High-performance Computing Implementations of Agent-based Economic Models for Realizing 1:1 Scale Simulations of Large Economies

Amit Gill, Madegedara Lalith, Sebastian Poledna, Muneo Hori, Kohei Fujita, and Tsuyoshi Ichimura

Abstract—We present a scalable high-performance computing implementation of an agent-based economic model using distributed + shared-memory hybrid parallelization paradigms, capable of simulating 1:1 scale models of large economies like the eurozone. Agent-based economic models consist of millions of agents interacting over several graphs, which are either centralized or scale-free in nature. While most of the interactions are bi-directional, the interaction graphs are dense and random and keep evolving as the simulation progresses. These characteristics cause a very large and unknown number of random communications among MPI processes, posing challenges to developing scalable parallel extensions. Further, random access to large volume of data makes the algorithms highly memory-bound, severely degrading computational performance. Adopting various strategies inspired by the real-world functioning of economies, we reduce the large unknown number of communications to a known handful number. Memory-intensive algorithms are improved to make these cache-efficient, and advanced MPI functions are used to minimize communication overhead, thereby attaining higher performance and scalability. Further, an MPI + OpenMP hybrid model is developed to best utilize modern many-core computing nodes with low per-core memory capacity. It is demonstrated that our implementation can simulate a full fledged economic model with 331 million agents within 108 seconds using 128 CPU cores attaining 70% strong scalability.

Index Terms—Agent-based Economic Models, High-performance Computing, One-to-one scale simulations, Large economies, Scale-Free Graphs, Message Passing interface, OpenMP

1 Introduction

Agent-based modeling has gained significant popularity in the computational economics community as an alternative to conventional approaches like Dynamic Stochastic General Equilibrium (DGSE) modeling. The economy of a country or an economic bloc is characterized by the interactions among individual entities, each with its own cognitive abilities, and these interactions give rise to complex, hard-to-predict macro-economic outcomes like recessions. In agent-based economic models (ABEMs) or agent-based computational economics (ACE), agents approximately mimic the decisions and actions of their real-life economic counterparts, reproducing the observed complex macro-economic behaviors as emergent phenomena. Agents' behavior may be governed by simple rules obtained from behavioral economics or machine-learning models trained with past data. The majority of the studies focus on the development of agents' rules to replicate macro-economic and micro-economic empirical stylized facts, such as the time-series properties of the output fluctuations and growth, as well as the cross-sectional distributional characteristics of firms.

There has been a significant interest in the agent-based modeling community to develop HPC enhanced models to simulate 1:1 scale models of large economies such as the European Economy ([1], [2], [3], [4]). To that end, the European Union, for example, has funded three multi-million Euro projects ([5], [6], [7]) to develop ABEMs over the last decade. However, to the best of the authors' knowledge, the ambition of performing 1:1 scale simulations of large economies has not been realized yet. Some implementations aiming for large-scale agent-based simulations of economies are [1] and [8]. Poledna *et al.*'s advanced ABEM [8] utilizes big data to set the agents' parameters. Their OpenMP-based shared-memory parallel implementation is capable of 1:1 scale simulations of a small open economy with 10 million agents. However, no metrics of computational performance have been reported. Though Deissenberg *et al.* [1] present an ambitious plan to develop an HPC-implementation of their ABEM called EURACE to conduct large-scale simulations of the European Economy, no publication on their realized implementations is available.

Most authors, e.g., [1], [6], draft their HPC plans relying on general-purpose agent-based programming frameworks like Pandora [9], Repast HPC [10], EcoLab [11], FLAME [12], etc. In the authors' point of view, these general-purpose

- *Amit Gill is with the Department of Civil Engineering, The University of Tokyo, 7-3-1 Hongo, Bunkyo City, Tokyo, 113-8654, Japan. E-mail: gill@eri.u-tokyo.ac.jp*
- *Madegedara Lalith, Kohei Fujita, and Tusyoshi Ichimura are with the Department of Civil Engineering and the Earthquake Research Institute, The University of Tokyo, 7-3-1 Hongo, Bunkyo City, Tokyo, 113-8654, Japan. E-mail: lalith, fujita, ichimura@eri.u-tokyo.ac.jp*
- *Sebastian Poledna is with the International Institute for Applied Systems Analysis, A-2361, Laxenburg, Austria and the Institute for Advanced Studies (IHS), Josefstädter Straße 39, 1080 Vienna. E-mail: poledna@iiasa.ac.at*
- *Muneo Hori is with the Research Institute for Value-Added-Information Generation, Japan Agency for Marine-Earth Science and Technology, 3173-25, Showa-machi, Kanazawa-ku, Yokohama-city, Kanagawa, 236-0001, Japan. E-mail: horimune@jamstec.go.jp*

Manuscript received April 19, 2005; revised August 26, 2015.

frameworks hit strong limitations in both the number of agents and computational performance due to complex interactions among the economic agents. Pandora, Repast HPC, and EcoLab facilitate interactions among agents located in two independent CPUs by copying the interacting agents into both the CPUs. These copies are widely known as ghost or halo copies. In general, the larger the number of ghost copies, the lower the computational efficiency. These platforms are therefore best suited for models in which agents interact within a limited neighborhood. Using ghost copies is not an effective strategy when each interaction involves certain constraints such as the agents having a limited consumption budget. It is fundamental in ABEMs that an arbitrary pair of agents must be able to interact, and most interactions involve certain constraints. Hence, these platforms can impose serious performance limitations on ABEMs unless the model is compromised. FLAME manages the interactions among agents of two processes by posting/receiving messages to/from queues maintained at each process. The size of these message queues can grow exponentially, hampering the computational efficiency of ABEMs, since the interactions are random and the number of interactions is several orders of magnitude larger than the number of agents, etc.

