idx = chains[ichain + ifreq];
        if (!idx) break;

        temp *= xpv[idx];
        if (!temp) goto zero;
    }

    for (int dof = 0; dof < ndofs; dof++)
        value[dof] += temp * surplus(i, dof);

zero :

    continue;
}
```

```
for (int i = 0; i < nno; i++)
{
    double temp = 1.0;
    for (int j = 0; j < DIM; j++)
    {
        double xp = LinearBasis(
            x[j], index(i, j).l, index(i, j).i);
        if (xp <= 0.0) goto zero;

        temp *= xp;
    }

    for (int dof = 0; dof < ndofs; dof++)
        value[dof] += temp * surplus(i, dof);

zero :

    continue;
}
```

Figure 2.5: Comparison of the interpolation kernels for an x86 code with (left) and without (right) sparse grid index compression (cf. (2.14)).

test	d	nno	level	# states	# xps/state
“7k”	59	7,081	3	16	237
“300k”	59	281,077	4	16	473

Table 2.1: Interpolation test cases for varying sparse grid levels.

version	“7k” test [sec]	“300k” test [sec]
gold	0.000820	0.018884
x86	0.000197	0.004251
avx	0.000204	0.004221
avx2	0.000204	0.004234
avx512	0.000225	0.000907
cuda	0.000122	0.000275

Table 2.2: Performance of the interpolation kernels on various target platforms (time measured in seconds). Note that the runtime reported for the $cuda$ version accounts both for the execution time of the kernel as well as the data transfers into the final value.

Cray XC50 “Piz Daint” system. Cray XC50 compute nodes combine Intel Xeon E5-2690 v3 CPUs with one NVIDIA P100 GPU.[6] Second, we use the Cray XC40 Iron Compute “Grand Tave” system, whose nodes consist of Intel Xeon Phi 7230.[7]

2.5.1 Performance of the interpolation kernels

To demonstrate the performance gains of the interpolation kernels with respect to the novel data format (see Sec. 2.4.2), we carried out two tests on ASGs of varying size. The detailed specification for each of the test cases are summarized in Tab. 2.1. Below, we first give a short description of every version of the interpolation kernel (cf. (2.14)) and then subsequently report on the achieved performance.

gold

The *gold* version denotes a scalar interpolation kernel that operates on the data format we were using in Brumm et al. (2015) and which was based on Heinecke and Pflueger (2013).

[6] More details can be found at http://www.cscs.ch/computers/piz_daint/.
[7] For more information, see http://www.cscs.ch/computers/grand_tave/.

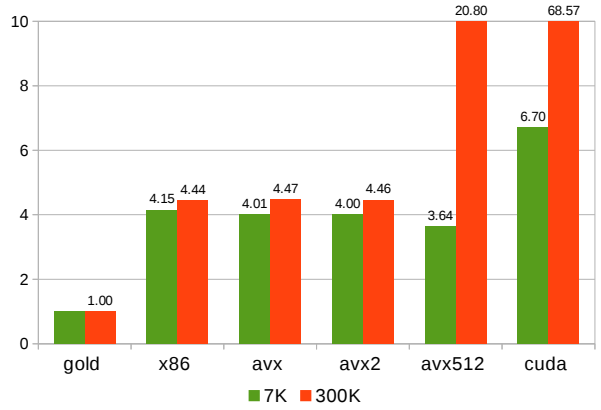


Figure 2.6: Normalized speedup gains of various interpolation kernels for the two test cases (cf. Fig. 2.1).

x86

The *x86* version leverages the novel data format in a most trivial way. The code is scalar—that is, no explicit vectorization is performed.

AVX/AVX2

In the *AVX/AVX2* kernels, the compute loops are manually vectorized. The *AVX2* additionally deploys vector FMA instructions where applicable. The effect of these optimizations is minimal due to the memory-bound nature of our problem.

AVX512

Unlike its *AVX/AVX2* siblings, the *AVX512* version has to deal with much less cache size per compute core. Therefore, it deploys OpenMP parallelization inside the interpolation kernel instead of high-level TBB work distribution (cf. Fig. 2.2). As long as the kernel performs the summing of the nno vectors, *AVX512* deploys an OpenMP 4 user-defined reduction with partial vector sums that are implemented in 512-bit wide intrinsics. By the nature of the algorithm, many partial vector sums end up making zero contributions. They are handled specially to initiate no actual memory flow and to reduce the cache pollution, yet causing imbalances in the reduction tree traversal.

CUDA

The *CUDA* version offloads the interpolation kernel to the NVIDIA Tesla P100 GPU. The scheduler uses a block size of 128, which is the closest to the $ndofs$ per point.[8] The nno is distributed across the maximum number of concurrent blocks for a given SM and register count. In this way, the whole kernel workload efficiently goes through in a single "wave" of blocks. The xpv array is mapped onto the shared memory. Unlike the "300k" test case, the "7k" benchmark is not sufficiently large to fully utilize the P100 compute resources and therefore demonstrates only a moderate speedup.

As an indicative performance measure, we consider the average execution time of a particular kernel. The data was generated by evaluating the interpolation kernels at 1,000 randomly sampled grid points in **B** and then taking the average runtime. The performance results for the various implementations are reported in Tab. 2.2 and Fig. 2.6. The latter is normalized with respect to the *gold* version. Note that all kernels—except *AVX512* and *CUDA*—are single-threaded and therefore delegate the thread parallelism to the upper-level TBB scheduler (see Fig. 2.2). As shown in Fig. 2.6, we find that deploying the novel data structure delivers a speedup of about 4×, whereas in combination with $AVX512$ and $CUDA$ (where there are also more compute resources available), we can reach a combined speedup of almost two orders of magnitude. For further details, please refer to the interpolation kernel source code (Scheidegger and Mikushin, 2018).

2.5.2 Single-node performance: KNL versus CPU/GPU clusters

To give a measure of how the single-node parallelization scheme discussed in Sec. 2.4.1 impacts the performance, we evaluate the first two sparse grid levels of a single time step from the OLG model as outlined in Sec. 2.2. This relatively small instance consists of $16 \cdot 119 = 1{,}904$ grid points, $16 \cdot 119 \cdot 59 = 112{,}336$ variables and constraints. The results are summarized in Fig. 2.7. They indicate a total speedup of 25× when going from a single CPU thread implementation to a more efficient version of utilizing both all CPU and GPU resources present on a "Piz Daint" compute node. In case of running the same experiment on "Grand Tave", we find that utilizing Xeon Phi KNL in a multi-threaded mode delivers a speedup of about 96× over a single-threaded version. Moreover, we observe that for our target application, "Piz Daint" nodes are about 2× faster than the ones from "Grand Tave".

[8] Note that the variable $ndofs = 2 \cdot d = 118$ corresponds to the 118 coefficients $\alpha = (\alpha^k, \alpha^v)$ that are used to approximate the policy and value functions (see Sec. 2.2.1).

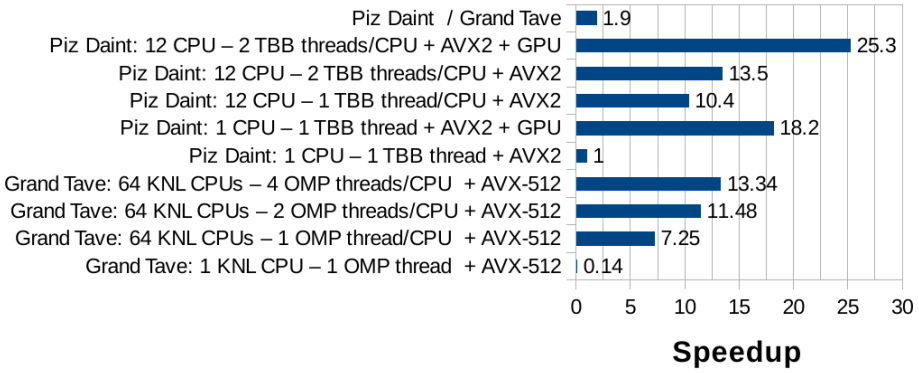


Figure 2.7: Comparison of wall times for different stochastic OLG code variants on a single node of “Piz Daint" and “Grand Tave". The speedup is normalized with respect to an optimized, single-threaded test instance on “Piz Daint", whose runtime corresponds to 2,243 seconds.

2.5.3 Strong Scaling

We now report the strong scaling efficiency of our code. The test problem is again a single time step of a 59-dimensional OLG model with 16 discrete, stochastic states. To provide a consistent benchmark, we used a nonadaptive sparse grid of refinement level 4 that was restarted from a sparse grid of level 2. This test case consists of $16 \cdot 281,077 = 4,497,232$ points and $16 \cdot 281,077 \cdot 59 = 265,336,688$ unknowns and constraints per time step. The economic test case was solved with increasingly larger numbers of nodes (from 1 to 4,096 nodes on “Piz Daint"). Fig. 2.8 shows the normalized execution time and scaling on different levels and their ideal speedups. We used 1 MPI process per multi-threaded node. In case of running the benchmark on “Piz Daint", the code scales nicely up to 4,096 nodes, where the overall efficiency still is around 70%.[9] Thus, combined with the single-node speedup gains reported in Sec. 2.5.2 we attain an overall speedup of more than four orders of magnitude for our benchmark. There is one dominant limitation to the strong scaling. It stems from the fact that within the lower refinement levels, the ratio of “points to be evaluated per thread” is often smaller than one with increasing node numbers, i.e., threads are idling. The better parallel efficiency on the higher refinement levels is due to the fact we have in this situation many more points available, resulting in a workload that is somewhat fairer distributed among the different MPI processes and their respective threads (see Fig. 2.8).

[9] Note that due to the limited size of “Grand Tave"—less than 200 nodes—we did not add the corresponding strong scaling figures here. From Fig. 2.8, it becomes evident that our code scales almost perfectly on such a small system.

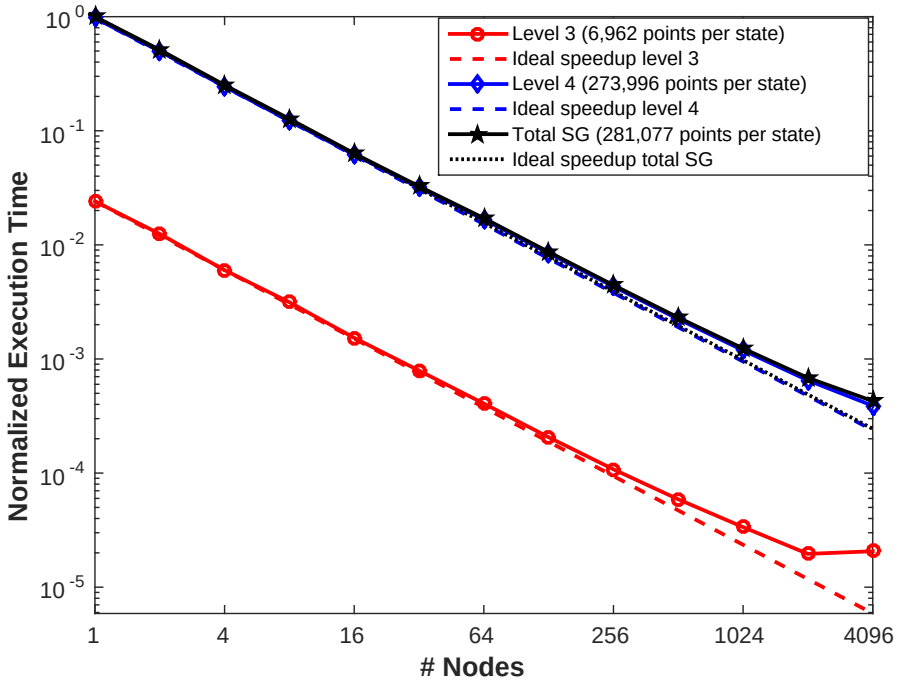


Figure 2.8: Strong scaling on “Piz Daint" for an OLG model using 4 levels of grid refinements, 16 discrete states, and $16 \cdot 281,077 = 4,497,232$ points and $265,336,688$ unknowns in total. “Total SG” shows the entire, normalized simulation time up to $4,096$ nodes, where the runtime for a single node corresponds to $20,471$ seconds. We also show normalized execution times for the computational sub-components on different levels, e.g., for level 3 using 6.962 points. Dashed lines show the ideal speedup.

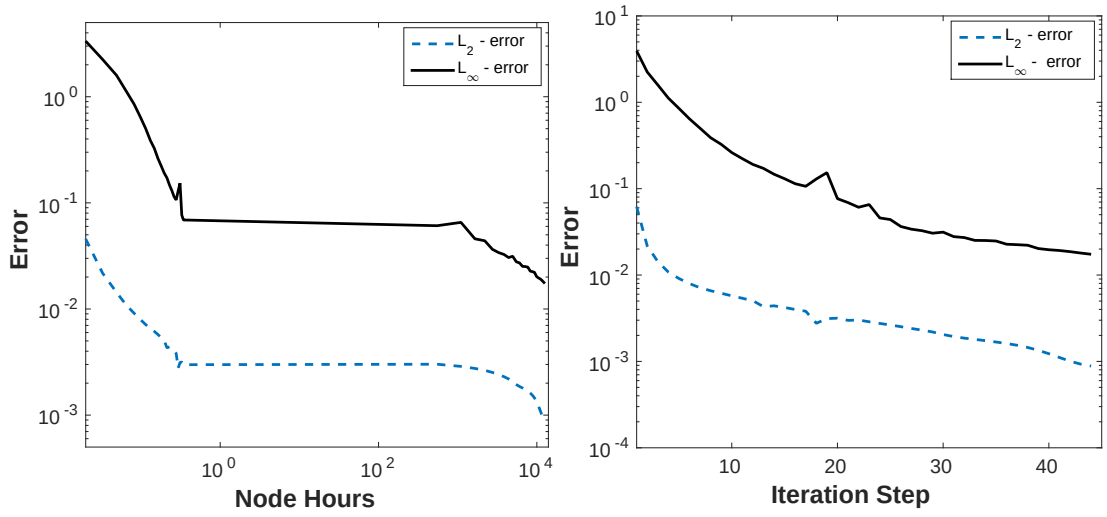


Figure 2.9: Comparison of the L_2 and $L_\infty-$error for adaptive sparse grid solutions of the 59-dimensional OLG model as a function of compute time or iterations spent on “Piz Daint".

2.5.4 Convergence of the time iteration algorithm

For the purpose of testing the convergence of our massively parallel time iteration algorithm, we compute equilibria for the model outlined in Sec. 2.2 and a decreasing refinement threshold ϵ (see Sec. 2.3). In the left panel of Fig. 2.9, we compare the decaying L_2- and L_∞- error for a complete simulation of a 59-dimensional model as a function of compute time. The right panel of Fig. 2.9 shows the decreasing errors as a function of iteration steps. It is apparent from Fig. 2.9 that convergence of the time iteration algorithm is rather slow. This is to be expected, as time iteration has, at best, a linear convergence rate in iterations (Maldonado and Svaiter, 2007). The time iteration was terminated once the average error dropped below the satisfactory level of 0.1 percent. For this iteration step, the ASGs consist in average of 73,874 points per state, however varying between a minimum of 69,026 points in state $z = 6$ and a maximum of 76,645 points in state $z = 1$.[10]

2.6 Conclusions

Solving mixed high-dimensional continuous/discrete dynamic stochastic economic models in competitive times, that is, in hours or days of human time at maximum imposes many challenges both from a modeling as well as from a computational perspective. We demonstrate in this paper that by combining ASGs (that ameliorate the curse of dimensionality imposed by the significant heterogeneity of the economic model) with efficient data structures, a time iteration algorithm (that deals with the recursive nature of the problem formulation), and with hybrid HPC compute paradigms (which drastically reduces the time-to-solution process), we can handle the difficulties imparted by this particular model class up to a level of complexity not seen before. By exploiting the generic structure of the economic model under consideration, we implemented a hybrid parallelization scheme that uses state-of-the-art parallel computing paradigms. It minimizes MPI interprocess communication by using TBB and AVX, AVX2 or AVX-512 vectorization (depending on the hardware available), and partially offloads the function evaluations to GPUs if available. In addition, we introduced a novel data compression scheme for ASGs that resulted in accelerating the compute time spent on interpolations by about 4×. Numerical tests on “Piz Daint" (a hybrid CPU/GPU system) and “Grand Tave" (a Xeon Phi KNL cluster)

[10]Note that we carried out our computations by setting $L_{max} = 6$ and fixing ϵ until the error level did not improve any further. We subsequently restarted the code with a decreased value of ϵ. This measure then slightly adds points to the grid and therefore further lowers the error. As the size of the classical sparse grid grows very fast in high dimensions when the level increases—from 119 ($l = 2$), 7,081 ($l = 3$), 281,077 ($l = 4$), 8,378,001 ($l = 5$), to $> 2 \cdot 10^8$ ($l = 6$)— adaptive sparse grids allow us to look at intermediate numbers of grid points.

at CSCS show that our code is highly scalable. In the case of a stochastic public finance OLG model with 60 generations and sixteen discrete states, we found excellent strong scaling properties up to 4,096 nodes, resulting in an overall speedup of more than four orders of magnitude compared to a single, optimized CPU thread.

Acknowledgements

All authors gratefully acknowledge financial support from PASC. This work was supported by grants from the Swiss National Supercomputing Centre under project IDs s555 and s790. Moreover, we gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.

Part III

3 GPU-Accelerated Dynamic Human Capital Models

3.1 Introduction

Economists use dynamic structural econometric models to study individual decision-making such as career decisions about schooling, work, occupational choice, the impact of economic mechanisms, and to forecast the effect of public policies on the welfare (see, e.g., Blundell et al. (2016)). These models specify the individuals' objective, their economic environment, and the institutional and informational constraints under which they operate. This paper is dedicated to Eckstein–Keane–Wolpin (EKW) models used to quantitatively study human capital investment decisions over the full life cycle of an individual (Aguirregabiria and Mira, 2010). Human capital comprises the knowledge, skills, competencies, and attributes embodied in individuals facilitating the creation of personal, social, and economic well-being (Becker, 1964). Differences in human capital attainment lead to inequality in various life outcomes such as labor market success and health across and within countries (OECD, 2001).

EKW models are formulated as discrete-state dynamic stochastic optimal control problems (Judd, 1998), and are usually solved by backward induction method (Bellman, 1961). This method suffers from the curse of dimensionality, which could be alleviated only under very strong restrictions on the fundamentals of the model (Rust, 1997). Solving a large-scale EKW model numerically by a backward induction scheme at each iteration step:

- requires a highly accurate approximation of economic functions at a vast number of discrete points;
- yields to a maximization problem that needs to be solved by brute force via grid search, and involves the numerical approximation of a complex integral.

Hence, achieving a fast time-to-solution is difficult for even a single solution of a realistically-sized EKW model. Moreover, to actually calibrate the model to empirical data, it needs to be solved hundreds of thousands times. According to Borella et al. (2019), a model parallelized in C, requires 22 minutes for each set of parameter values to be solved on high-end workstations, totalling at least three or four weeks runtime for a single cohort. As a consequence, economists restrict themselves to working with simplified models, and a small number of calibrations. Even the simplified models are only solved to a low level of numerical accuracy and no uncertainty quantification or sensitivity analysis is done. This situation is particularly unsatisfactory as the amount of populational data is ever-increasing and allows to add important economic mechanisms that are so far ignored to ease the computational burden.

This work is based on the RESPY project (Gabler and Raabe, 2020), a research code for the flexible specification, simulation, and estimation of EKW models. Being developed in pure Python, RESPY is an example of readable and descriptive scientific software, which expresses each component of EKW model with simple and natural Python language constructs. As almost any pure Python software, RESPY's scalability is limited by nature of Python runtime environment. Therefore, we have set out our goal to create an "enhanced" variant of RESPY, which rewrites the original software very closely, but in a more HPC-friendly manner. Specifically, our contribution is threefold:

1. an optimal memory layout for EKW model data, so that the CPU could perform backward induction with minimum possible pipeline stalls,
2. a partitioning algorithm to distribute the EKW problem to hundred thousand of parallel workers,
3. a generic C++ model core, which is compatible with CUDA and HIP GPU runtimes used by 5 out of 6 currently fastest supercomputers.

The computational kernel of the RESPY EKW solver consists of dense linear algebra and reduction operations implemented mainly in Python `Pandas.DataFrame` (Wes McKinney, 2010). The main purpose of `Pandas.DataFrame` in EKW solver is to store the economic agents data, and apply `groupby` filters to construct various aggregates and subviews. Being particularly good for prototyping, the Pandas framework quickly hits its limited scalability (Petersohn et al., 2020). We tackle the scalability limitation in three directions. First, we develop forward and inverse translation formulas between each possible discrete choice and the index of its corresponding data value in a flat 1-dimensional array. Second, we replace Python-based data structures with native C++ arrays. Finally, we implement each period of backward induction process in

such way, that iterations over all possible discrete choices are performed on-the-fly, without precomputing and storing the list of them. Ultimately, our optimizations demonstrated two orders of magnitude speedup over the original Python version, and allow to simulate much larger EKW problems.

While in this article we focus on EKW models, the presented performance tuning approach could be useful for any discrete-state optimal control problems.

We organize the remainder of this article as follows. In Section 3.2, we briefly introduce the class of EKW models. We discuss the economic framework, their mathematical formulation, and outline the calibration procedures that allow to fit them to observational data. We then, in Section 3.3, present an empirical application of the model. In Section 3.4 we focus on the implementation details, including the general discrete-state dynamic programming algorithm, its novel parallelization scheme. Finally, in Section 3.9, we substantiate our performance claims for the algorithm in the empirical application. Section 3.10 concludes.

3.2 Eckstein-Keane-Wolpin models

To demonstrate the capabilities of the massively-parallelized solution algorithm for EKW models introduced in this article, we now present their formal setup. We first describe the economic framework in Section 3.2.1, then turn in Section 3.2.2 to their mathematical formulation, and finally outline in Section 3.2.3 the calibration procedure.

3.2.1 Economic framework

EKW models describe the sequential decision-making of economic agents under uncertainty (Gilboa, 2009; Machina and Viscusi, 2014). At time $t = 1, \ldots, T$, where $t \in \mathbb{N}$, each individual observes the state of the economic environment $s_t \in S$, and chooses an action a_t from the set of admissible actions $\mathscr{A}$. In previewing our application, we model the human capital investment decisions of individuals between age 16 and 65. The state s_t contains, for example, information about an individual's labor market experience and educational attainment, and individuals can choose to either join the labor market, go to school, or simply stay at home. The decision has two consequences: an individual receives an immediate reward $u_t(s_t, a_t)$, and the economy evolves to a new state s_{t+1}. Going forward, we will use the term "utility" instead of a reward. Economists use the term utility to refer to a level of overall satisfaction. In the context of our application, the utility from working is partly determined by the wage an

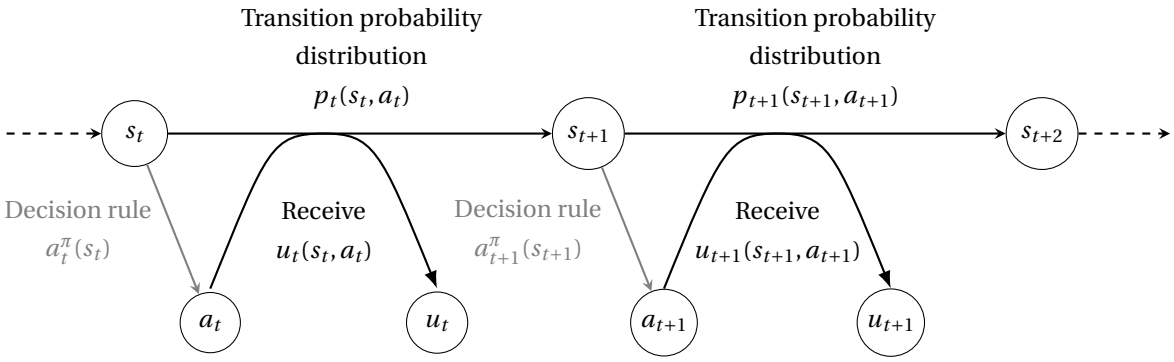


Figure 3.1: Timing of events in the discrete-state dynamic EKW model for two generic time periods.

individual receives but also other job amenities. The transition from s_t to s_{t+1} is affected by the action, but remains uncertain. Individuals are assumed to be forward-looking. Thus, they do not simply choose the alternative with the highest immediate utility. Instead, they take the future consequences of their current action into account.

A policy $\pi = (a_1^\pi(s_1), \ldots, a_T^\pi(s_T))$ provides the individual with instructions for choosing an action in any possible future state. It is a sequence of decision rules $a_t^\pi(s_t)$ that specify the action at a particular time t for any possible state s_t under π. As individuals follow their policy, they receive a sequence of utilities that depends on the objective transition probability distribution $p_t(s_t, a_t)$ for the evolution of state s_t to s_{t+1} induced by the model. Individuals have rational expectations (Muth, 1961)—that is, their subjective beliefs about the future agree with the objective transition probabilities of the model.

Figure 3.1 depicts the timing of events in the model for two generic periods. At the beginning of period t, an individual fully learns about the immediate utility of each alternative, chooses one of them, and receives its immediate utility. Then the state evolves from s_t to s_{t+1}, and the process is repeated in $t+1$. Individuals face uncertainty, and they seek to maximize the expected total discounted utilities. An exponential discount factor $0 < \delta < 1$ parameterizes their time preference and captures a taste for immediate over future utilities.

The subsequent equation (3.1) provides the formal representation of the individual's objective: Given an initial state s_1, individuals implement the policy π from the set of all possible policies Π that maximizes the expected total discounted utilities over all T

decision periods given the information $\mathcal{I}_1$ available in the first period:

$$\max_{\pi \in \Pi} \mathrm{E}^{\pi}_{s_1} \left[\sum_{t=1}^{T} \delta^{t-1} u_t(s_t, a^{\pi}_t(s_t)) \,\middle|\, \mathcal{I}_1 \right]. \tag{3.1}$$

The superscript of the expectation emphasizes that each policy π induces a different probability distribution over the sequences of utilities.

3.2.2 Mathematical formulation and a solution algorithm

EKW models are set up as a standard Markov decision process (MDP) (Puterman, 1994; White, 1993). When making sequential decisions under uncertainty, the decision-maker seeks to implement the optimal policy π^* with the largest expected total discounted utilities $v^{\pi^*}_1(s_1)$ as formalized in equation (3.1). In principle, this requires evaluating all policies' performance based on all possible sequences of utilities, each weighted by the probability with which they occur. Fortunately, however, the multi-stage problem can be solved by a sequence of simpler inductively defined single-stage problems.[1]

The value function $v^{\pi}_t(s_t)$ captures the expected total discounted utilities under policy π from period t onwards for an individual experiencing state s_t:

$$v^{\pi}_t(s_t) = \mathrm{E}^{\pi}_{s_t} \left[\sum_{j=0}^{T-t} \delta^j \, u_{t+j}(s_{t+j}, a^{\pi}_{t+j}(s_{t+j})) \,\middle|\, \mathcal{I}_t \right].$$

Then we can determine $v^{\pi}_1(s_1)$ for any policy by recursively evaluating equation (3.2):

$$v^{\pi}_t(s_t) = u_t(s_t, a^{\pi}_t(s_t)) + \delta \, \mathrm{E}^{\pi}_{s_t} \left[\, v^{\pi}_{t+1}(s_{t+1}) \,\middle|\, \mathcal{I}_t \right]. \tag{3.2}$$

Equation (3.2) expresses the total value $v^{\pi}_t(s_t)$ of adopting policy π going forward as the sum of its immediate utility and all expected discounted future utilities.

The principle of optimality (Bellman, 1954) allows to construct π^* by solving the optimality equations (3.3) for all s and t recursively:

$$v^{\pi^*}_t(s_t) = \max_{a_t \in A} \left\{ u_t(s_t, a_t) + \delta \, \mathrm{E}^{\pi^*}_{s_t} \left[\, v^{\pi^*}_{t+1}(s_{t+1}) \,\middle|\, \mathcal{I}_t \right] \right\}. \tag{3.3}$$

[1] Optimal decisions in an MDP are a deterministic function of the current state s only—that is, an optimal decision rule is always deterministic and Markovian. We restrict our notation to this special case right from the beginning.

The optimal value function $v_t^{\pi^*}$ is the sum of the expected discounted utilities in t over the remaining time horizon assuming the optimal policy is implemented going forward. The optimal action is choosing the alternative with the highest total value:

$$a_t^{\pi^*}(s_t) = \underset{a_t \in A}{\arg\max} \left\{ u_t(s_t, a_t) + \delta \, \mathrm{E}_{s_t}^{\pi^*} \left[v_{t+1}^{\pi^*}(s_{t+1}) \middle| \mathscr{I}_t \right] \right\}.$$

Algorithm 4 illustrates the canonical way to solve the MDP by a backward induction procedure. In the final period, T, there is no future to take into account, and the optimal action is choosing the alternative with the highest immediate utilities in each state. With the decision rule for the final period at hand, the other optimal decisions can be determined recursively following equation (3.3) as the calculation of their expected future utilities is straightforward given the relevant transition probabilities.

2

From a computational perspective, it is important to note that the number of states and actions can be large, and can leverage high-performance parallel and distributed computing methodology. Section 3.4 describes how the original backward induction algorithm can be partitioned for execution on multicore CPUs and GPUs.

3.2.3 Calibration procedure

EKW models are calibrated to data on observed individual decisions and experiences under the hypothesis that the individual behaves according to the model. This requires the full parameterization θ of the model as we need to specify the functional form of the immediate utility functions and impose distributional assumptions about the

[2]Note that algorithm 4 is closely related to reinforcement learning. The latter is branch of machine learning that, among other things, is concerned with solving dynamic programming problems. Reinforcement learning is a data-driven approach to solving dynamic programming problems, where the data can be created artificially via simulations or by real-life observations (see Sutton and Barto (2018) for a thorough introduction). This type of machine learning refers to training a model to maximize a reward via iterative trial and error. Both optimal control and reinforcement learning aim to find a policy that optimizes a long-term performance measure. In contrast to optimal control, reinforcement learning usually does not assume a priori known transition dynamics and cost. Hence, general reinforcement learning algorithms have to treat these quantities as random variables. However, if reinforcement learning algorithms are applied to a fully known Markov decision process such as the ones we are working with in this chapter, the reinforcement learning problem can be considered equivalent to optimal control. The dynamic programming recursion and, therefore, all related algorithms, such as value function iteration, can be used to solve this problem. Both reinforcement learning and optimal control aim to find a solution to an optimization problem where the effect of the current decision can be delayed. For further details on optimal control, dynamic programming, and reinforcement learning, we refer to Sutton and Barto (2018).

for $t = T, \ldots, 1$ **do**
  if $t = T$ **then**
    $v_T^{\pi^*}(s_T) = \max_{a_T \in A} \left\{ u_T(s_T, a_T) \right\} \qquad \forall\, s_T \in S$
  else
    Compute $v_t^{\pi^*}(s_t)$ for each $s_t \in S$ by
    $v_t^{\pi^*}(s_t) = \max_{a_t \in A} \left\{ u_t(s_t, a_t) + \delta\, \mathrm{E}_{s_t}^{\pi} \left[v_{t+1}^{\pi^*}(s_{t+1}) \middle| \mathcal{I}_t \right] \right\}$
    and set
    $a_t^{\pi^*}(s_t) = \arg\max_{a_t \in A} \left\{ u_t(s_t, a_t) + \delta\, \mathrm{E}_{s_t}^{\pi} \left[v_{t+1}^{\pi^*}(s_{t+1}) \middle| \mathcal{I}_t \right] \right\}.$
  end if
end for

Algorithm 4: Backward induction procedure.

unobservables of the model. The goal of the calibration is to learn about the utility functions and preference parameters that govern individual decision-making.

Economists have access to information for $i = 1, \ldots, N$ individuals in each time period $t = 1, \ldots, T_i$. For every observation (i, t) in the data, we observe the action a_{it}, some components $\bar{u}_{it}$ of the utility, and a subset $\bar{s}_{it}$ of the state s_{it}. Therefore, from the researcher's point of view, we need to distinguish between two types of state variables—that is, $s_{it} = (\bar{s}_{it}, \epsilon_{it})$. At time t, the economist and individual both observe $\bar{s}_{it}$ while ϵ_{it} is only observed by the individual. In summary, the data $\mathscr{D}$ has the following structure:

$$\mathscr{D} = \{a_{it}, \bar{s}_{it}, \bar{u}_{it} : i = 1, \ldots, N; t = 1, \ldots, T_i\},$$

where T_i is the number of observations for which we observe individual i.

Numerous calibration procedures for different settings exist (Davidson and MacKinnon, 2003; Gourieroux and Monfort, 1996). We briefly outline the likelihood-based and simulation-based calibration. Independent of the calibration criterion, it is necessary to solve for the optimal policy π^* at each candidate parameterization of the model.

Likelihood-based calibration seeks to find the parameterization $\hat{\theta}$ that maximizes the likelihood function $\mathscr{L}(\theta \mid \mathscr{D})$, i.e., the probability of observing the given data as a function of θ. As we only observe a subset $\bar{s}_t$ of the state, we can determine the probability $p_{it}(a_{it}, \bar{u}_{it} \mid \bar{s}_{it}, \theta)$ of individual i at time t in $\bar{s}_{it}$ choosing a_{it} and receiving u_{it} given parametric assumptions about the distribution of ϵ_{it}. The objective function

takes the following form:

$$\hat{\theta} \equiv \underset{\theta \in \Theta}{\arg\max} \underbrace{\prod_{i=1}^{N} \prod_{t=1}^{T_i} p_{it}(a_{it}, \bar{u}_{it} \mid \bar{s}_{it}, \theta)}_{\mathscr{L}(\theta|\mathscr{D})}.$$

In simulation-based calibration, our goal is to find the parameterization $\hat{\theta}$ that yields a simulated data set from the model that closest resembles the observed data. More precisely, the goal is often to minimize the weighted squared distance between a set of moments M_D computed on the observed data and the same set of moments computed on the simulated data $M_S(\theta)$.[3] The objective function takes the following form:

$$\hat{\theta} \equiv \underset{\theta \in \Theta}{\arg\min} (M_D - M_S(\theta))' W (M_D - M_S(\theta)).$$

3.3 A fully specified EKW model

We now present an exemplifying analysis of a canonical EKW model on human capital investment that is confronted with real data. The model was initially studied in Keane and Wolpin (1997) to explore the career decisions of young men about their schooling, work, and occupational choice, and will serve us below as an ideal benchmark to measure performance since it's solutions are known. We first outline the model's basic setup and provide some descriptive statistics of the empirical data used for its calibration.

3.3.1 Basic setup

We follow individuals over their working life from young adulthood at age 16 to retirement at age 65 where the decision period $t = 16, \ldots, 65$ is one (discrete-valued) school year. Figure 3.2 illustrates the initial decision problem as individuals decide $a \in \mathscr{A}$ whether to work in a blue-collar or white-collar occupation ($a = 1, 2$), to serve in the military ($a = 3$), to attend school ($a = 4$), or to stay at home ($a = 5$). Individuals are already heterogeneous when entering the model. They differ with respect to their level of completed schooling h_{16} and have one of four different $\mathscr{J} = \{1, \ldots, 4\}$ alternative-specific skill endowments $\boldsymbol{e} = \left(e_{j,a}\right)_{\mathscr{J} \times \mathscr{A}}$.

[3] Note that the full solution of the model is required to carry out this maximum likelihood computation, as its moments need to be simulated.

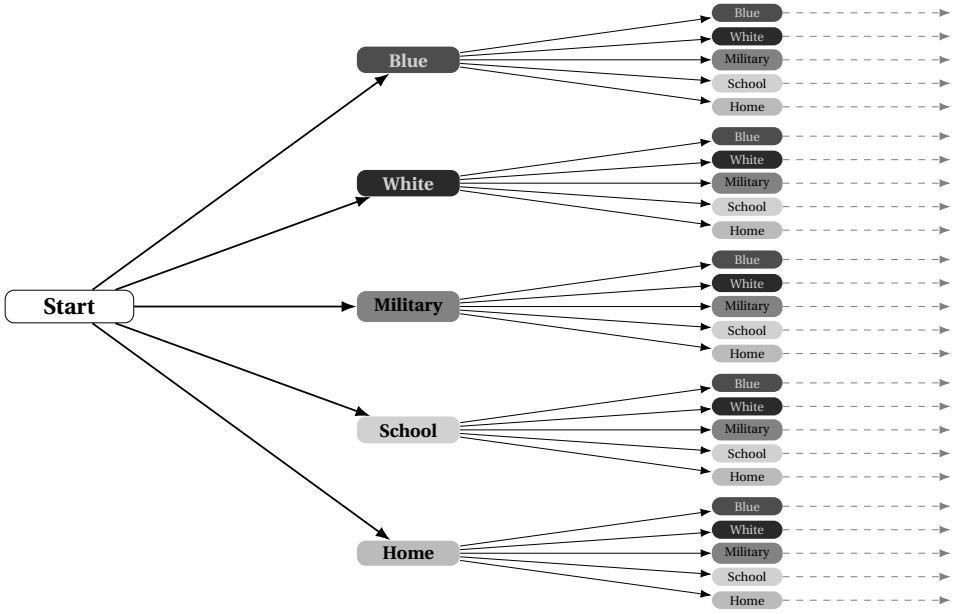


Figure 3.2: Decision tree for the initial two periods of the model that reveals the curse of dimensionality that plagues the applications of EKW models. In the final period of our examples ($T = 49$), there are 5^{49} different states.

The immediate utility $u(\cdot)$ of each alternative consists of a non-pecuniary utility $\zeta_a(\cdot)$ and, at least for the working alternatives, an additional wage component $w_a(\cdot)$. Both depend on the level of human capital as measured by their occupation-specific work experience $\boldsymbol{k}_t = \left(k_{a,t}\right)_{a\in\{1,2,3\}}$, years of completed schooling h_t, and alternative-specific skill endowment $\boldsymbol{e}$. The immediate utilities are influenced by last-period choices a_{t-1} and alternative-specific productivity shocks $\boldsymbol{\epsilon_t} = \left(\epsilon_{a,t}\right)_{a\in\mathcal{A}}$ as well. Their general form is given by:

$$u(\cdot) = \begin{cases} \zeta_a(\boldsymbol{k}_t, h_t, t, a_{t-1}) & \\ \quad + w_a(\boldsymbol{k}_t, h_t, t, a_{t-1}, e_{j,a}, \epsilon_{a,t}) & \text{if} \quad a \in \{1,2,3\}\,. \\ \zeta_a(\boldsymbol{k}_t, h_t, t, a_{t-1}, e_{j,a}, \epsilon_{a,t}) & \text{if} \quad a \in \{4,5\} \end{cases}$$

Work experience $\boldsymbol{k}_t$ and years of completed schooling h_t evolve deterministically as

$$\begin{aligned} k_{a,t+1} &= k_{a,t} + \mathbf{I}[a_t = a] \quad \text{if} \quad a \in \{1,2,3\} \\ h_{t+1} \quad &= h_t \quad + \mathbf{I}[a_t = 4]. \end{aligned}$$

The productivity shocks $\boldsymbol{\epsilon}_t$ are uncorrelated across time and follow a multivariate normal distribution with mean $\mathbf{0}$ and covariance matrix $\boldsymbol{\Sigma}$. Given the structure of the utility functions and the distribution of the shocks, the state at time t is $s_t =$

$\{\boldsymbol{k}_t, h_t, t, a_{t-1}, \boldsymbol{e}, \boldsymbol{\epsilon}_t\}$.

Theoretical and empirical research from specialized disciplines within economics informs the specification of each $u_a(\cdot)$ and we discuss the exact functional form of the per-period utility in the blue-collar occupation as an example.[4] Equation (3.4) shows the parameterization of the non-pecuniary utility from working in a blue-collar occupation:

$$\begin{aligned}\zeta_1(\boldsymbol{k}_t, h_t, a_{t-1}) = \alpha_1 &+ c_{1,1} \cdot \mathbf{I}[a_{t-1} \neq 1] + c_{1,2} \cdot \mathbf{I}[k_{1,t} = 0] \\ &+ \vartheta_1 \cdot \mathbf{I}[h_t \geq 12] + \vartheta_2 \cdot \mathbf{I}[h_t \geq 16] \\ &+ \vartheta_3 \cdot \mathbf{I}[k_{3,t} = 1].\end{aligned} \tag{3.4}$$

It includes job amenities α_1 and mobility and search costs $(c_{1,1}, c_{1,2})$ that capture the extra effort for individuals who only recently started working in a blue-collar occupation. Additional components depend on whether an individual has a high school ϑ_1 or college ϑ_2 degree. There is a detrimental impact of leaving the military after a single year ϑ_3.

The wage component $w_1(\cdot)$ is given by the product of the market-equilibrium rental price r_1 and an occupation-specific skill level $x_1(\cdot)$. The overall level of human capital determines the latter. This specification leads to a standard logarithmic wage equation in which the constant term is the skill rental price $\ln(r_1)$ and wages follow a log-normal distribution.

The occupation-specific skill level $x_1(\cdot)$ is determined by a skill production function, which includes a deterministic component $\Gamma_1(\cdot)$ and a multiplicative stochastic productivity shock $\epsilon_{1,t}$:

$$x_1(\boldsymbol{k}_t, h_t, t, a_{t-1}, e_{j,1}, \epsilon_{1,t}) = \exp\left(\Gamma_1(\boldsymbol{k}_t, h_t, t, a_{t-1}, e_{j,1}) \cdot \epsilon_{1,t}\right).$$

Equation (3.5) shows the parameterization of the deterministic component of the skill

[4] All additional details are available at https://bit.ly/ekw-handout.

production function:

$$\begin{aligned}\Gamma_1(\boldsymbol{k}_t, h_t, t, a_{t-1}, e_{j,1}) = e_{j,1} &+ \beta_{1,1} \cdot h_t + \beta_{1,2} \cdot \mathbf{I}[h_t \geq 12] \\ &+ \gamma_{1,1} \cdot k_{1,t} + \gamma_{1,2} \cdot (k_{1,t})^2 \\ &+ \gamma_{1,3} \cdot \mathbf{I}[k_{1,t} > 0] + \gamma_{1,4} \cdot t \\ &+ \gamma_{1,5} \cdot \mathbf{I}[t < 18] \\ &+ \gamma_{1,6} \cdot \mathbf{I}[a_{t-1} = 1] \\ &+ \gamma_{1,7} \cdot k_{2,t} + \gamma_{1,8} \cdot k_{3,t}.\end{aligned} \tag{3.5}$$

There are several notable features. Skills increase with schooling $\beta_{1,1}$ and blue-collar work experience $(\gamma_{1,1}, \gamma_{1,2})$. There are so-called sheep-skin effects associated with completing a high school $\beta_{1,2}$ and graduate $\beta_{1,3}$ education that capture the impact of completing a degree beyond just the associated years of schooling. Also, there is a first-year blue-collar experience effect $\gamma_{1,3}$ while skills depreciate when not employed in a blue-collar occupation in the preceding period $\gamma_{1,6}$. Other work experience $(\gamma_{1,7}, \gamma_{1,8})$ is transferable.

The goal of the following calibration step is to learn about the quantitative values of the parameters characterizing the immediate utility from each alternative.

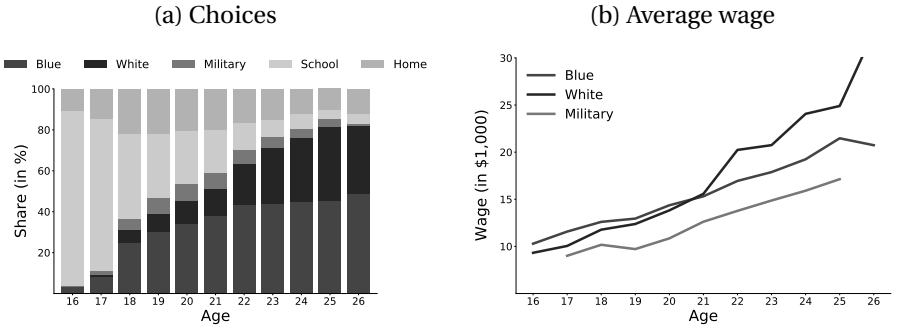
$$\theta = \left\{ \{c_{1,i}\}_{i=1,2}, \{\nu_{1,i}\}_{i=1,2,3}, e_{j,1} \{\beta_{1,i}\}_{i=1,2}, \{\gamma_{1,i}\}_{i=1,\ldots,6}, \ldots \right\}$$

3.3.2 Empirical data

We analyze the original dataset used by Keane and Wolpin (1997) and thus only provide a brief description here.[5] The authors construct their sample based on the National Longitudinal Survey of Youth 1979 (NLSY79) (Bureau of Labor Statistics, 2019). The NLSY79 is a nationally representative sample of young men and women living in the United States in 1979 and born between 1957 and 1964. Individuals were followed from 1979 onwards and repeatedly interviewed about their schooling decisions and labor market experiences. Based on this information, individuals are assigned to either working in one of the three occupations, attending school, or simply staying at home.

[5] We use the same data as in Keane and Wolpin (1997). Please see https://bit.ly/ekw-data for details.

Figure 3.3: Data overview



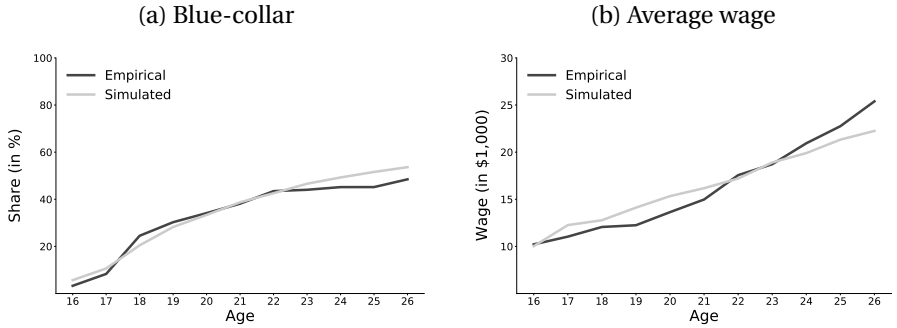
Notes: The wage is a full-time equivalent deflated by the gross national product deflator, with 1987 as the base year. We do not report the wage if less than ten observations are available.

Keane and Wolpin (1997) restrict attention to white males that turn 16 between 1977 and 1981 and exploit the information collected between 1979 and 1987. Thus individuals in the sample are all between 16 and 26 years old. While the sample initially consists of 1,373 individuals at age 16, this number drops to 256 at the age of 26 due to sample attrition, missing data, and the short observation period. Overall, the final sample consists of 12,359 person-period observations.

Figure 3.3 summarizes our information about choices and wages by age. We show the distribution of choices on the left and report average wages on the right. Initially, roughly 86% of individuals enroll in school, but this share steadily declines with age. Nevertheless, about 39% obtain more than a high school degree and continue their schooling for more than twelve years. As individuals leave school, most of them initially pursue a blue-collar occupation. However, the relative share of the white-collar occupation increases as individuals entering the labor market later have higher levels of schooling. At age 26, about 48% work in a blue-collar occupation and 34% in a white-collar occupation. The share of individuals in the military peaks around age 20 when it amounts to 8%. At its maximum around age 18, approximately 20% of individuals stay at home.

Overall, average wages start at about $10,000 at age 16 but increase considerably up to about $25,000 at age 26. While wages in the blue-collar occupation are initially highest with about $10,286, wages in the white-collar occupation and military start around $9,000. However, wages in the white-collar occupation increase steeper over time and overtake blue-collar wages around age 21. At the end of the observation period, wages in the white-collar occupation are about 50% higher than blue-collar wages with $32,756 as opposed to only $20,739. Military wages remain the lowest

Figure 3.4: Model fit



Notes: We simulate a sample of 1,000 individuals using the calibrated model.

throughout. We fit the model to the empirical data using maximum likelihood calibration. Figure 3.4 shows the overall agreement between the empirical data and a dataset simulated using the calibrated parameters within the support of the data. On the left, we show the choice probability of working in a blue-collar occupation while plotting the average wage across all occupations on the right. Overall, the values of the model's calibrated parameters are in broad agreement with the relevant literature. For example, individuals discount future utilities by 6% per year, and wages increase by about 7% with each additional year of schooling.

3.4 Respy performance review

EKW model software is formed by two large functional blocks: backward propagation and simulation. In terms of complexity, we can mostly disregard the simulation because it's essentially a query to $Emax$ database, which is created by a backward propagation block. Therefore, our research shall be mostly concentrated on designing and implementing backward propagation architecture for performance and scalability.

3.4.1 Initial code performance

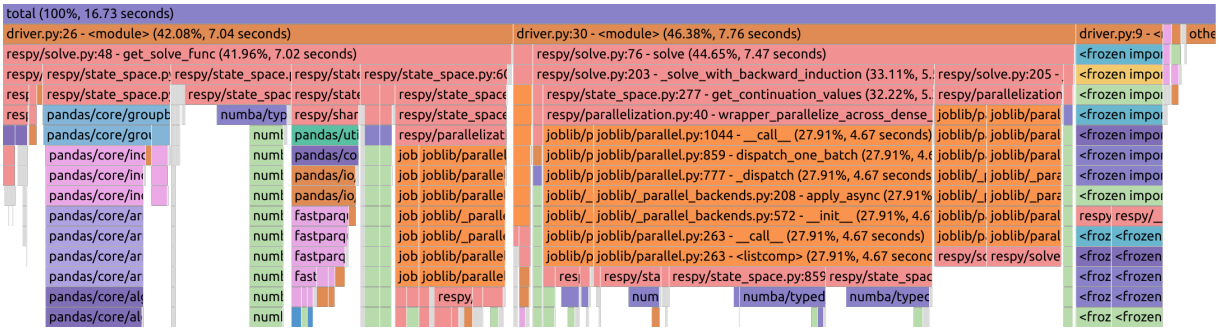
We construct two test cases based on well-known, influential, and empirically motivated model specifications to evaluate Respy's initial performance profiles. The first one `kw94_one` provided a benchmarking setup for the development of an approximate solution method for EKW models that is still in use today (Keane and Wolpin, 1994). The second one `kw97_advanced` was the first successful estimation of an EKW model

on empirical data and was already discussed in detail in the earlier sections.

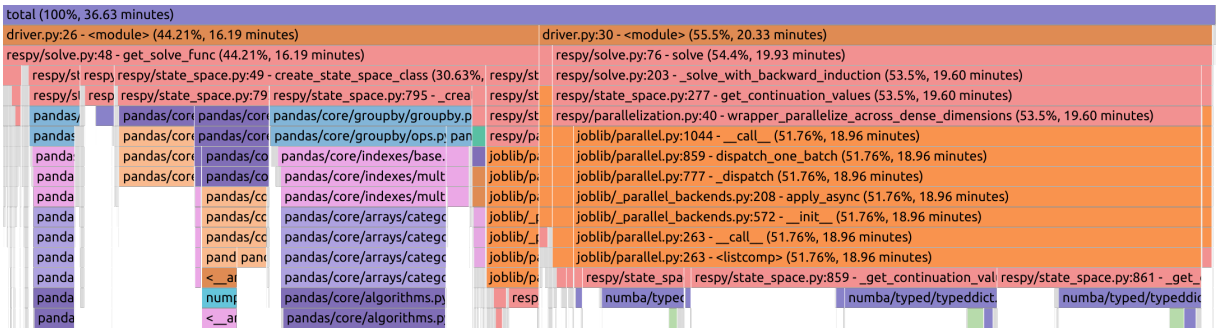
The evaluation results are presented in Fig. 3.5.

Figure 3.5: Initial Respy code performance profiles

(a) Test configuration `kw94_one` initial profile



(b) Test configuration `kw97_advanced` initial profile



The original version of Respy package deploys `pandas.DataFrame` to perform computationally intensive EKW operations. The user-defined callback functions for immediate rewards and state variables updates are defined as expressions in the model configuration provided by the user separately in a YAML file. These expressions are evaluated by `pandas.DataFrame` during the backward propagation procedure. The `pandas.DataFrame` is a tabular processor; it is tuned to operate the data organized similarly to an in-memory database. This design makes prototyping very fast, which is much appreciated by Python developers. On the other hand, due to `pandas.DataFrame`, the Respy hits multiple performance penalties:

1. Only the data physically stored in memory could be operated by `pandas.DataFrame`, however, the states could be computed and used on-the-fly.

2. The `pandas.DataFrame` data is always stored in separately-allocated columns, making any row-spanning calculations slower.