This paper presents the details of an efficient, from scratch developed HPC implementation capable of simulating 1:1 scale models of large economies, finally realizing the ABEM communities' long-sought ambition. Our HPC implementation is efficient enough to simulate several hundred millions of interacting agents within minutes using high-end workstations or small computer clusters. The contents of this paper can be considered an extension of our preliminary distributed-memory parallel implementation [13] to a several-fold more efficient and scalable MPI+OpenMP hybrid parallel implementation targeting the modern many-core computing nodes with low per-core memory capacity.

Agents in ABEMs interact over several centralized graphs like financial networks, and several scale-free graphs like goods market, job market, etc. Most of these interactions are dynamic, random, and bi-directional in nature. These complex interactions among agents pose serious challenges to developing a scalable parallel implementation. To attain high performance in distributed-memory parallel implementation, the agents must be partitioned (i.e., distributed) among MPI-ranks ensuring a balanced load distribution while minimizing the number of communications among the MPI-ranks. As it is not possible to consider the interactions over all the graphs and partition the agents to minimize communications, we partition the agents based on a representative graph of the labor market and adopt various strategies to ensure that the agents can freely interact over the remaining graphs. Most of our strategies are inspired by real-life economic interactions. For example, banks have many branches across a nation, and the customers visit local branches instead of the head office. We use the same strategy for the parallelization of credit network and introduce local branches of banks in each CPU. Similarly, we adopt various other strategies by mimicking a real-life consumption market, labor market, etc., to reduce the number of communications. To efficiently communicate the data, we use suitably designed MPI-datatypes and utilize advanced MPI functions. We use non-blocking communications to utilize communication time by overlapping communication with computation. To improve the computational efficiency, serial algorithms are improved to make them cache-friendly and less memory-intensive. Details of these improvements and their performance advantages are presented in this paper. We also present an MPI + OpenMP hybrid implementation targeting the optimal use of many core computing nodes with low per-core memory capacity. The performance of the implementation has been evaluated by simulating a small economy with 10 million agents and a large economy with 331 million agents. Some results from the early stage of the development, which is capable of simulating small economies up to 10 million agents, have been presented in [13]. The current implementation possesses a high strong scalability of up to 67%, which is significantly high for this class of problems; strong scalability is a standard measure of a parallel program's capacity to effectively utilize a larger number of CPUs. Moreover, improved serial algorithms significantly boost the computational performance, enabling large economies to be simulated using a relatively small amount of computational resources, even on workstations, and that, too, within minutes.

The rest of the paper is organized as follows. In section 2, we present the ABEMs from the perspective of high-performance computing, and the challenges involved in implementing a scalable solution are discussed. Section 3 describes our implementation in detail, explaining the various strategies adopted to address the challenges presented in section 2. In section 4, we analyze the computational performance of the strategies presented in 3. Section 5 presents the MPI + OpenMP hybrid implementation, and section 6 concludes.

2 ABEMs from the perspective of HPC

A typical ABEM simulates an economy as both a time- and event-driven dynamic system of interacting heterogeneous and autonomous agents. An agent can be active in the markets of several industries to satisfy its needs. An individual agent's behavior is defined by rules based on behavioral economics, and its actions are influenced by its current state and past experiences. Every interaction between two agents in a market results in relevant decision-making and exchange of some data between the two. The markets evolve over time depending on the satisfaction or dissatisfaction of the agents with the counterparts they select.

Fig. 1 presents a schematic diagram of a typical ABEM. The model consists of various agents: firms as a part of industries, foreign firms, households consisting of investors, workers, and inactive households, banks, central government, and central bank. Most of the stylized ABEMs do not include the geographical distribution of agents. However, we assign geographical location to each agent, as this has a significant influence on the assessment of the economic impacts of natural disasters, etc. In our implementation, agents operate from their location and are assumed to be connected by various networks.

High performance computing involves many CPU cores, distributed over multiple computing nodes, working on disjoint subsets of the data in parallel. In distributed parallel

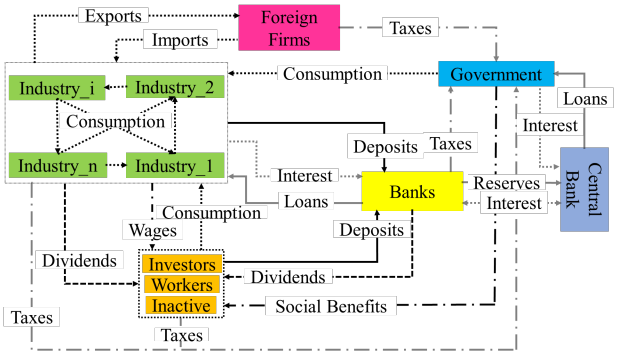
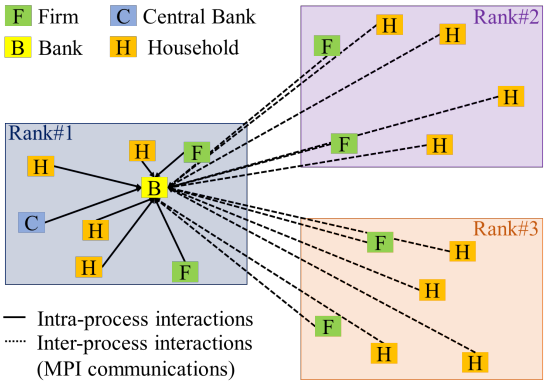
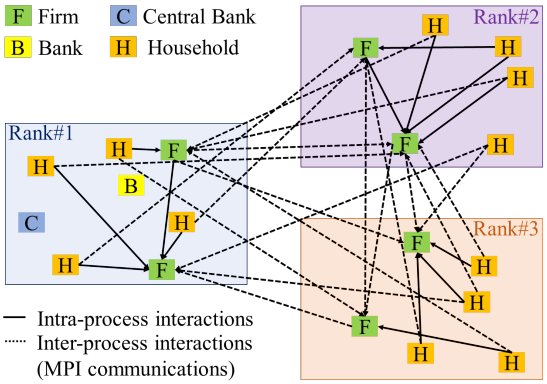


Fig. 1: Schematic diagram of an ABEM



(a) Bank-customers interactions on centralized graphs



(b) Buyer-seller interactions on scale-free graphs

Fig. 2: Schematic diagrams of typical centralized and scale-free interaction graphs in an ABEM

implementations, each CPU core is assigned its own private memory space, and data are exchanged among CPU cores using the *Message Passing Interface (MPI)*. Often, each CPU core is referred to as MPI-process or MPI-rank; for the sake of brevity, we use the term *rank*. Distributing equal workload among MPI-ranks so that all can finish the assigned tasks at almost the same time with the least idle time is the key to attaining high performance. Since the speed of communication networks is much slower than that of CPUs, minimizing the number and volume of communications among the ranks is equally important in terms of attaining high performance. The nature, frequency, and volume of communications depend on the type of interactions among the agents located in different ranks. Hence, understanding the nature of different interactions in ABEMs is important in implementing a high-performing parallel code. The rest of this section describes details of the different types of interactions among the agents and the difficulties that they pose.

2.1 Uni- and bi-directional interactions

The interactions among agents can be divided into two categories: uni-directional and bi-directional. Interactions like paying tax to the government and depositing money in the bank are uni-directional, whereas interactions involving taking a bank loan, a government subsidy payment, buying goods in the market, applying for jobs, etc., are bi-directional. Uni-directional interactions involve some data transfer from the source agents to the target agents and are easy to parallelize, as all such interactions emanating from a rank can be clubbed together into one message, which requires only one communication. On the other hand, bi-directional interactions consist of three stages: data transfer from the source agent to the target agent, some decision-making by the target agent based on the data received, and a reply to the source agent. Thus, these interactions require two or more communications and cause a huge load imbalance, as the target agent processes the requests serially.

2.2 Interactions over centralized graphs

Agents' interactions with the banks and the government take place over centralized graphs. Some of the interactions, like paying tax to the government, are uni-directional, while others, like taking loans from the bank, are bi-directional. Fig. 2a shows a schematic diagram of bank-customers interactions.

2.2.1 *Interactions of firms, households, and bank with government over centralized graphs*

All local firms, foreign firms, workers, investors, inactive households, and banks pay various kinds of taxes to the central government. The government pays social benefits and subsidies to the households, and subsidies to some of the industrial sectors. Tax paying is uni-directional and can easily be parallelized by making each rank collect tax from its agents and then sending the collected tax to the rank containing the government agent. On the other hand, social benefit payments by the government is bi-directional as the amount of social benefits to be paid to a household depends on the latter's economic status. A simple solution to reduce the number of communications involved is to make the government agent collect the economic status of each household using `MPI_Gatherv()`[1], then make it calculate the amount of social benefits payable to each household, and finally deliver the social benefits to each household using `MPI_Scatterv()`. However, this introduces a significant load imbalance, as all the waiting ranks will idle until the rank containing the government agent calculates the social benefits of millions of households.

1. All the typewriter format terms starting with `MPI_` are functions or data types of MPI.

2.2.2 Interaction of firms and households with the bank

All firms and households, depending on their financial situation, interact with the bank to deposit their surplus income, withdraw their deposited money, or take a loan. Depositing is uni-directional, whereas withdrawing and taking a loan are bi-directional. Similar to the social benefit transfers by the government, withdrawing and taking a loan involve three steps: gathering the loan applications using `MPI_Gatherv()`, processing the applications one by one, and scattering the reply to the applicants using `MPI_Scatter()`. This imposes a high computational load on the rank containing the bank. It is also important to note here that not all the agents may apply for loans at the same point in time, and it may be that not all agents will apply for loans in each period. Therefore, the interaction graph between loan applicants and the bank is dynamic in nature and keeps evolving with time, which makes its parallel implementation more challenging.

2.3 Interactions over scale-free graphs

Interactions in the goods market and also in the labor market take place over scale-free graphs. Almost all these interactions are bi-directional, random, and dynamic in nature. A typical buyer-seller interaction graph is presented in Fig. 2b. As the interactions are driven by continuously varying states of individual agents, the interaction graphs randomly grow within each iteration. The bi-directional interactions over random graphs make it a challenging task to implement scalable parallel extensions.