3. Compared to native C arrays or matrices, each `pandas.DataFrame` a column is a separate object whose data format is not compatible even with NumPy; as a result, any operation on this data is essentially several times slower than native C array data processing.

One possibility would be to swap `pandas.DataFrame` with `cuDF` by RapidsAI: a very similar database engine gets deployed on a GPU, with all data kept in GPU memory by default. We do not take this path, as it fundamentally has the same penalties, which just get slightly scaled but hit again at larger problem sizes.

In fact, the performance issues above could only be addressed by re-writing the EKW model code in C++. The next sections present how we move from algorithmic and hardware definitions to efficient CPU and GPU implementations.

3.5 Algorithmic definitions

EKW model engine could be expressed in terms of Combinatorics. In EKW, each *combination* of choices is a *state*, and the backward propagation process in a direct simulation of a combinatorial system over the entire *state space* of choices. The backward propagation process is similar to "picking a key" to a padlock with several digits. A program implementation of this method would require a *multiloop* – a stacked nest of indexed loops. A *multi-index* is a set of loops indices identifying each state of the multiloop.

Unlike a simple padlock, the state variables in the EKW model are limited by total sum, significantly reducing the resulting state space. The total sum of state variables values cannot exceed the number of periods. Practically, if an agent is allowed to make a choice each year during 60 years, then he cannot make more than 60 choices in total. Therefore, we refer to the set of state variables as *sum-constrained*.

We develop EKW models with *lagged actions* support. The concept of lagged actions is borrowed from regression modelling, where *lagged dependent variables* are used to express that the current state of a dependent variable is heavily determined by its past state. Likewise, in EKW the inclusion of lagged actions should better preserve the bias. As a result, lagged actions are becoming additional variables identifying the model state. However, unlike the usual state variables, they are not constrained by the total number of periods. Therefore, we refer to the set of lagged actions as *unconstrained*.

Driven by its combinatorial nature, the EKW model problem experiences a very rapid complexity growth with an increase of dimensions, which is often referred to as *the*

combinatorial explosion, or *the curse of dimensionality*. Leveraging large distributed computing environments such as supercomputers or grid systems remains the main method to solve this type of combinatorial problems directly.

In order to distribute the state space for a parallel system such as a GPU or supercomputer, we leverage two main combinatorial operations: *ranking* (indexing) of a given combination, and *sequencing* (restoring) a combination for a given lexicographical rank (index). In particular, a function returning an index of a given combination acts as a "hashing function" to access an array of data items stored for each combination. Similarly, a function that restores a combination from the given index, could be used to quickly partition the volume of iterations between the parallel workers, and provide each worker with a starting combination.

We discuss the flavors of combinatorial setups required for EKW backward propagation and derive the essential utility functions in Appendix A.

3.6 Hardware definitions

Due to the curse of dimensionality, the EKW model pushes the limits of the hardware. Given that the EKW model is a memory-bound type of problem, our goal is to preserve the optimal data throughput, which is in general identified by registers, caches, and dynamic random access memory.

Registers. The register file is the on-chip fast data storage of immediate and most frequently used elements of the execution context. For the EKW model, the execution context of each worker is identified by the current combination and its upper limits, QRNG state, the currently accumulated $Emax$ value, some other loop indices, such as the number of actions and the number of draws, and stack frames of the function calls.

Each CPU thread typically has access to 16 general-purpose scalar registers, as well as to about 16 general-purpose vector registers. In fact, the real number of registers in CPU hardware is much larger due to many pipeline optimization techniques applied over the years of development. However, the compilers could only operate the mentioned amount of "programmer-visible" registers, and push the excess data to the stack, which is mapped onto the cache memory. If the application code is not vectorized, the use of vector registers is very limited. The user may choose to sustain a bit more register space by artificially deploying vector registers as a storage space for scalars, but the overhead of accessing it is significant compared to the cache. Overall, the CPU could barely operate the EKW model of a few dimensions, using registers

alone, which does not happen in practice but makes much more sense for the GPU case explained below.

Each GPU thread may have access to up to 255 general-purpose registers. At the same time, each GPU multiprocessor has a fixed total number of registers available, effectively limiting the maximum number of threads in flight. A typical optimal register footprint that allows fair utilization of all GPU resources is ≈32 registers per thread. Any larger register consumption per thread would often limit the number of threads in flight and will not allow leveraging all available hardware compute units. Like on the CPU, excess registers could be spilled into the L2 cache, but the effective size of cache per thread is much smaller due to the much larger number of threads. For this reason, GPU applications are strongly discouraged from inducing register spilling in order to avoid dramatic performance penalties. Overall, the meaningful use of GPU could be maintained with at most ≈100 registers per thread, which could hold up to ≈40 EKW model dimensions.

Cache. The L2 cache sizes are similar on modern high-end CPUs and GPUs, but its purpose is different. While on the CPU, the use of cache is essential in almost any routine task, the use of L2 cache for user data on GPU is considered exceptional practice risky for performance, as the cache is quickly exhausted by a large number of in-flight threads. One legitimate use of cache on GPU is function frame support, which makes certain registers available in order to pass arguments to a function call, according to the calling convention (in CUDA terms this usecase is called *local memory*).

The EKW model should not make use of a cache for any data. The size of $Emax$ array is too large to fit into the cache, and its access pattern is *streaming*, that is, elements are accessed one time at most.

Dynamic random access memory (DRAM). GPU DRAM is a huge volatile storage, which is connected directly to the GPU chip. Like any DRAM, it still has a latency of hundereds of CPU cycles. Moreover, memory transactions have minimum possible size of 32 consecutive bytes. This DRAM property creates an important weak point for almost any EKW data structure design. Like in the Monte-Carlo method, a typical next period $Emax$ array access pattern consists of reading individual 4-byte values at random memory locations by each GPU thread. Therefore, the average memory efficiency would be ≈12%, and is hard to improve, unless each $Emax$ is identified by a set of multiple data records, or each memory access is acompanied by a large amount of computations.

GPU warp. The warp is a group of threads with consecutive thread indexes, which are

bundled together and executed simultaneously on a single GPU code. Each warp is fully executed on a single GPU core.

GPU block. The block is a group of threads with consecutive thread indices designed to work as a batch or an elemental subset of the user-defined problem. In the case of matrix operations, a block is usually a submatrix. At runtime, a thread block is divided into a number of warps for execution on the cores of an SM. The size of a warp depends on the hardware.

GPU compute grid. Compute grid is a GPU's approach to enumerate blocks of threads, similar to how OpenMP enumerates individual threads with *omp_get_thread_num()*. But unlike OpenMP, compute grid is a 3-dimensional enumeration implemented close to hardware level, so that both programmer and GPU could better handle grid-based problems, such as numerical simulations, image processing or tensor arithmetics. Problems that do not require multidimensional indexing could use just one dimension, and set the two others to 1, which is exactly the case of EKW.

Occupancy. The occupancy of a GPU is the number of warps running concurrently on a multiprocessor divided by the maximum number of warps that can run concurrently. This metric is often used to understand the saturation of GPU resource usage, such as threads, registers, and shared memory. The higher the occupancy, the better is usually the utilization of different GPU resources. However, occupancy maximization should not be set out as an ultimate goal per se, as there are exceptions.

GPU Shared memory. In addition to cache, GPU provides a programmable indexed cache called shared memory. This memory is shared among threads belonging to the same block. It amounts to at most 48 KBytes per block. As for the registers, the shared memory size per block impacts the total number of blocks in flight and should be chosen reasonably small. In the EKW model code, the shared memory could serve the following purposes. First of all, combinatorial algorithms leverage lookup tables to speedup ranking, sequencing, and popcount operations. Secondly, shared memory is often used as a scratch space for parallel reduction. Particularly, if the Monte Carlo loop is parallelized among multiple threads of a block, the $Emax$ value could be reduced in shared memory.

Kernel. GPU kernel is a special type of function, which is executed by a collective of GPU threads, according to the Single Instruction Multiple Threads (SIMT) paradigm. In the context of the EKW model, by kernels, we denote two particular kernels: the backward propagation kernel and the simulation kernel. When the EKW model is executed for a given problem setting, parameter values such as dimensions are inlined into the code to minimize the unknowns for compiler-driven optimizations. Yet, one

big unknown which remains is the user-defined callbacks, whose complexity cannot be estimated beforehand. While we leave the potential side-effects and optimization issues of callback to the user, we still ensure the work items distribution on the GPU to be most evenly aligned to the total number of in-flight blocks a GPU could handle simultaneously, using the occupancy calculator.

Schedule. By schedule, we denote the mapping between the compute grid and the problem-defined work items, which largely determines how the logical work items would be dispatched onto the hardware. Specifically, we have a degree of freedom in choosing how the combinations domain space should be divided among the threads. We define the schedule for both GPU and multithreaded CPU in a similar way, except that the optimal number of workers on GPU additionally takes into account the resources requested by a particular kernel. During the EKW code development, we have considered several scheduling strategies, which are presented in the following sections.

Warp divergence. Threads belonging to the same warp must execute the same instruction at all times. If the code paths of threads within the same warp diverge, their execution is serialized. That is, each thread executing a unique code path will be waited by all other threads of warp until the next common instruction.

Memory coalescing. Memory coalescing is an automatic mandatory procedure of "gathering" simultaneous memory operations requested by threads of a single warp. The GPU memory bus must convert multiple natural individual values reads and writes into larger transactions. The conversion is most efficient, when there is a way to gather requests from all threads of warp into a single 32/64/128-byte memory transaction, which incorporates all requested values. In other words, *memory coalescing is optimized for collective accessing of continuous memory regions.* As the memory I/O is two orders of magnitude slower than compute cores, it's very important to use it efficiently. In *EKW* we are able to address this requirement only partially. The current period of backward induction iterates through discrete choices, such that the $Emax$ values in memory are accessed continuously. However, the next period $Emax$ values are read at random, which results into poor coalescing. Our best hope is that the user could provide sufficiently "heavy" computations to be performed while the memory transaction is in flight.

EKW model has been deployed on NVIDIA A100 and AMD Instinct MI200 GPUs; the reference cards for this hardware could be found in NVIDIA (2020) and AMD (2020).

3.7 Multithreaded implementation

The best source of EKW model scalability is its huge state space. The generic parallelization scheme is therefore based on *partitioning* the state space among the compute units. The partitioning must be performed not only evenly, but also align with more specific hardware features, and choose the most suitable algorithmic trade-offs. For instance, the preferred lexicographically-ordered $Emax$ storage contradicts with an ability of GPU threads to execute fully independent code paths. For CPU-based multithreaded implementation this is not an issue, due to the massive branch processing (branch prediction) logic available in the modern x86 hardware. For this reason, we consider multithreaded implementation described here a baseline, and develop a more complex GPU implementation on top of it in the next section.

The generic multithreaded partitioning consists of the following steps:

- Calculate the volume (population count) of EKW model state space
- Calculate the even $(start, limit)$ intervals of state space to be handled by each individual compute unit (worker)

The model state space is calculated using binomial coefficients, with some corrections applied due to sum-constrained choices. The biggest challenge here is to avoid performance and integer overflow issues during the evaluation of $C_{n,k}$ binomial coefficients. We follow the method of Lemire et al. (2019) to minimize performance costs of integer division, predict the integer overflow and avoid it to the maximum possible extent.

Although the number of combinations to be processed by each worker is trivial to obtain, a more challenging task is to convert the starting combination index into the combination itself, which we call *sequencing*. Furthermore, sequencing needs to be really efficient, in order to be performed for thousands of working threads on GPU. We developed a sequencing function based on decomposing a given combination index into binomial coefficients, and optimized it with a cachable lookup table for best performance.

We provide an efficient and scalable generic partitioning method, which could be even parallelized by itself for large distributed systems.

3.8 GPU implementation

GPU implementation largely follows the multithreaded approach, but deals with further challenges due to the GPU hardware architecture:

- Calculate the optimal *schedule* for the given kernel, constrained by the GPU limitations
- The growth of register usage, due to the need to maintain the combinations iterator state
- Thread divergence problem in 1-1 mapping
- Alternative: give the whole warp to a single combination:
  - No warp divergence due to a separate loop on each thread
  - All threads of warp handle the same combination but collectively perform the parallel reduction of the Monte-Carlo sampling loop

Naive 1-1 mapping (Fig. 3.6) presents an illustrative example of how the maximum possible theoretical, algorithmic parallelism can effectively become the worst practical choice for the GPU architecture. In this schedule, each GPU thread is intended to perform the portion of work fully independently of each other. This idea is often applied in CPU multithreading, which indeed may work well, thanks to many hardware and software means. In GPU, threads are not fully independent by design. Threads belonging to the same warp execute synchronously and largely share the execution context. Threads of the same warp execute the same instruction at all times, while threads of different warps are independent and asynchronous. That is, on GPU, only the warps could be considered similar to CPU threads in the sense of independent execution. Given that threads of the same warp always execute the same instruction, they are not able to diverge their control flow, e.g., due to a different logical expression value. Instead, all threads of warp must visit all instructions of the logically divergent block of code, regardless of the actual need. During the execution of the unexpected instructions, the threads which must not take them are simply masked, or *stalled*, until the other threads of warp finish executing the same instructions as they intend to. As a result, in the event of control divergence, each thread of warp works twice: on his own instructions and on unexpected instructions in a stalled state. If there is more branching in the control flow, each thread will wait even longer until the control flow could finally return to an instruction common for all threads of the warp. In fact, nearly half of the EKW code is vulnerable to the divergence issue, given that any loop with dynamically changing bounds is diverging:

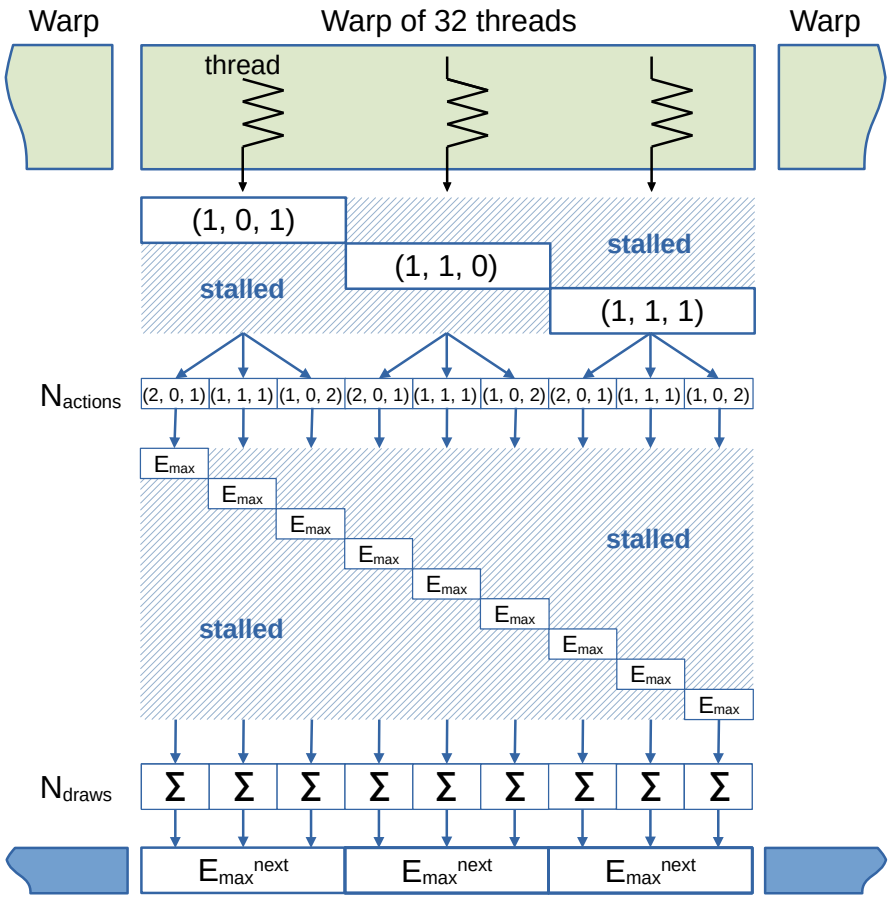


Figure 3.6: GPU execution flow with each thread processing independent work item, and the resulting stalls due to warp divergence.

- Combinations iterator: each thread runs its own (start, limit) interval of multi-loop, the loop boundary checks do not evaluate equally ⇒ divergence
- *Emax* value read: *Emax* index calculation is done by sequencing loop, which bounds are defined by the current combination, which is different on each thread ⇒ divergence

Fortunately, there still exist EKW code parts that can execute in parallel unaffected by the divergence issue:

- Monte Carlo loop: when the new *Emax* value is calculated, the loop iterates through a constant number of draws and can be expressed with a parallel reduction

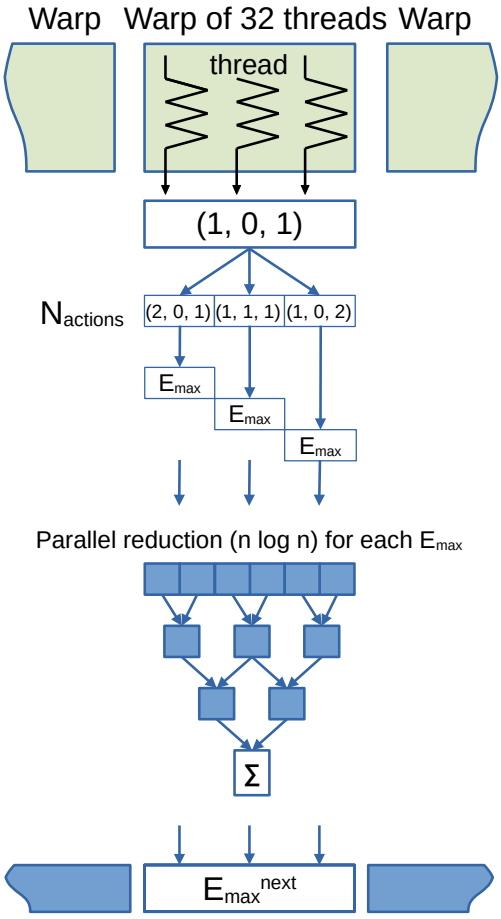


Figure 3.7: GPU execution flow with each warp processing independent work item and warp-shared parallel reduction of Monte Carlo loop.

- $Emax$ value write: $Emax$ writes are performed in order (the same lexicographic order, which is maintained by the combinations iterator)
- Random number generation with Sobol QRNG does not use any branching

The intermediate rewards and state variables updates are provided by the user, so their control flow is implementation-defined. Although the reference KW'94 and KW'97 examples do not add any noticeable divergence, it's highly expected that the real research code will, due to the need to express individual behavior with conditional decision-making patterns. We would like to study these complex cases carefully and offer practical recommendations enabling the efficient use of GPUs.

A less obvious scheduling strategy would be to keep the serial execution of code portions that do not parallelize efficiently due to the divergent execution stalls (Fig. 3.7).

3.8.1 Further GPU performance considerations

EKW model simulation is in fact an atypical kind of workload for a GPU. While the GPU hardware is designed to serve as a number cruncher, the EKW model actually does not do much computations by itself. The building of a EKW discrete choice tree has a lot of common with a class of graph traversal problems. We try to avoid some properties of graph traversal, most unfortunate for performance, such as a hierarcial data structure. But although we use plain arrays, the "look up a neighbour node" type of workload still makes the majority of the EKW model runtime. Overall, the following performance properties could be outlined:

- GPU does FP calculations; the only part of the code which does FP is the Monte-Carlo loop
- Monte-Carlo loop requires normalized QRNG
- GPU keeps $Emax$ in on-board GPU global memory:
  - $Emax$ writes are lexicographically ordered, i.e., the memory is filled with $Emax$ values continuously.
  - $Emax$ reads are random, only 4 bytes (float) per read.

    The memory read must always pass through the cache. The cache-streaming mode may help to minimize the cache pollution, when for example the local memory could benefit from caching on its own. On NVIDIA A100, the granularity of L2 memory transactions can be set to 32, 64, or 128 Bytes. Random accesses should use smaller granularity, in order to minimize over-fetch, we use `cudaDeviceSetLimit(cudaLimitMaxL2FetchGranularity, 32)`. And even with all these optimizations, at least 32 bytes of memory will be read per each actual read of 4 bytes.

3.9 Performance evaluation

For performance evaluation, we look into the medium-sized problem setup:

- The number of state variables: 4
- The number of lagged actions: 1
- The number of periods: 240

- The number of actions: 4
- The number of draws: 100

The EKW backward propagation process performance is presented in Listing 1. Here, the model is executed on an ordinary 4-core laptop, demonstrating the aggregate combinations processing rate of $\approx 3Mio$ per second.

```
p14s ~/h/e/T/e/build (emax)> ./test_eskew_emax --gtest_filter="*medium"
Note: Google Test filter = *medium
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from eskew_emax
[ RUN      ] eskew_emax.test_emax_medium
425167380 iterations in total, 8 workers, 53145923 iterations per worker
Using CPU backend with 8 workers
26633/425167380 (1877519 combs/sec)
3088322/425167380 (3044723 combs/sec)
7084827/425167380 (3512785 combs/sec)
11204528/425167380 (3713716 combs/sec)
15597975/425167380 (3882848 combs/sec)
20034970/425167380 (3993228 combs/sec)
...
420760659/425167380 (3162287 combs/sec)
422554635/425167380 (3152077 combs/sec)
424109318/425167380 (3140247 combs/sec)
425050178/425167380 (3124080 combs/sec)
[       OK ] eskew_emax.test_emax_medium (137379 ms)
[----------] 1 test from eskew_emax (137381 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (137381 ms total)
[  PASSED  ] 1 test.
```

Listing 1: Performance evaluation of medium-sized problem setup.

We designed the GPU backend execution to add no difference from the user perspective. As show in Listing 2, the only difference is a much larger number of workers.

3.10 Conclusion

We have designed and implemented an EKW model backward propagation core, which is able to solve medium-sized problems on an ordinary laptop computer in a few minutes. This result has been obtained without any GPU acceleration. It is sufficient to solve all “classical” EKW problems instantly and to solve $\times 100$ larger problems on a routine basis.

Although CPU and GPU versions of EKW model core software could share the same components, the stall-free execution schedule for GPU threads requires very careful