2.3.1 Buyer-seller interactions in the goods market

Each period, all buyers (i.e., firms, foreign firms, households, and government agencies) visit a random set of sellers (i.e., local firms and foreign firms) from each industry to buy capital goods and consumption goods. Most ABEMs assume that all the sellers of the economy are accessible to all the buyers. The probability of a seller being chosen by a buyer depends on the seller's size (i.e., the quantity of goods produced) and the price of its products; sellers with larger quantities and lower prices have a higher probability of being visited by the buyers. To give a fair chance to all buyers, the order in which they go shopping is randomized in each period. Each visit of a buyer to a seller located in another rank needs two MPI communications between their home ranks because of the bi-directional nature of the interaction. The number of sellers visited by a buyer is unpredictable, as it depends on its budget and the quantity of goods available for sale with the seller at the time of the buyer's visit. Therefore, the goods market produces a very large number of random, one-to-one, and bi-directional communications between ranks. Developing a scalable algorithm for such a scenario is challenging.

A simple solution is to make each seller agent receive purchase requests from its buyers to a queue and process the requests sequentially. However, sequential processing of requests means that the buyers will have to wait until the sellers they visited process all the requests and send a reply message and that they will be able to decide whether they have to visit another seller only after the reply. There will be a large load imbalance among MPI ranks, as the ranks containing sellers with higher selling probability receive a larger number of requests. This solution will also produce an event-driven algorithm consisting of many buying events, decreasing the opportunities for parallel processing and forcing even satisfied buyers to participate in buying events until the last buyer satisfies its needs. For a number of such reasons, the overall performance of this simple solution can be even worse than the serial code.

2.3.2 Employee-employer interactions in the labor market

In each period, firms hire or fire workers according to their labor demand, and unemployed workers search for jobs. As the system is dynamically evolving, it is not possible to partition the agents such that each rank contains a sufficient number of workers to satisfy the labor demand of the firms assigned to it. Some ranks can thus have excess labor while some others are short of labor. At the start of each period, the unemployed workers situated in ranks that have more workers than jobs have to search for jobs in the labor-deficient ranks. A job application by a worker to a firm in another rank is bi-directional and involves two point-to-point communications: the application by the worker and the firm's reply. If the worker can secure a job with a firm located in a remote rank, the firm will send messages to the worker either to deliver his wages to him or to inform him that he has been fired. Like the goods market, the number of firms visited by a worker in search of a job is unpredictable. Obviously, implementing a scalable solution for such a scenario is also challenging.

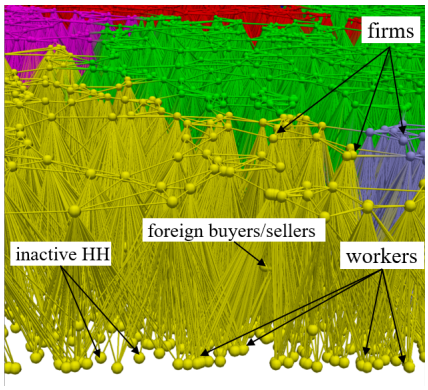
3 Scalable High-performance implementation

This section presents the details of the strategies used to develop an HPC-enhanced ABEM, particularly the overall design of the implementation, the partitioning of agents to assign balanced workload to MPI ranks, various strategies to address the challenges discussed in the previous section, and cache-friendly algorithms and data structures for enhancing serial performance.

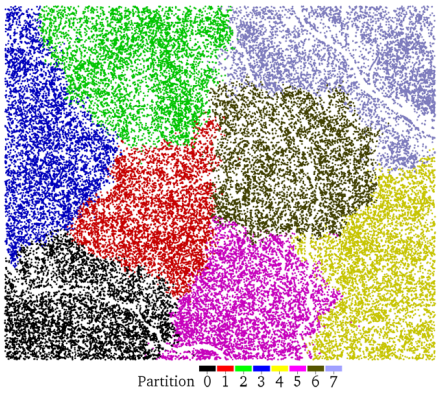
3.1 Partitioning

Balanced distribution of workload among ranks while minimizing the number of communications and exploiting opportunities to hide the communications are widely used strategies in developing a scalable parallel code. As explained in the previous section, the agents interact over different sets of centralized or scale-free graphs. The two scale-free graphs are the biggest barriers to developing a scalable solution, as they are random and dynamic. As the topological connectivity is significantly different in the two scale-free graphs, computational work is distributed among ranks based only on a representative, scale-free graph of employer-employee interactions, and other interaction graphs are made available at a low communication cost using various strategies.

We construct the representative employer-employee interaction graph by joining each firm f_i with n_i nearest workers, where n_i is the initial labor demand of f_i. From the remaining agents, inactive households and foreign firms



(a) Graph connecting agents



(b) Partitions

Fig. 3: A graph connecting firms and other agents, and partitions generated using METIS [14].

are included in the graph by connecting them to the nearest one or two firms, whereas investors, banks, government, and central bank are not included in the graph but assigned to the partitions later. To ensure that the graph sufficiently reflects the employer-employee interactions, low weights are assigned to the links connecting inactive households and foreign firms to the firms. Further, to ensure that the graph is connected, the firms are also linked with neighboring firms using triangulation with low link weights. The nodes of the graph are assigned a nodal weight according to the computational workload of their associated agent. The graph is partitioned using the `METIS_PartGraphKway()` function of the METIS library [14], such that each partition has a nearly equal sum of nodal weights and the partition boundaries cut the minimum number of edges (i.e., ${}^{w}N^{r} \approx \sum_{1}^{{}^{f}N^{r}} n_i$, where ${}^{w}N^{r}$ and ${}^{f}N^{r}$ are the numbers of workers and firms finally assigned to the rank r). Fig. 3 shows part of a small graph and the eight partitions generated. After partitioning, firms' investors are assigned to the partitions that include the respective firms. Bank, government, and central bank are assigned to the master rank (e.g., rank 0) so that they can interact among themselves without any communications.