```
./test_combinations_partitioner_cuda
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from eskew_combinations
[ RUN      ] eskew_combinations.test_partitioner
300292131 iterations in total, 104414 workers, 2876 iterations per worker
Using NVIDIA GeForce RTX 3080 backend with 104414 workers
[       OK ] eskew_combinations.test_partitioner (4388 ms)
[----------] 1 test from eskew_combinations (4388 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (4388 ms total)
[  PASSED  ] 1 test.

./test_combinations_partitioner_hip
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from combinations
[ RUN      ] combinations.test_partitioner
300292131 iterations in total, 71669 workers, 4190 iterations per worker
Using Radeon RX Vega backend with 71669 workers
[       OK ] combinations.test_partitioner (3633 ms)
[----------] 1 test from combinations (3633 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (3633 ms total)
[  PASSED  ] 1 test.
```

Listing 2: Performance evaluation of a GPU-enabled model execution.

planning of *Emax* data layout and access ordering. We have considered multiple approaches to address this challenge and compared their characteristics. Overall, an efficient use of GPU shifts the prohibitively large problem size threshold towards the GPU DRAM size limit. In other words, the GPU version of the EKW model backward propagation is able to process the state space in very short times and is only limited by the size of GPU on-board memory, which typically does not exceed 80GB.

A Scalable iterations over constrained multiloops

A.1 Introduction

In combinatorics, a *state* of the system is identified by the *combination* of choices. A direct simulation of the system often requires iterating through the entire *state space* of choices. A program to "pick a key" to a combination lock with 3 digits by iterating through all combinations is shown in Listing 3. In this example, a *multiloop* is a stacked nest of indexed loops. A *multi-index* is a set of loops indices identifying each state of the multiloop.

By the term *scalability*, we denote an execution schedule and resource utilization, which could sustain a solution to the largest possible problem in the least time. In particular, scalability involves minimization of memory usage in order to push the performance bottleneck created by much smaller memory throughput compared to processor unit throughput. It could be facilitated by computing and consuming the immediate result "on-the-fly", or repeated computing of the same immediate result instead of storing it (often referred to as "trading compute for memory").

We propose a compute engine and its software implementation to iterate over the user-specified multiloops on a large distributed system, a supercomputer, or a grid system. Our implementation shall be scalable enough to be practically efficient on the largest currently available distributed systems.

A.2 Requirements

A.2.1 Parameters

In a simple case, a combinatorial problem is identified by the following 3 parameters:

```cpp
#include <array>
#include <iterator>
#include <iostream>

int main(int argc, char* argv[])
{
  const constexpr int m = 9;
  for (int i1 = 0; i1 <= m; i1++)
    for (int i2 = 0; i2 <=m; i2++)
      for (int i3 = 0; i3 <= m; i3++)
      {
        std::array combination = { i1, i2, i3 };
        std::copy(
          std::begin(combination),
          std::end(combination),
          std::ostream_iterator<int>(std::cout, " "));
        std::cout << std::endl;
      }

  return 0;
}
```

Listing 3: A program for iterating through all combinations of a combination lock with 3 digits.

- **k** - length of sequence.
- **m** - max allowed sequence element value.
- **Callable** - a user-defined function to be called on each combination.

As an example, we provide all combinations generated by $C_{k=3}^{m=2}$ in Listing 4.

$$
\begin{array}{llll}
(0,0,0)_0, & (0,0,1)_1, & (0,0,2)_2, & (0,1,0)_3, \\
(0,1,1)_4, & (0,1,2)_5, & (0,2,0)_6, & (0,2,1)_7, \\
(0,2,2)_8, & (1,0,0)_9, & (1,0,1)_{10}, & (1,0,2)_{11}, \\
(1,1,0)_{12}, & (1,1,1)_{13}, & (1,1,2)_{14}, & (1,2,0)_{15}, \\
(1,2,1)_{16}, & (1,2,2)_{17}, & (2,0,0)_{18}, & (2,0,1)_{19}, \\
(2,0,2)_{20}, & (2,1,0)_{21}, & (2,1,1)_{22}, & (2,1,2)_{23}, \\
(2,2,0)_{24}, & (2,2,1)_{25}, & (2,2,2)_{26} &
\end{array}
$$

Listing 4: All combinations of a problem $C_{k=3}^{m=2}$.

In addition to the basic setup, we provide a parameter to limit each combination with a given sum:

- **n** - sequence elements sum.

$$(0,2,2)_0,\quad (1,1,2)_1,\quad (1,2,1)_2,\quad (2,0,2)_3,$$
$$(2,1,1)_4,\quad (2,2,0)_5$$

Listing 5: All combinations of a problem ${}^{n=4}C^{m=2}_{k=3}$.

As an example, we provide all combinations generated by ${}^{n=4}C^{m=2}_{k=3}$ in Listing 5.

Furthermore, each sum-constrained iterator is provided in two variants:

- $\sum_{i=0}^{k} e_i = n$: combination elements sum must be equal to the given constraint.
- $\sum_{i=0}^{k} e_i \leq n$: combination elements sum must be less or equal to the given constraint.

Technically, $\sum_{i=0}^{k} e_i \leq n$ iterator is implemented as a set of $\sum_{i=0}^{k} e_i = n$ iterators executed one after another. The combinations generated by **sum_less_or_equal** case of ${}^{n\leq 4}C^{m=2}_{k=3}$ are shown in Listing 6.

$$(0,0,0)_0,\quad (0,0,1)_1,\quad (0,0,2)_2,\quad (0,1,0)_3,$$
$$(0,1,1)_4,\quad (0,1,2)_5,\quad (0,2,0)_6,\quad (0,2,1)_7,$$
$$(0,2,2)_8,\quad (1,0,0)_9,\quad (1,0,1)_{10},\quad (1,0,2)_{11},$$
$$(1,1,0)_{12},\quad (1,1,1)_{13},\quad (1,1,2)_{14},\quad (1,2,0)_{15},$$
$$(1,2,1)_{16},\quad (2,0,0)_{17},\quad (2,0,1)_{18},\quad (2,0,2)_{19},$$
$$(2,1,0)_{20},\quad (2,1,1)_{21},\quad (2,2,0)_{22}$$

Listing 6: All combinations of a problem ${}^{n\leq 4}C^{m=2}_{k=3}$.

Finally, we allow a combination to be joined of two independent sum-constrained and unconstrained parts:

- **k1** - length of sum-constrained sequence.
- **m1** - max allowed sum-constrained sequence element value.
- **k2** - length on unconstrained sequence.
- **m2** - max allowed unconstrained sequence element value.

In this most complex case, a problem ${}^{n\leq 4}C^{m1=2,m2=3}_{k1=3,k2=1}$ yields a set of combinations shown in Listing 7

$(0,0,0,0)_0$, $(0,0,0,1)_1$, $(0,0,0,2)_2$, $(0,0,0,3)_3$,
$(0,0,1,0)_4$, $(0,0,1,1)_5$, $(0,0,1,2)_6$, $(0,0,1,3)_7$,
$(0,0,2,0)_8$, $(0,0,2,1)_9$, $(0,0,2,2)_{10}$, $(0,0,2,3)_{11}$,
$(0,1,0,0)_{12}$, $(0,1,0,1)_{13}$, $(0,1,0,2)_{14}$, $(0,1,0,3)_{15}$,
$(0,1,1,0)_{16}$, $(0,1,1,1)_{17}$, $(0,1,1,2)_{18}$, $(0,1,1,3)_{19}$,
$(0,1,2,0)_{20}$, $(0,1,2,1)_{21}$, $(0,1,2,2)_{22}$, $(0,1,2,3)_{23}$,
$(0,2,0,0)_{24}$, $(0,2,0,1)_{25}$, $(0,2,0,2)_{26}$, $(0,2,0,3)_{27}$,
$(0,2,1,0)_{28}$, $(0,2,1,1)_{29}$, $(0,2,1,2)_{30}$, $(0,2,1,3)_{31}$,
$(0,2,2,0)_{32}$, $(0,2,2,1)_{33}$, $(0,2,2,2)_{34}$, $(0,2,2,3)_{35}$,
$(1,0,0,0)_{36}$, $(1,0,0,1)_{37}$, $(1,0,0,2)_{38}$, $(1,0,0,3)_{39}$,
$(1,0,1,0)_{40}$, $(1,0,1,1)_{41}$, $(1,0,1,2)_{42}$, $(1,0,1,3)_{43}$,
$(1,0,2,0)_{44}$, $(1,0,2,1)_{45}$, $(1,0,2,2)_{46}$, $(1,0,2,3)_{47}$,
$(1,1,0,0)_{48}$, $(1,1,0,1)_{49}$, $(1,1,0,2)_{50}$, $(1,1,0,3)_{51}$,
$(1,1,1,0)_{52}$, $(1,1,1,1)_{53}$, $(1,1,1,2)_{54}$, $(1,1,1,3)_{55}$,
$(1,1,2,0)_{56}$, $(1,1,2,1)_{57}$, $(1,1,2,2)_{58}$, $(1,1,2,3)_{59}$,
$(1,2,0,0)_{60}$, $(1,2,0,1)_{61}$, $(1,2,0,2)_{62}$, $(1,2,0,3)_{63}$,
$(1,2,1,0)_{64}$, $(1,2,1,1)_{65}$, $(1,2,1,2)_{66}$, $(1,2,1,3)_{67}$,
$(2,0,0,0)_{68}$, $(2,0,0,1)_{69}$, $(2,0,0,2)_{70}$, $(2,0,0,3)_{71}$,
$(2,0,1,0)_{72}$, $(2,0,1,1)_{73}$, $(2,0,1,2)_{74}$, $(2,0,1,3)_{75}$,
$(2,0,2,0)_{76}$, $(2,0,2,1)_{77}$, $(2,0,2,2)_{78}$, $(2,0,2,3)_{79}$,
$(2,1,0,0)_{80}$, $(2,1,0,1)_{81}$, $(2,1,0,2)_{82}$, $(2,1,0,3)_{83}$,
$(2,1,1,0)_{84}$, $(2,1,1,1)_{85}$, $(2,1,1,2)_{86}$, $(2,1,1,3)_{87}$,
$(2,2,0,0)_{88}$, $(2,2,0,1)_{89}$, $(2,2,0,2)_{90}$, $(2,2,0,3)_{91}$

Listing 7: All combinations of a problem ${}^{n\leq 4}C^{m_1=2,m_2=3}_{k_1=3,k_2=1}$.

Note: in all cases, combinations are always iterated strictly in lexicographical order. This property shall allow to develop of straight-forward conversion methods between a combination and its index (rank) in the lexicographical sequence of combinations. In Listings 4,5,6,7 the rank of each combination is denoted with a subscript.

A.3 Usability

Goals:

- Avoid the development of a new program code for each problem size.

A.4 Scalability

- Support iterations partitioning between a very large (≈100K) number of parallel workers.

A.4.1 Serial implementation

The combinations are iterated with a recursive templated function. Each nested call to a function adds a dimension until the specified sequence length is reached. Each function call iterates over a loop ranging from the starting element value to the maximum allowed value. The current element values of all frames of recursion are stacked into a sequence and are propagated to the next level of recursion as in functional Currying. As a result, a compiler may leverage recursion and loop unrolling on the constant ranges for better code optimization. The implementation is based on the generic C++17 code presented on StackOverflow.

A.4.2 Parallel implementation support

Iterations over a multiloop could be trivially partitioned by sub-dividing the outermost loop, for example, with OpenMP, as shown in Listing 8. In order to scale the parallel execution up to a larger number of workers, the loops could even be ‘collapsed‘ in the OpenMP sense, meaning that the ranges of all nested loops are linearized into a single range, which is then partitioned evenly between the workers. This procedure may work very well for generic cases but is not suitable here for two main reasons.

Firstly, in order to find a starting point for each worker from a collapsed loop nest, the original multi-dimensional index must be reconstructed out of a linearized flat index.

```
#pragma omp parallel for collapse(3)
for (int i1 = 0; i1 <= m; i1++)
  for (int i2 = 0; i2 <=m; i2++)
    for (int i3 = 0; i3 <= m; i3++)
    {
      std::array combination = { i1, i2, i3 };
      std::copy(
         std::begin(combination),
        std::end(combination),
        std::ostream_iterator<int>(std::cout, " "));
      std::cout << std::endl;
    }
```

Listing 8: Partitioning multiloop with OpenMP directive and `collapse` clause.

This procedure is formally simple but involves performing a lot of integer divisions, which are expensive, even in modern hardware. The overhead of determining the starting point could only be neglected if each worker handles a significant number of iterations.

Secondly, the sum-constrained appears to be a blocker for simple partitioning algorithms. The partitioning could be performed on an unconstrained problem first and constrained at a later step during the parallel execution. In this case, a huge volume of combinations could be skipped (discarded) at entry and introduce imbalances in the processing speed of individual workers.

As simple partitioning is not suitable for our setup, we allow the partitioning to be done separately. A separate engine shall determine a starting point for each worker, which will be read and used at the start of the execution. Therefore, we additionally provide parameters to run a given number of iterations, starting from the specified combination:

- **start** - a combination to start with.
- **limit** - a number of combinations to iterate from the start.

A.5 Lexicographical ranking of combinations

In this Section we describe the methods for ranking (indexing) of a given combination and for restoring a combination for a given lexicographical rank (index). These two operations are essential for the use of combinations in practical applications and distributed systems. In particular, a function returning an index of a given combination acts as a “hashing function” to access an array of data items stored for each combination. Similarly, a function that restores a combination from the given index, could be

used to quickly partition the volume of iterations between the parallel workers, and provide each worker with a starting combination.

A.5.1 Unconstrained ranking

The simplest case of finding the index of combination is solved on StackOverflow. The corresponding code for our software package is presented in Listing 9.

```
template<
  uint32_t k, // length of sequence
  uint32_t m  // max allowed sequence element value
>
static uint32_t rank(const uint32_t* sequence)
{
  // The rank of a single entry sequence is zero.
  if constexpr (k == 1) return 0;

  uint32_t mul = 1, result = 0;
  for (uint32_t i = 0; i < k; i++, mul *= (m + 1))
  {
    result += mul * sequence[k - i - 1];
  }
  return result;
}
```

Listing 9: A function for ranking unconstrained combinations.

The sequencing code is an exact inverse of the ranking code and is shown in Listing 10.

```
template<
  uint32_t k, // length of sequence
  uint32_t m  // max allowed sequence element value
>
static std::array<uint32_t, k> sequence(uint32_t rank)
{
  std::array<uint32_t, k> result;
  uint32_t mul = 1;
  for (uint32_t i = 0; i < k - 1; i++)
    mul *= (m + 1);
  for (uint32_t i = 0; i < k; i++, mul /= (m + 1))
  {
    result[i] = rank / mul;
    rank -= mul * result[i];
  }

  return result;
}
```

Listing 10: A function for sequencing unconstrained combinations.

A.5.2 Constrained ranking

Constrained ranking performance may benefit from caching `popcount` values for a certain problem in a lookup table, which could then be used for all subsequent calls to rank function for the same dimensions. This version of the algorithm is hereinafter referred to as *stateful*, while the non-cached version is referred to as *stateless*.

Ranking in $\sum_{i=0}^{k} e_i = n$ mode

The stateless algorithm for constrained combinations ranking has been initially proposed by Mike Earnest on Math.StackExchange, its C++ version is presented in Listing 11. The corresponding stateful implementation is presented in Listing 12.

```
template<
  uint32_t n, // sequence elements sum
  uint32_t k, // length of sequence
  uint32_t m  // max allowed sequence element value
>
static uint32_t rank(const uint32_t* sequence)
{
  using Combinations = combinations::sum_equal::Combinations;

  // Evaluate, each time omitting the leading term of the sequence.
  uint32_t result = 0;
  for (uint32_t i = 0, sum = n; i < k - 1; i++)
  {
    for (uint32_t j = 0; j < sequence[i]; j++)
      result += Combinations::popcount(sum - j, k - i - 1, m);

    sum -= sequence[i];
  }

  return result;
}
```

Listing 11: A stateless function for ranking constrained combinations in $\sum_{i=0}^{k} e_i = n$ mode (stateless version).

Ranking in $\sum_{i=0}^{k} e_i \leq n$ mode

A.5.3 Combined constrained and unconstrained ranking

Ranking in $\sum_{i=0}^{k} e_i = n$ mode

For the ranking function, we trivially combine the formulas for constrained and unconstrained cases and get a program in Listing 19.

```
template<
  uint32_t n, // sequence elements sum
  uint32_t k, // length of sequence
  uint32_t m  // max allowed sequence element value
>
class Rank<n, k, m>
{
  std::vector<uint64_t> dp_;

public :

  Rank() : dp_((k - 1) * n)
  {
    auto dp = reinterpret_cast<uint64_t(*)[n]>(dp_.data());

    using Combinations = combinations::sum_equal::Combinations;

    // Evaluate, each time omitting the leading term of the sequence.
    for (uint32_t j = 1; j < k; j++)
      for (uint32_t i = 1; i <= n; i++)
        dp[j - 1][i - 1] = Combinations::popcount(i, j, m);
  }

  auto rank(const uint32_t* sequence) const
  {
    auto dp = reinterpret_cast<const uint64_t(*)[n]>(dp_.data());

    // Evaluate, each time omitting the leading term of the sequence.
    uint64_t result = 0;
    for (uint32_t i = 0, sum = n; i < k - 1; i++)
    {
      for (uint32_t j = 0; j < sequence[i]; j++)
        result += dp[k - i - 2][sum - j - 1];

      sum -= sequence[i];
    }

    return result;
  }
};
```

Listing 12: A stateful function for ranking constrained combinations in $\sum_{i=0}^{k} e_i = n$ mode (stateless version).

```
template<
  uint32_t n, // sequence elements sum
  uint32_t k, // length of sequence
  uint32_t m  // max allowed sequence element value
>
static std::array<uint32_t, k> sequence(uint64_t rank)
{
  using Combinations = combinations::sum_equal::Combinations;

  std::array<uint32_t, k> result {};

  // https://math.stackexchange.com/a/4495062/945948
  uint32_t sum = n;
  for (uint32_t i = 0; (i < k - 1) && sum; i++)
  {
    // Find the largest value of "j" for which the sum of populations
    // is still below the rank value.
    result[i] = 0;
    for (int j = 0; (j <= std::min(m, sum)) && (result[i] < sum); j++)
    {
      auto count = Combinations::popcount(sum - j, k - i - 1, m);
      if (rank < count) break;

      rank -= count;
      result[i]++;
    }

    sum -= result[i];
  }

  result[k - 1] = sum;

  return result;
}
```

Listing 13: A stateless function for sequencing constrained combinations in $\sum_{i=0}^{k} e_i = n$ mode.

```
template<
  uint32_t n, // sequence elements sum
  uint32_t k, // length of sequence
  uint32_t m  // max allowed sequence element value
>
class Sequence<n, k, m>
{
  std::vector<uint32_t> dp_;

public :

  Sequence() : dp_((n + 1) * (m + 1) * (k + 1))
  {
    auto dp = reinterpret_cast<uint32_t(*)[m + 1][k + 1]>(dp_.data());

    std::vector<uint32_t> dp_sum_((n + 1) * (k + 1));
    auto dp_sum = reinterpret_cast<uint32_t(*)[k + 1]>(dp_sum_.data());

    for (uint32_t x = 0; (x <= n) && (x <= m); x++)
      dp[x][x][1] = 1;

    for (uint32_t l = 0; l < 2; l++)
      for (uint32_t s = 0; s <= n; s++)
        for (uint32_t x = 0; x <= m; x++)
          dp_sum[s][l] += dp[s][x][l];

    for (uint32_t l = 2; l <= k; l++)
      for (uint32_t s = 0; s <= n; s++)
        for (uint32_t x = 0; (x <= s) && (x <= m); x++)
        {
          dp[s][x][l] = dp_sum[s - x][l - 1];
          dp_sum[s][l] += dp[s][x][l];
        }
  }

  auto sequence(uint64_t rank) const
  {
    auto dp = reinterpret_cast<const uint32_t(*)[m + 1][k + 1]>(dp_.data());

    ++rank;
    uint32_t s = n;
    std::array<uint32_t, k> result;

    for (uint32_t i = 0; i < k; i++)
    {
      for (uint32_t x = 0; x <= m; x++)
      {
        auto cur = dp[s][x][k - i];
        if (cur < rank)
        {
          rank -= cur;
          continue;
        }

        s -= x;
        result[i] = x;
        break;
      }
    }

    return result;
  }
};
```

Listing 14: A stateful function for sequencing constrained combinations in $\sum_{i=0}^{k} e_i = n$ mode.

```
template<
  uint32_t n, // sequence elements sum
  uint32_t k, // length of sequence
  uint32_t m  // max allowed sequence element value
>
static uint32_t rank(const uint32_t* sequence)
{
  using Combinations = combinations::sum_equal::Combinations;

  // The rank of a single entry sequence is this entry itself.
  if constexpr (k == 1) return sequence[0];

  // Evaluate, each time omitting the leading term of the sequence.
  uint32_t result = 0;
  for (uint32_t i = 0, sum = n; i < k; i++)
  {
    for (uint32_t j = 0; j < sequence[i]; j++)
    {
      for (uint32_t s = 0; s <= sum - j; s++)
        result += Combinations::popcount(s, k - i - 1, m);
    }

    sum -= sequence[i];
  }

  return result;
}
```

Listing 15: A stateless function for ranking constrained combinations in $\sum_{i=0}^{k} e_i \leq n$ mode.

For the sequencing function, we need to split the combination into two diverse parts. All combinations are divided into groups by the first constrained part because having different constrained parts already defines the order between two sequences. And we also know that in each group, sequences differ only by unconstrained part, and for every unique constrained part, there are exactly $(m_2+1)^{k_2}$ different unconstrained parts. And then, you can represent the problem of ranking combined sequences as some problem of encoding positions in the matrix to an integer. Each (i, j) element of $n \times m$ matrix could be mapped onto a single integer as $i \cdot m + j$. Now imagine that the sequences from the same group are written in the same rows, and then different sequences in the same row are written in different columns, and the same logic is used.

Integer division and modulo operations are costly but are required because the combinations are handled as numbers in base (m_2+1), similar to ranking, which converts a number in base (m_2+1) to a number in base 10.

Listing 20

```
template<
  uint32_t n, // sequence elements sum
  uint32_t k, // length of sequence
  uint32_t m  // max allowed sequence element value
>
class Rank<n, k, m>
{
  bool fits32bit = true;
  std::vector<uint64_t> dp_;

public :

  Rank() : dp_(k * (n + 1) * n)
  {
    auto dp = reinterpret_cast<uint64_t(*)[n + 1][n]>(dp_.data());

    using Combinations = combinations::sum_equal::Combinations;

    std::vector<uint64_t> dp_sum(n + 1);

    // Evaluate, each time omitting the leading term of the sequence.
    for (uint32_t j = 0; j < k; j++)
    {
      // Prefix sum.
      dp_sum[0] = Combinations::popcount(0, j, m);
      for (uint32_t i = 1; i <= n; i++)
        dp_sum[i] = dp_sum[i - 1] + Combinations::popcount(i, j, m);

      // Triangle lookup table.
      for (uint32_t sum = 1; sum <= n; sum++)
      {
        dp[j][sum][sum - 1] = dp_sum[sum];
        for (uint32_t i = sum - 1; i > 0; i--)
          dp[j][sum][i - 1] = dp_sum[i] + dp[j][sum][i];
      }
    }

    for (auto& value : dp_)
    {
      if (value <= std::numeric_limits<uint32_t>::max())
        continue;

      fits32bit = false;
      break;
    }

    if (fits32bit)
    {
      auto dp64bit = dp_;
      auto dp32bit = reinterpret_cast<uint32_t*>(dp_.data());
      for (int i = 0; i < dp_.size(); i++)
        dp32bit[i] = static_cast<uint32_t>(dp_[i]);
    }
  }

  auto rank(const uint32_t* sequence) const
  {
    // The rank of a single entry sequence is this entry itself.
    if constexpr (k == 1) return sequence[0];

    uint64_t result = 0;
    if (fits32bit)
    {
      auto dp = reinterpret_cast<const uint32_t(*)[n + 1][n]>(dp_.data());

      // Evaluate, each time omitting the leading term of the sequence.
      for (uint32_t i = 0, sum = n; i < k; i++)
      {
        if (sequence[i] == 0) continue;

        result += dp[k - i - 1][sum][sum - sequence[i]];
        sum -= sequence[i];
      }
    }
    else
    {
      auto dp = reinterpret_cast<const uint64_t(*)[n + 1][n]>(dp_.data());
```

```
template<
  uint32_t n, // sequence elements sum
  uint32_t k, // length of sequence
  uint32_t m  // max allowed sequence element value
>
static std::array<uint32_t, k> sequence(uint64_t rank)
{
  using Combinations = combinations::sum_equal::Combinations;

  std::array<uint32_t, k> result;

  // https://math.stackexchange.com/a/4495062/945948
  uint32_t sum = n;
  for (uint32_t i = 0; i < k - 1; i++)
  {
    // Find the largest value of "j" for which the sum of populations
    // is still below the rank value.
    int j = 0, asum = 0;
    for (j = 0; j <= std::min(m, sum); j++)
    {
      auto asum_old = asum;
      for (uint32_t s = 0; s <= sum - j; s++)
      {
        auto count = Combinations::popcount(s, k - i - 1, m);
        asum += count;
        if (asum > rank)
        {
          asum = asum_old;
          j++;
          goto finish;
        }
      }
    }

  finish :
    j--;
    result[i] = j;
    sum -= j;
    rank -= asum;
  }

  result[k - 1] = rank;

  return result;
}
```

Listing 17: A stateless function for sequencing constrained combinations in $\sum_{i=0}^{k} e_i \leq n$ mode.

```
template<
  uint32_t n, // sequence elements sum
  uint32_t k, // length of sequence
  uint32_t m  // max allowed sequence element value
>
class Sequence<n, k, m>
{
  std::vector<uint32_t> dp_;

public :

  Sequence() : dp_((n + 1) * (m + 1) * (k + 1))
  {
    auto dp = reinterpret_cast<uint32_t(*)[m + 1][k + 1]>(dp_.data());

    std::vector<uint32_t> dp_sum_((n + 1) * (k + 1));
    auto dp_sum = reinterpret_cast<uint32_t(*)[k + 1]>(dp_sum_.data());

    for (uint32_t x = 0; (x <= n) && (x <= m); x++)
      dp[x][x][1] = 1;

    for (uint32_t l = 0; l < 2; l++)
      for (uint32_t s = 0; s <= n; s++)
        for (uint32_t x = 0; x <= m; x++)
          dp_sum[s][l] += dp[s][x][l];

    for (uint32_t l = 2; l <= k; l++)
      for (uint32_t s = 0; s <= n; s++)
        for (uint32_t x = 0; (x <= s) && (x <= m); x++)
        {
          dp[s][x][l] = dp_sum[s - x][l - 1];
          dp_sum[s][l] += dp[s][x][l];
        }

    for (uint32_t x = 0; x <= m; x++)
      for (uint32_t l = 0; l <= k; l++)
        for (uint32_t s = 1; s <= n; s++)
          dp[s][x][l] += dp[s - 1][x][l];
  }

  auto sequence(uint64_t rank) const
  {
    auto dp = reinterpret_cast<const uint32_t(*)[m + 1][k + 1]>(dp_.data());

    ++rank;
    uint32_t s = n;
    std::array<uint32_t, k> result;

    for (uint32_t i = 0; i < k; i++)
      for (uint32_t x = 0; x <= m; x++)
      {
        auto cur = dp[s][x][k - i];
        if (cur < rank)
        {
          rank -= cur;
          continue;
        }
        s -= x;
        result[i] = x;
        break;
      }

    return result;
  }
};
```

Listing 18: A stateful function for sequencing constrained combinations in $\sum_{i=0}^{k} e_i \leq n$ mode.