3.2 Scalable solutions for the centralized graphs

Decentralizing the functionalities of the agent at the central node of centralized graphs by adding one local-central-node to each partition and connecting only the local-central-node to the global-central-node (see Fig. 4) is an effective way of drastically reducing the high communication overhead discussed in section 2.2.

Once decentralized, not only do the interactions involve load and store to the individual rank's private memory, but they can also take place in parallel. To evaluate the gains of the decentralization, consider N number of agents equally distributed over P number of MPI-ranks. If the N agents have bi-directional interactions with the global-central-node located in rank 0, and if each message involves d number of `double` precision values, the total size of the messages is $2 \times (N \times \frac{P-1}{P}) \times (8 \times d)$ bytes. If the central node takes δt time to process each request and if we ignore the time for message passing, the amount of wasted CPU time due to resulting load imbalance is $\delta t \times N \times \frac{P-1}{P}$. On the other hand, when decentralized, the total message size is $(P-1) \times (8 \times d)$ bytes and the amount of wasted CPU time is 0. N being in the range of 10 to 100 million, and P being in the range of 10 to several hundreds, the time saved by the decentralized system is significant. Hence, decentralization improves the scalability while reducing the total execution time proportional to P.

3.2.1 Interactions with the government

The simple solution discussed in section 2.2.1 has poor scalability because of the high load imbalance and the resulting serialization. As discussed above, a scalable solution is to decentralize the government by introducing a new agent, *LocalGovt*, in each rank which will serve as the government agent. The *LocalGovt* collects taxes from all other agents, calculates and pays social benefits to households, and buys a portion of goods required by the government. At the end of the financial period, *LocalGovt* communicates the amount of taxes collected, social benefits paid, and the goods bought to its parent agent, the government, which is present in the master rank using just one communication. This solution evenly distributes the computational workload, eliminates all bi-directional communications, and drastically reduces the overall number of communications.

3.2.2 Interactions with the bank

Just like the interactions with the government, the problem can be solved by introducing a new agent, *LocalBank*, in each rank that is able to locally process all the deposit and withdrawal requests from the customers. This not only improves scalability and reduces run-time, but also reduces the memory requirements of the master rank, as the master rank doesn't need to store the account details of the millions of bank customers.

$$\frac{E_k(t-1)}{\sum_{i=1}^{I}\left(L_i(t-1)+\Delta L_i(t)\right)} \geq \zeta \tag{1}$$

$$\frac{E_k(t-1)}{\sum_{i=1}^{I} L_i(t-1)} \geq \zeta \tag{2}$$

Decentralization of some interactions may, however, require somewhat compromising modifications to be made to the agent-based model. One such example is interactions with customers requesting loans. In typical agent-based models, a bank extends loans to customers based on

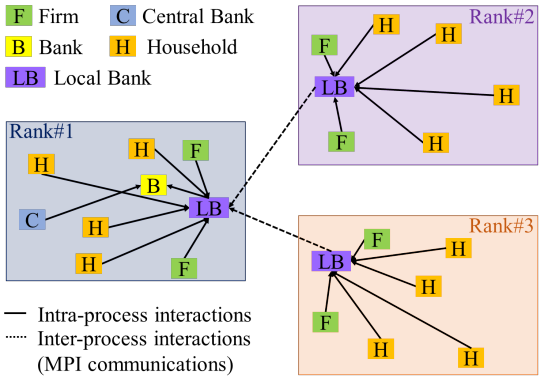


Fig. 4: Interactions of *LocalBanks* with rest of the agents

conditions like those in Equation 1, where $E_k(t-1)$ is the bank's equity at the start of t^{th} period, $1/\zeta$ is the maximum allowable leverage for the bank, $\Delta L_i(t)$ and $L_i(t-1)$ are the requested amount of loan and the outstanding loan of the i^{th} customer, and I is the total number of customers. The denominator makes the processing of loan requests sequential, as *LocalBanks* have to consult the bank's headquarters in rank 0, which requires a large number of communications and makes this interaction serial. Since $\Sigma_{i=1}^{I}\Delta L_i(t) << L(t-1)$, a solution with a negligible compromise is to use the condition 2, instead of 1. Numerical tests confirmed that this modification has imperceptible effects. At the end of the financial period t, the bank's headquarters, located in rank 0, collects the financial reports (i.e., total deposits, loans, interests received, etc.) from all *LocalBanks* and informs each local branch of the values of $\sum_{i=1}^{I} L_i(t)$ and $E_k(t)$ so that local branches can independently extend loans to customers in the respective ranks in the next period $t+1$.