```
template<
  uint32_t n,  // sum-constrained sequence elements sum
  uint32_t k1, // length of sum-constrained sequence
  uint32_t m1, // max allowed sum-constrained sequence element value
  uint32_t k2, // length of unconstrained sequence
  uint32_t m2  // max allowed unconstrained sequence element value
>
static uint32_t rank(const uint32_t* sequence)
{
  auto result1 = rank<n, k1, m1>(sequence);

  // https://stackoverflow.com/questions/56695041/it-is-possible-to-get-the-index-of-a-combination-without-generating-it
  uint32_t mul = 1;
  uint64_t result2 = 0;
  for (uint32_t i = 0; i < k2; i++, mul *= (m2 + 1))
    result2 += mul * sequence[k1 + (k2 - i - 1)];

  return result1 * mul + result2;
}
```

Listing 19: A stateless function for ranking combined constrained and unconstrained combinations in $\sum_{i=0}^{k} e_i = n$ mode.

```
template<
  uint32_t n,  // sum-constrained sequence elements sum
  uint32_t k1, // length of sum-constrained sequence
  uint32_t m1, // max allowed sum-constrained sequence element value
  uint32_t k2, // length of unconstrained sequence
  uint32_t m2  // max allowed unconstrained sequence element value
>
static std::array<uint32_t, k1 + k2> sequence(uint32_t rank)
{
  auto popcount2 = combinations::Combinations::popcount(k2, m2);

  auto result2 = sequence<n, k1, m1>(rank / popcount2);
  std::array<uint32_t, k1 + k2> result;
  memcpy(result.data(), result2.data(), result2.size() * sizeof(result2[0]));

  rank %= popcount2;

  for (uint32_t i = 0; i < k2; i++)
  {
    if (!rank)
    {
      result[k1 + k2 - i - 1] = 0;
      continue;
    }

    result[k1 + k2 - i - 1] = rank % (m2 + 1);
    rank /= (m2 + 1);
  }

  return result;
}
```

Listing 20: A stateless function for sequencing combined constrained and unconstrained combinations in $\sum_{i=0}^{k} e_i = n$ mode.

```
template<
  uint32_t n, // sum-constrained sequence elements sum
  uint32_t k1, // length of sum-constrained sequence
  uint32_t m1, // max allowed sum-constrained sequence element value
  uint32_t k2, // length of unconstrained sequence
  uint32_t m2 // max allowed unconstrained sequence element value
>
class Sequence<n, k1, m1, k2, m2>
{
  std::vector<uint32_t> dp_;

  uint64_t popcount2;

public :

  Sequence() : dp_((k1 + 1) * (m1 + 1) * (n + 1)),
    popcount2(combinations::Combinations::popcount(k2, m2))
  {
    auto dp = reinterpret_cast<uint32_t(*)[m1 + 1][n + 1]>(dp_.data());
    dp[k1][0][0] = 1;

    {
      std::vector<uint32_t> dp_sum_((k1 + 1) * (n + 1));
      auto dp_sum = reinterpret_cast<uint32_t(*)[n + 1]>(dp_sum_.data());

      for (uint32_t i = k1; i >= 1; i--)
      {
        for (uint32_t s = 0; s <= n; s++)
          for (uint32_t d = 0; d <= m1; d++)
            dp_sum[i][s] += dp[i][d][s];

        for (uint32_t d = 0; d <= m1; d++)
          for (uint32_t s = 0; s <= n; s++)
            if (d + s <= n)
              dp[i - 1][d][d + s] += dp_sum[i][s];
      }
    }
  }

  auto sequence(uint64_t rank) const
  {
    auto dp = reinterpret_cast<const uint32_t(*)[m1 + 1][n + 1]>(dp_.data());

    std::array<uint32_t, k1 + k2> result;

    uint64_t id1 = rank / popcount2;
    uint64_t id2 = rank % popcount2;
    id1++;

    for (uint32_t i = 0, sum = n; i < k1; i++)
    {
      uint32_t d = 0;
      for ( ; d <= m1; d++)
      {
        uint32_t cur = dp[i][d][sum];
        if (cur >= id1) break;

        id1 -= cur;
      }

      result[i] = d;
      sum -= d;
    }

    for (int i = 0; i < k2; i++)
    {
      result[k1 + k2 - i - 1] = id2 % (m2 + 1);
      id2 /= m2 + 1;
    }

    return result;
  }
};
```


Listing 21: A stateful function for sequencing combined constrained and unconstrained combinations in $\sum^{k} e_i = n$ mode

Ranking in $\sum_{i=0}^{k} e_i \leq n$ mode

```
template<
  uint32_t n,  // sum-constrained sequence elements sum
  uint32_t k1, // length of sum-constrained sequence
  uint32_t m1, // max allowed sum-constrained sequence element value
  uint32_t k2, // length of unconstrained sequence
  uint32_t m2  // max allowed unconstrained sequence element value
>
static uint32_t rank(const uint32_t* sequence)
{
  uint32_t result1 = rank<n, k1, m1>(sequence);

  uint32_t result2 = 0, mul = 1;
  for (uint32_t i = 0; i < k2; i++, mul *= (m2 + 1))
  {
    result2 += mul * sequence[k1 + (k2 - i - 1)];
  }

  return result1 * mul + result2;
}
```

Listing 22: A stateless function for ranking combined constrained and unconstrained combinations in $\sum_{i=0}^{k} e_i \leq n$ mode.

A.6 Usage

The combinations engine could be deployed either from the C++ or from the Python code. In C++, the user-defined function plugs into the engine as a lambda function, as shown in Listing 24. In Python, the user may choose to plug a Python function directly or to plug an embedded C++ function, which has an infrequently-used connection to Python objects of the main code. In the latter case, performance will be better because the Python interpreter will not be involved in each iteration of the multiloop. The conservation of end-to-end native binary representation of a combinatorial program code is an essential prerequisite for the best performance. We do, however, acknowledge the importance of Python scripting in research code; therefore, we believe the proposed interfacing option could be a reasonable compromise.

```
template<
  uint32_t n,  // sum-constrained sequence elements sum
  uint32_t k1, // length of sum-constrained sequence
  uint32_t m1, // max allowed sum-constrained sequence element value
  uint32_t k2, // length of unconstrained sequence
  uint32_t m2  // max allowed unconstrained sequence element value
>
static std::array<uint32_t, k1 + k2> sequence(uint32_t rank)
{
  auto popcount2 = combinations::Combinations::popcount(k2, m2);

  auto result2 = sequence<n, k1, m1>(rank / popcount2);
  std::array<uint32_t, k1 + k2> result;
  memcpy(result.data(), result2.data(), result2.size() * sizeof(result2[0]));

  rank %= popcount2;

  for (uint32_t i = 0; i < k2; i++)
  {
    if (!rank)
    {
      result[k1 + k2 - i - 1] = 0;
      continue;
    }

    result[k1 + k2 - i - 1] = rank % (m2 + 1);
    rank /= (m2 + 1);
  }

  return result;
}
```

Listing 23: A stateless function for sequencing combined constrained and unconstrained combinations in $\sum_{i=0}^{k} e_i \leq n$ mode.

```cpp
#include "combinations/combinations.h"
#include <iostream>

int main(int argc, char* argv[])
{
    // Define the combinations dimensions
    // (must be compile-time constants).
    constexpr const uint32_t
        length_of_sequence = 3,
        max_allowed_sequence_element_value = 4;

    // Generate combinations on-the-fly and consume
    // (use) each combination right away, just once
    // it is generated.
    combinations::Combinations::iterate<
        length_of_sequence,
        max_allowed_sequence_element_value
        >([&](auto... args)
    {
        // Use the current combination (here, we just
        // print it via C++17 parameter pack expansion).
        ((std::cout << args << ' '), ...);
        std::cout << std::endl;
    });