3.3 Scalable solution for scale-free graphs

As discussed in section 2.3, implementing a scalable solution for scale-free graphs is challenging because of their random, dense, and dynamic nature. However, it is the highly stylized nature of the ABEMs which makes it difficult to parallelize the goods market and the labor market. In almost all ABEMs, the customers buy goods directly from manufacturing firms, whereas in the real world, customers buy goods from intermediate entities like supermarkets. Our scalable solution for scale-free graphs is to closely mimic the real-world functioning of the goods market and labor market, by introducing new intermediate agents *SalesOutlet* and *RecruitmentAgency*.

3.3.1 Interactions on goods market

To mimic the real world, a firm $f_i^{\bar{r},s}$ from an industrial sector s located in a given rank $\bar{r}$ is set to sell its products to the customers in any rank r through $f_i^{\bar{r},s}$'s *SalesOutlet* $o_i^{r,s}$, where $\bar{r}, r \in \{0,\ldots,P-1\}$, and $s \in \{0,\ldots,S-1\}$; P and S are the total number of ranks and the total number of industries respectively. This enables the consumers in rank r to directly buy goods of firm $f_i^{\bar{r},s}$ from $o_i^{r,s}$ without requiring any MPI communications, which addresses all the problems related to the goods market discussed in section 2.3.1.

In each period t, there are two interactions between a firm $f_i^{\bar{r},s}$ and its *SalesOutlet* $o_i^{r,s}$: supply of products to $o_i^{r,s}$ before the start of the goods market and collection of sales records from $o_i^{r,s}$ at the end of the goods market. Firms distribute products among their *SalesOutlet*s proportionally to the demand for their products in each rank. Firm $f_i^{\bar{r},s}$ supplies $y_i^{r,s}(t)$ of its total production $Y_i^{\bar{r},s}(t)$ to its *SalesOutlet* $o_i^{r,s}$ as per Equation 3, where $D^{r,s}(t)$ is the demand for goods of sector s in rank r.

$$y_i^{r,s}(t) = Y_i^{\bar{r},s}(t)\frac{D^{r,s}(t)}{\sum_{r=0}^{P-1} D^{r,s}(t)} \tag{3}$$

In our implementation, each rank r estimates its demand $D^{r,s}(t)$ by summing the budget allocated by rank r's buyers for the goods of sector s and finds the global demand (i.e., $\sum_{r=0}^{P-1} D^{r,s}(t)$) for sector s using `MPI_Allreduce()`. The firm $f_i^{\bar{r},s}$ communicates its total production $Y_i^{\bar{r},s}(t)$ to its *SalesOutlet*s located in other ranks using `MPI_Scatter()`. After receiving $Y_i^{\bar{r},s}(t)$, each *SalesOutlet* $o_i^{r,s}$ computes its portion of products as per Equation 3 and sells to the local buyers in rank r. The reporting of sales records of each sector s is done in two steps; first, the master rank collects the sales record of all $o_i^{r,s}$ using `MPI_Reduce()` and then it scatters the reduced results to the ranks where there are owner firms $f_i^{\bar{r},s}$ using `MPI_Scatter()`.

Although this simple solution eliminates a large number of random communications, careful planning is required to communicate a large volume of data between *SalesOutlets* and their owners. Fig. 5 shows the desired layout of firms and their *SalesOutlets*. Each small box indicates a number of firms of a given industry, located in a rank. The boxes on the left indicate the firms in each rank and the boxes on the right indicate their corresponding *SalesOutlets*. Using a common MPI collective communication is not the most efficient solution, as it involves at least $2S$ number of messages: S calls to `MPI_Allgatherv()` to distribute products to sell, and another S calls to `MPI_Allscatterv()` to collect the sales record. This will introduce a significant overhead when it comes to simulating large economies with many industrial sectors. The communication overhead can be reduced by combining these S independent communications into a single communication using `MPI_Alltoallw()`, which generalizes the collective communications by allowing counts, displacements, and MPI data types to be specified separately. By appropriately defining the MPI data types and displacements at the sending and receiving ends, a single call to `MPI_Alltoallw()` is sufficient to communicate the data of all the firms to their *SalesOutlets* or of all *SalesOutlets* to their owners.

3.3.2 Interactions on the labor market

The partitions, generated based on a representative graph of the labor market (see section 3.1), only ensure that the labor demand in each rank will be approximately satisfied, i.e., ${}^{w}N^{r} \approx \sum_{1}^{{}^{f}N^{r}} n_i$, where ${}^{w}N^{r}$ and ${}^{f}N^{r}$ are the numbers of workers and firms assigned to the rank r, and n_i is the labor demand of the i^{th} firm. The presence of ranks with ${}^{w}N^{r} < \sum_{1}^{{}^{f}N^{r}} n_i$ causes a significant number of inter-process interactions among workers and firms located in different ranks, as discussed in section 2.3.2. Further, since

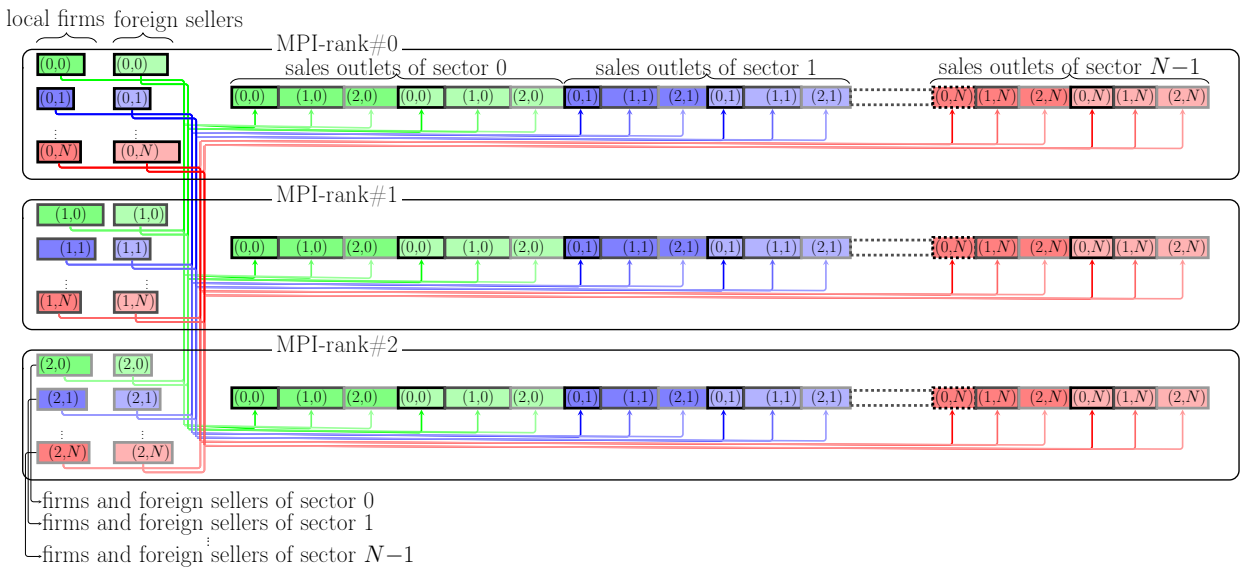


Fig. 5: Layout of firms and their *SalesOutlets* in different MPI-ranks. Each colored box represents a set of local or foreign firms assigned to each rank.

the labor demand of firms is time-dependent, the imbalance may worsen, and the number of interactions may increase. To get rid of these large inter-process interactions, an intermediate agent, *RecruitmentAgency* R^r, is introduced into each rank r which acts as a labor supplier for the firms and job provider for the job seekers. The firms and workers of two ranks r and $\bar{r}$ interact indirectly through R^r and $R^{\bar{r}}$. The actions of *RecruitmentAgency* R^r are summarized below:

1) Firing: In each rank r,
   a) Each firm $f_i^{r,s}$ reports vacancies v_i and fired labors l_i^{fired} to R^r
   b) R^r fires local workers and sends the list of fired non-local workers to $R^{\bar{r}}$, where $r \neq \bar{r}$
2) Local hiring: In each rank r,
   a) Unemployed workers ${}^u w^r$ register with R^r
   b) R^r assigns available jobs $V^r (= \sum v_i)$ to ${}^u w^r$. The imbalance $E^r (= V^r - \sum {}^w w^r)$ indicates the number of extra jobs ($E^r > 0$) or extra workers ($E^r < 0$)
3) Master rank's *RecruitmentAgency*, R^0, collects E^r of each R^r, and assigns labor from $R^{\bar{r}}$ to R^r in a greedy manner so that vacancies of R^r are filled.
4) R^0 sends labor assignments to other R^rs, based on which they exchange the labor and make workers employed at a firm in a remote rank.
5) After the receipt of wages from firms $f^{r,s}$, R^r pays the wages to local workers, and sends wages of non-local workers, if any, to $R^{\bar{r}}$

The introduction of *RecruitmentAgency* agents has eliminated random interactions in the labor market; moreover, their functioning is not affected by the dynamic nature of the graph. All these factors make this a scalable solution.

3.4 Communication hiding

Overlapping the communications and computations, or communication hiding, is a standard technique for reducing the overhead of communications and thereby improving the scalability. Even though blocking MPI functions are mentioned in the text, non-blocking MPI functions (e.g., `MPI_Iscatter()` in place of `MPI_Scatter()`) are used in the actual implementation. We overlap almost all the communications by posting a non-blocking send and receive of a message as soon as the data to be sent become available and finalize the reception of the message just before the start of the event that uses that data.

3.5 Algorithmic improvements

Apart from the above strategies to overcome the difficulties posed by different agents' interactions, the algorithms of data-intensive and extensively called functions are improved to make them cache-friendly. Frequent random read and/or write access to large data arrays, which are larger than the available cache memory of a rank, can add a significant run-time overhead due to the long delays in accessing the main memory or RAM. The goods market, or `buy()` function (Algorithm 1), is the most data-intensive function in an ABEM; hundreds of millions of buyers randomly interact with several millions of *SalesOutlets*. We observed that `buy()` consumes 90% of the total run-time.

The rest of this section provides some details of the algorithmic and data structural changes to reduce the run-time of `buy()`. The effectiveness of these improvements is demonstrated in the next section.