    return 0;
}
```

Listing 24: A usage example of the combinations API.

Bibliography

Aguirregabiria, V. and Mira, P. (2010). Dynamic discrete choice structural models: A survey. *Journal of Econometrics*, 156(1):38–67.

Aldrich, E. M. (2014). Chapter 10 - GPU computing in economics. In Schmedders, K. and Judd, K. L., editors, *Handbook of Computational Economics Vol. 3*, volume 3 of *Handbook of Computational Economics*, pages 557–598. Elsevier.

AMD (2020). Introducing AMD CDNA™ 2 architecture. https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf. [Online; accessed 28-November-2022].

Auerbach, A. and Kotlikoff, L. (1987). *Dynamic Fiscal Policy*. Cambridge University Press.

Backus, D. K., Kehoe, P. J., and Kydland, F. E. (1992). International real business cycles. *Journal of Political Economy*, pages 745–775.

Barillas, F. and Fernández-Villaverde, J. (2007). A generalization of the endogenous grid method. *Journal of Economic Dynamics and Control*, 31(8):2698–2712.

Becker, G. S. (1964). *Human Capital*. Columbia University Press, New York City, NY, USA.

Bell, N. and Hoberock, J. (2011). Thrust: A productivity-oriented library for CUDA. *GPU Computing Gems*, 7.

Bellman, R. (1961). *Adaptive Control Processes: A Guided Tour*. Rand Corporation. Research studies. Princeton University Press, USA.

Bellman, R. E. (1954). The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503–515.

Bengui, J., Mendoza, E. G., and Quadrini, V. (2013). Capital mobility and international sharing of cyclical risk. *Journal of Monetary Economics*, 60(1):42–62.

Bilbiie, F. O. (2008). Limited asset markets participation, monetary policy and (inverted) aggregate demand logic. *Journal of economic theory*, 140(1):162–196.

Blundell, R., Costa Dias, M., Meghir, C., and Shaw, J. (2016). Female labor supply, human capital, and welfare reform. *Econometrica*, 84(5):1705–1753.

Bokanowski, O., Garcke, J., Griebel, M., and Klompmaker, I. (2013). An adaptive sparsegrid semi-lagrangian scheme for front propagation. *J. Scient. Comput.*, 55(3):575—-605. also available as INS Preprint No. 1207.

Borella, M., De Nardi, M., and Yang, F. (2019). Are marriage-related taxes and social security benefits holding back female labor supply? Working Paper 26097, National Bureau of Economic Research.

Brumm, J. and Grill, M. (2014). Computing equilibria in dynamic models with occasionally binding constraints. *Journal of Economic Dynamics and Control*, 38:142–160.

Brumm, J., Kubler, F., and Scheidegger, S. (2017). *Computing Equilibria in Dynamic Stochastic Macro-Models with Heterogeneous Agents*, volume 2 of *Econometric Society Monographs*, pages 185–230. Cambridge University Press.

Brumm, J., Mikushin, D., Scheidegger, S., and Schenk, O. (2015). Scalable high-dimensional dynamic stochastic economic modeling. *Journal of Computational Science*, 11:12–25.

Brumm, J. and Scheidegger, S. (2014). Using adaptive sparse grids to solve high-dimensional dynamic models. *Working Paper – Available at SSRN 2349281*.

Brumm, J. and Scheidegger, S. (2017). Using adaptive sparse grids to solve high-dimensional dynamic models. *Econometrica*, 85(5):1575–1612.

Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P. (2004). Brook for gpus: Stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 777–786, New York, NY, USA. ACM.

Bungartz, H.-J. and Dirnstorfer, S. (2003). Multivariate quadrature on adaptive sparse grids. *Computing*, 71:89–114.

Bungartz, H.-J. and Griebel, M. (2004). Sparse grids. *Acta Numerica*, 13:1–123.

Bureau of Labor Statistics (2019). *National Longitudinal Survey of Youth 1979 cohort, 1979-2016 (rounds 1-27)*. Center for Human Resource Research, Ohio State University, Columbus, OH, USA.

Cai, Y., Judd, K. L., and Lontzek, T. S. (2015). The social cost of carbon with economic and climate risks. *ArXiv e-prints*.

Cai, Y., Judd, K. L., Thain, G., and Wright, S. J. (2013). Solving dynamic programming problems on a computational grid. *Computational Economics*, pages 1–24.

Christiano, L. J. and Fisher, J. D. (2000). Algorithms for solving dynamic models with occasionally binding constraints. *Journal of Economic Dynamics and Control*, 24(8):1179–1232.

Datta, M., Mirman, L. J., Morand, O. F., and Reffett, K. L. (2005). Markovian equilibrium in infinite horizon economies with incomplete markets and public policy. *Journal of Mathematical Economics*, 41(4):505–544.

David S. Bizer, K. L. J. (1989). Taxation and uncertainty. *The American Economic Review*, 79(2):331–336.

Davidson, R. and MacKinnon, J. G. (2003). *Econometric theory and methods*. Oxford University Press, New York City, NY, USA.

Debortoli, D. and Galí, J. (2017). Monetary policy with heterogeneous agents: Insights from tank models. *Manuscript, September*.

Deftu, A. and Murarasu, A. (2013). Optimization techniques for dimensionally truncated sparse grids on heterogeneous systems. In *21st Euromicro International Conference on Parallel,Distributed and Network-Based Processing*, pages 351–358. IEEE.

Den Haan, W. J., Judd, K. L., and Juillard, M. (2011). Computational suite of models with heterogeneous agents ii: Multi-country real business cycle models. *Journal of Economic Dynamics and Control*, 35(2):175–177.

Diamond, P. A. (1965). National debt in a neoclassical growth model. *American Economic Review*, 55(5):1126–1150.

Dongarra, J. J. and van der Steen, A. J. (2012). High-performance computing systems: Status and outlook. *Acta Numerica*, 21:379–474.

Dou, W. W., Fang, X., Lo, A. W., and Uhlig, H. (2017). Comparing solution methodologies for macro-finance models with nonlinear dynamics. Unpublished results.

Evans, R. W., Kotlikoff, L. J., and Phillips, K. L. (2012). Game over: Simulating unsustainable fiscal policy. Working Paper 17917, National Bureau of Economic Research.

Feldstein, M. and Liebman, J. (2001). Social security. NBER Working Papers 8451, National Bureau of Economic Research, Inc.

Fernández-Villaverde, J., Rubio-Ramírez, J., and Schorfheide, F. (2016). None. volume 2 of *Handbook of Macroeconomics*, pages 527 – 724. Elsevier.

Gabler, J. and Raabe, T. (2020). respy - a framework for the simulation and estimation of eckstein-keane-wolpin models.

Gaikwad, A. and Toke, I. M. (2009). GPU based sparse grid technique for solving multidimensional options pricing PDEs. In *Proceedings of the 2nd Workshop on High Performance Computational Finance*, WHPCF '09, pages 6:1–6:9, New York, NY, USA. ACM.

Garcke, J. and Griebel, M. (2012). *Sparse Grids and Applications*. Lecture Notes in Computational Science and Engineering Series. Springer.

Gilboa, I. (2009). *Theory of Decision under Uncertainty*. Cambridge University Press, New York City, NY, USA.

Gourieroux, C. and Monfort, A. (1996). *Simulation-Based Econometrics*. Oxford University Press, Oxford, United Kingdom.

Hager, C., Hüeber, S., and Wohlmuth, B. (2010). Numerical techniques for the valuation of basket options and its greeks. *J. Comput. Fin.*, 13(4):1–31.

Hasanhodzic, J. and Kotlikoff, L. J. (2013). Generational risk - is it a big deal?: Simulating an 80-period olg model with aggregate shocks. Working Paper 19179, National Bureau of Economic Research.

Heene, M., Kowitz, C., and Pflüger, D. (2013). Load balancing for massively parallel computations with the sparse grid combination technique. In *PARCO*, pages 574–583.

Hegland, M. (2003). Adaptive sparse grids. *Anziam Journal*, 44:C335–C353.

Heinecke, A. and Pflueger, D. (2013). Emerging architectures enable to boost massively parallel data mining using adaptive sparse grids. *International Journal of Parallel Programming*, 41(3):357–399.

Hintermaier, T. and Koeniger, W. (2010). The method of endogenous gridpoints with occasionally binding constraints among endogenous variables. *Journal of Economic Dynamics and Control*, 34(10):2074–2088.

Holtz, M. (2011). *Sparse Grid Quadrature in High Dimensions with Applications in Finance and Insurance*. Lecture Notes in Computational Science and Engineering. Springer, Dordrecht.

Hupp, P., Jacob, R., Heene, M., Pflüger, D., and Hegland, M. (2013). Global communication schemes for the sparse grid combination technique. In *PARCO*, pages 564–573.

Judd, K. L. (1998). *Numerical methods in economics*. Scientific and Engineering. The MIT press, Cambridge, Massachusetts, USA.

Judd, K. L., Maliar, L., Maliar, S., and Valero, R. (2014). Smolyak method for solving dynamic economic models: Lagrange interpolation, anisotropic grid and adaptive domain. *Journal of Economic Dynamics and Control*, 44:92–123.

Kaplan, G., Moll, B., and Violante, G. L. (2018). Monetary policy according to hank. *American Economic Review*, 108(3):697–743.

Keane, M. P. and Wolpin, K. I. (1994). The solution and estimation of discrete choice dynamic programming models by simulation and interpolation: Monte Carlo evidence. *Review of Economics and Statistics*, 76(4):648–672.

Keane, M. P. and Wolpin, K. I. (1997). The career decisions of young men. *Journal of Political Economy*, 105(3):473–522.

Klimke, A. and Wohlmuth, B. (2005). Algorithm 847: Spinterp: piecewise multilinear hierarchical sparse grid interpolation in matlab. *ACM Trans. Math. Softw.*, 31(4):561–579.

Kollmann, R., Maliar, S., Malin, B. A., and Pichler, P. (2011). Comparison of solutions to the multi-country real business cycle model. *Journal of Economic Dynamics and Control*, 35(2):186–202. Computational Suite of Models with Heterogeneous Agents II: Multi-Country Real Business Cycle Models.

Krueger, D. and Kubler, F. (2004). Computing equilibrium in OLG models with stochastic production. *Journal of Economic Dynamics and Control*, 28(7):1411–1436.

Krueger, D. and Kubler, F. (2006). Pareto-Improving Social Security Reform when Financial Markets are Incomplete!? *American Economic Review*, 96(3):737–755.

Krueger, D., Mitman, K., and Perri, F. (2016). Chapter 11 - macroeconomics and household heterogeneity. volume 2 of *Handbook of Macroeconomics*, pages 843 – 921. Elsevier.

Kydland, F. E. and Prescott, E. C. (1982). Time to build and aggregate fluctuations. *Econometrica: Journal of the Econometric Society*, pages 1345–1370.

Landau, L., Lifšic, E., Ziesche, P., and Pitaevskij, L. (2007). *Hydrodynamik*. Lehrbuch der theoretischen Physik : in 10 Bänden / L.D. Landau; E.M. Lifschitz. In dt. Sprache hrsg. von Paul Ziesche. Deutsch.

Lemire, D., Kaser, O., and Kurz, N. (2019). Faster remainder by direct computation: Applications to compilers and software libraries. *Software: Practice and Experience*, 49(6):953–970.

Ljungqvist, L. and Sargent, T. (2000). *Recursive macroeconomic theory*. Mit Press.

Ma, X. and Zabaras, N. (2009). An adaptive hierarchical sparse grid collocation algorithm for the solution of stochastic differential equations. *J. Comput. Phys.*, 228(8):3084–3113.

Machina, M. J. and Viscusi, K., editors (2014). *Handbook of the Economics of Risk and Uncertainty*. North Holland Publishing Company, Amsterdam, Netherlands.

Maldonado, W. L. and Svaiter, B. (2007). Hölder continuity of the policy function approximation in the value function approximation. *Journal of Mathematical Economics*, 43(5):629–639.

Morand, O. F. and Reffett, K. L. (2003). Existence and uniqueness of equilibrium in nonoptimal unbounded infinite horizon economies. *Journal of Monetary Economics*, 50(6):1351–1373.

Murараşu, A., Buse, G., Pflüger, D., Weidendorfer, J., and Bode, A. (2012). Fastsg: A fast routines library for sparse grids. *Procedia Computer Science*, 9:354–363.

Murarasu, A., Weidendorfer, J., Buse, G., Butnaru, D., and Pflüeger, D. (2011). Compact data structure and parallel alogrithms for the sparse grid technique. In *16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

Murarasu, A. F. (2013). *Advanced Optimization Techniques for Sparse Grids on Modern Heterogeneous Systems*. PhD thesis, Technische Universität München.

Murarasu, A. F. and Weidendorfe, J. (2012). Building input adaptive parallel applications: A case study of sparse grid interpolation. In *Computational Science and Engineering (CSE), 2012 IEEE 15th International Conference on*, pages 1–8.

Muth, J. F. (1961). Rational expectations and the theory of price movements. *Econometrica*, 29(3):315–335.

Nobile, F., Tempone, R., and Webster, C. G. (2008). A sparse grid stochastic collocation method for partial differential equations with random input data. *SIAM Journal on Numerical Analysis*, 46(5):2309–2345.

NVIDIA (2020). Nvidia a100 tensor core gpu architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf. [Online; accessed 28-November-2022].

OECD (2001). *The Well-Being of Nations: The Role of Human and Social Capital*. OECD Publication Service, Paris, France.

Petersohn, D., Ma, W. W., Lee, D. J. L., Macke, S., Xin, D., Mo, X., Gonzalez, J. E., Hellerstein, J. M., Joseph, A. D., and Parameswaran, A. G. (2020). Towards scalable dataframe systems. *CoRR*, abs/2001.00888.

Pflüger, D. (2010). *Spatially Adaptive Sparse Grids for High-Dimensional Problems*. PhD thesis, TU München, München.

Pflüger, D. (2012). Spatially adaptive refinement. In Garcke, J. and Griebel, M., editors, *Sparse Grids and Applications*, Lecture Notes in Computational Science and Engineering, pages 243–262, Berlin Heidelberg. Springer.

Pflüger, D., Bungartz, H.-J., and Peherstorfer, B. (2014). *Density Estimation with Adaptive Sparse Grids for Large Data Sets*, chapter 50, pages 443–451. SIAM.

Puterman, M. L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, New York City, NY, USA.

Rabitz, H. and Alis, O. F. (1999). General foundations of high-dimensional model representations. *Journal of Mathematical Chemistry*, 25(2–3):197–233.

Reinders, J. (2007). *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition.

Rust, J. (1997). Using randomization to break the curse of dimensionality. *Econometrica*, 65(3):487–516.

Rust, J. P. (1996). Numerical dynamic programming in economics. In Amman, H. M., Kendrick, D. A., and Rust, J., editors, *Handbook of Computational Economics*, volume 1, chapter 14, pages 619–729. Elsevier, 1 edition.

Scheidegger, S. and Mikushin, D. (2018). Interpolation backends for hddm-solver. https://github.com/apc-llc/hddm-solver-postprocessors/tree/2016v1_olgtax. [Online; accessed 7-January-2018].

Scheidegger, S., Mikushin, D., Kubler, F., and Schenk, O. (2018). Rethinking large-scale economic modeling for efficiency: Optimizations for gpu and xeon phi clusters. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 610–619.

Skjellum, A., Gropp, W., and Lusk, E. (1999). *Using MPI*. MIT Press.

Smolyak, S. (1963). Quadrature and interpolation formulas for tensor products of certain classes of functions. *Soviet Math. Dokl.*, 4:240–243.

Stokey, N., Lucas, R., and Prescott, E. (1989a). Recursive methods in economic dynamics. *Cambridge MA*.

Stokey, N. L., Lucas, Jr., R. E., and Prescott, E. C. (1989b). *Recursive Methods in Economic Dynamics*. Harvard University Press, Cambridge, MA.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Waechter, A. and Biegler, L. T. (2006). On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Math. Program.*, 106(1):25–57.

Wes McKinney (2010). Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56–61.

White, D. J. (1993). *Markov Decision Processes*. John Wiley & Sons, New York City, NY, USA.

Winschel, V. and Kraetzig, M. (2010). Solving, estimating, and selecting nonlinear dynamic models without the curse of dimensionality. *Econometrica*, 78(2):803–821.

Zenger, C. (1991). Sparse grids. In Hackbusch, W., editor, *Parallel Algorithms for Partial Differential Equations*, volume 31 of *Notes on Numerical Fluid Mechanics*, pages 241–251. Vieweg.

Dmitry Mikushin

1008 Prilly
Switzerland
+41789259090
dmitry@parallel-computing.pro
mikush.in
GitHub: dmikushin, APC LLC

Academic experience

2019–present **Doctoral Assistant**, *University of Lausanne*, Switzerland

Group of Prof. Simon Scheidegger. R&D for computational finance and advanced data analytics (Machine Learning, Gaussian processes). Development of high-dimensional optimization and classification frameworks as GPU-backed cloud services (C++, Fortran, CUDA, Python).

2015–2019 **Research Associate**, *University of Zurich*, Institut für Banking und Finance, Switzerland

Group of Dr. Simon Scheidegger. Computing global solutions to annually calibrated dynamic stochastic general equilibrium models for policy analysis (overlapping generation models or OLG). Given prototypes from economists within the group, developing production code for large hybrid computing systems equipped with NVIDIA or Intel accelerators (CSCS/Piz Daint, NERSC/Cori). Fine-tuning of value function interpolation kernels on a sparse grid with linear or polynomial basis (AVX, CUDA, Intel Thread Building Blocks). Co-authored a journal paper.

2012–2015 **Doctoral Assistant**, *University of Lugano*, Switzerland

Group of Prof. Olaf Schenk. Assisting education and research activities. Worked on manual optimization and designed tools for automatic optimization of stencil codes in seismic and weather prediction models (CUDA, OpenACC). Main author for a conference paper.

2013 **Visiting Scholar**, *Rutgers University*, New Jersey, US

Group of Prof. Eddy Zheng Zhang. Analyzed the efficiency of atomics on different families of NVIDIA GPUs. Tuned GPU kernels optimizations in KernelGen compiler.

2008–2011 **Junior scientist**, *Supercomputer simulation laboratory for climate modeling, Research Computing Center*, Lomonosov Moscow State University, Russia

Numerical and performance evaluation of various mesoscale and regional models. Experimented with porting key model dynamics blocks on Cell Broadband Engine and GPU architectures.

2006–2007 **Contractor**, *Global Energy Problems Lab*, Moscow Energy Institute

Implemented toolbox for Voronoi tesselation and regression analysis in C#.

Industrial experience

2019–present **Support engineer (part time)**, *Luxoft, a DXC Technology Company*

Emergency & technical support for previously developed backpricing service (financial customer).

2018–2019 **System Analyst (part time)**, *REG.RU*

Overseeing the development of Cloud GPU services for scientific and industrial machine learning. Helping with internal GPU-based spin-off products.

2017–2019 **Lead Technical Expert**, *Luxoft Inc.*

GPGPU for high-performance Monte-Carlo backpricing valuation (financial customer). Complete rewrite of Scala code into C++ & CUDA, skip-ahead optimizations for Sobol QRNG. Design and implementation of efficient GPU kernels for lidar simulation (automotive customer).

2014–present **Owner**, *Applied Parallel Computing LLC (CUDA Education & Research in EMEA)*, `http://parallel-computing.pro/`

2011–2012 **CTO**, *Applied Parallel Computing LLC (CUDA Education & Research in EMEA)*, `http://parallel-computing.pro/`

Managing technological aspects in company's GPGPU training and software development business. Created course list on comprehensive CUDA training program, implemented original presentations and hands-ons, later used in CUDA 4.x Handbook in Russian. Served as trainer on events in Germany, Ireland and Russia. Organizing and reviewing work of 7 company's contracted trainers/developers. Responsible for interaction with customers and partners worldwide.

2009–2011 **DevTech Engineer**, *NVIDIA*, Moscow, Russia

Ported parts of numerical weather prediction models onto GPUs: spectral solver benchmark (Russian Met Office), GPU kernels generator for COSMO model (Deutscher Wetterdienst et al). Supported customers and developers on CUDA programming in HPC applications, provided training sessions. PhysX game physics engine: implemented SPU-interacting radix sort for rigid bodies broad phase algorithm on Cell Broadband Engine processor (Sony PlayStation 3), made first experimental Tegra/ARM ports of PhysX engine, helped with Linux port.

Awards

2013 **PhD fellowship**, *Rutgers University, Department of Computer Science*

2011 **CUDA Certificate 016-2011/29.10.2011**, *NVIDIA*, Moscow, Massively parallel processors, CUDA architecture and programming environment

2008 **PhD fellowship**, *Institute of Numerical Mathematics, Russian Academy of Science*

2008 **T-Platforms PowerXCell 8i Programmers Contest, second award**, *Optimization of mathematical modeling package for hydrodynamics "GeoPhyCell"*

2008 **Best Student Diploma, second award**, *Numerical modeling of mesoscale aerosol transfer due to hydrological inhomogeneity of the boundary layer*

Education

2012–2015 **PhD studies**, *University of Lugano*, Institute of Computational Science, Switzerland

2008–2011 **PhD (ABD – passed qualification and comprehensive examinations)**, *Institute of Numerical Mathematics, Russian Academy of Science*, Moscow

2003–2008 **Specialist (5-year B.S. + M.S program)**, *Faculty of Computational Mathematics and Cybernetics*, Lomonosov Moscow State University, *Computational Technologies and Modeling*

Master thesis

title *Numerical modeling of mesoscale aerosol transfer due to hydrological inhomogeneity of the boundary layer*

supervisors Dr. Vasily N. Lykossov, Dr. Victor M. Stepanenko

Implemented and analyzed Smolarkiewicz transport scheme, the positive-definite method of Lax–Wendroff class. Resulting source code was incorporated into regional non-hydrostatic model of atmosphere and boundary layer (NH3D) and used to trace passive aerosol. Experiments with real terrains showed significant numerical accuracy improvement both in mass conservation and approximation order over leapfrog and first order transport schemes.

Publications

[1] Johannes Brumm et al. "Scalable high-dimensional dynamic stochastic economic modeling". In: *Journal of Computational Science* 11 (2015), pp. 12–25. ISSN: 1877-7503. DOI: `https://doi.org/10.1016/j.jocs.2015.07.004`. URL: `https://www.sciencedirect.com/science/article/pii/S1877750315300053`.

[2] Nianchuan Jian et al. "A GPU-based phase tracking method for planetary radio science applications". In: *Measurement Science and Technology* 31.4 (Jan. 2020), p. 045902. DOI: `10.1088/1361-6501/ab58e5`. URL: `https://dx.doi.org/10.1088/1361-6501/ab58e5`.

[3] Andrey Kuzmin, Dmitry Mikushin, and Victor Lempitsky. "End-to-End learning of cost-volume aggregation for real-time dense stereo". In: *2017 IEEE 27th International Workshop on Machine Learning for Signal Processing (MLSP)*. 2017, pp. 1–6. DOI: 10.1109/MLSP.2017.8168183.

[4] Dmitry Mikushin and Victor Stepanenko. "The Implementation of Regional Atmospheric Model Numerical Algorithms for CBEA-Based Clusters". In: *Parallel Processing and Applied Mathematics*. Ed. by Roman Wyrzykowski et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 525–534.

[5] Dmitry Mikushin et al. "KernelGen – The Design and Implementation of a Next Generation Compiler Platform for Accelerating Numerical Models on GPUs". In: *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. 2014, pp. 1011–1020. DOI: 10.1109/IPDPSW.2014.115.

[6] Simon Scheidegger et al. "Rethinking large-scale Economic Modeling for Efficiency: Optimizations for GPU and Xeon Phi Clusters". In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2018, pp. 610–619. DOI: 10.1109/IPDPS.2018.00070.

Selected talks

Dmitry Mikushin, Nikolay Likhogrud, Sergey Kovylov. KernelGen: A Prototype of Auto-parallelizing Fortran/C compiler for NVIDIA GPUs, *HPC Advisory Council 2013*, available online

Active skills

CS/Research	**Explore new environments/software and teach others to use them, design & perform experiments to analyse hardware/software properties and generalize findings into practically useful methods/tools**
HPC/Engineer	**Fluency in full development & support cycle of HPC applications for Linux clusters: programming, parallelization for different architectures, debugging, profiling**
Bitcoin mining	**Basic understanding of SHA256 and Zero-knowledge proof. Optimization of Equihash miner for NVIDIA GPUs. Available on GitHub**
Compilers Dev	**Basic knowledge of compilers internal structure, contributions to LLVM. Designed and developed KernelGen – a prototype of auto-parallelizing Fortran/C compiler for NVIDIA GPUs, targeting numerical modelling code**
GPU low-level	**Experience with NVIDIA GPU binary format and Fermi/Kepler assembler, profiling & optimization**
Numericals	**Practical experience with linear solvers, PDEs and related cache-aware optimizations**
NWP	**Engineer-level experience with numerical weather prediction models: WRF-ARW, COSMO**

Teaching

2014 **Parallel & Distributed Computing**, *University of Lugano*
The fundamentals of concurrent execution. Threading in Java. The basics of OpenMP and MPI.

2013–2014 **Parallel & Distributed Computing Lab**, *University of Lugano*
Configuration & deployment of scientific codes on modern GPU-enabled HPC facilities, by example of SWE tsunami simulation model and CSCS "Tödi" cluster

Languages

English	fluent technical
Russian	native