3.5.1 Updating sellers' distribution

As mentioned in section 2.3.1, the chance $p_i^{r,s}$ of a *SalesOutlet* $o_i^{r,s}$ being visited by buyers of rank r depends on the size and the price set by $f_i^{\bar{r},s}$, which is $o_i^{r,s}$'s parent firm located in the rank $\bar{r}$. At the start of each period, t, the $p_i^{r,s}$ for each *SalesOutlet* in each industry s is calculated, and corresponding cumulative distributions $\mathbf{P}^{r,s}$, where $\mathbf{P}^{r,s} = \{P_i^{r,s} | P_i^{r,s} = \sum_{j=1}^{i} p_j^{r,s}\}$, are generated. To choose a seller in industrial sector s, a buyer draws a uniform random

Algorithm 1: Pseudocode for `buy()`

$\mathbf{o}^{r,s} \leftarrow$ vector of active *SalesOutlets* of the sector s in the rank r;
$\mathbf{U_b} \leftarrow$ vector of unsatisfied buyers of the sector s in the rank r;
$N_a \leftarrow \mathbf{o}^{r,s}.size()$; `// number of active-sellers`
$p_i^{r,s} \leftarrow$ probability of i^{th} *SalesOutlet* of the vector S;
Generate a cumulative probability distribution vector $\mathbf{P}^{r,s} = \{P_i^{r,s} | P_i^{r,s} = \sum_{j=1}^{i} p_j^{r,s}\}$;
while $0 < \mathbf{U_b}.size()$ *and* $0 < N_a$ **do**
  $\mathbf{U_b}^{'} \leftarrow$ an empty vector to store remaining unsatisfied buyers
  for $i = 0; i < \mathbf{U_b}.size(); i+ = 1$ **do**
    Draw a uniform random number x such that $0 < x \leq P_{N_s}^{r,s}$ for the buyer, $\mathbf{U_b}[i]$;
    Find the index j of the vector $\mathbf{P}^{r,s}$ such that $P_{j-1}^{r,s} < x \leq P_j^{r,s}$;
    $\mathbf{U_b}[\mathbf{i}]$ visits the seller $\mathbf{o}^{r,s}[j]$ i.e., $o_j^{r,s}$ to buy;
    if $\mathbf{U_b}[i]$ *can not buy everything it needs* **then**
      $\mathbf{U_b}^{'}.push\_back(\mathbf{U_b}[\mathbf{i}])$;
    if $o_j^{r,s}$ *is sold-out* **then**
      N_a = `Update_cumulative_probability_distribution()`
  $\mathbf{U_b} \leftarrow \mathbf{U_b}^{'}$

number x such that $0 < x \leq P_{N_s}^{r,s}$, where N_s is the number of total *SalesOutlets* in industrial sector s. Searching the distribution $\mathbf{P}^{r,s}$, each buyer identifies the seller $o_j^{r,s}$, which satisfies $P_{j-1}^{r,s} < x \leq P_j^{r,s}$, and visits $o_j^{r,s}$ to buy the required goods. If the seller $o_j^{r,s}$ thus selected is active (i.e., goods are available for sale), the buyer's visit is successful and it purchases all it needs or all that is available. Buyers keep visiting sellers until they have either purchased their desired quantities of goods or all the *SalesOutlets* are sold out. As the `buy()` progresses, the number of sold-out *SalesOutlets* increases, rapidly increasing the number of unsuccessful buy attempts of unsatisfied buyers. Each buy attempt requires random access to arrays containing $\mathbf{P}^{r,s}$ and $\mathbf{o}^{r,s}$, which are large in volume, making each unsuccessful buy attempt a large waste of computational resources. Therefore, sold-out sellers are removed from the distribution as soon as they are sold-out, and the distribution is regenerated as illustrated in Fig. 6. We refer to this implementation as *naive_update_distr* (Algorithm 1 in the supplementary file). Deleting random elements of large arrays requires large amounts of data to be copied, and regenerating the distribution requires each element $p_i^{r,s}$ of active *SalesOutlets* to be accessed to update their values and to be re-normalized by dividing by $P_{N_a}^{r,s}$, where N_a is the number of active *SalesOutlets*. These time consuming memory operations on large arrays render *naive_update_distr* highly inefficient.

Compared to the deletion of elements from $\mathbf{P}^{r,s}$, a more computationally efficient approach is to disable any sold-out *SalesOutlet* $o_j^{r,s}$ by updating the cumulative distribution $\mathbf{P}^{r,s}$ as follows.

$$P_i^{r,s} = \begin{cases} P_i^{r,s} - p_j^{r,s}, & \text{for } j \leq i \leq N_s \\ P_i^{r,s}, & \text{otherwise} \end{cases} \quad (4)$$

Given a random value x, the selection criterion $P_{k-1}^{r,s} < x \leq P_k^{r,s}$ ensures that disabled sellers will not be selected from the updated $\mathbf{P}^{r,s}$. Further, there is no need to update $p_j^{r,s}$, as disabling does not change the probability $p_j^{r,s}$ of an active seller j. This implementation is referred to as *improved_update_distr* (Algorithm 2 in the supplementary file).

The *improved_update_distr* implementation is efficient when whole $\mathbf{P}^{r,s}$ can fit into the available cache memory of the respective MPI rank. However, when simulating large-scale models, like the 1:1 scale model of the eurozone, the memory requirement for $\mathbf{P}^{r,s}$ for some industries can be several times the size of available cache memory per rank. When the size of $\mathbf{P}^{r,s}$ is so large, both the random selection of seller from $\mathbf{P}^{r,s}$ and disabling agents introduce a large run-time overhead. Selection of a seller according to the distribution $\mathbf{P}^{r,s}$, requires the j to be found such that $P_{j-1}^{r,s} < x \leq P_j^{r,s}$ using an algorithm similar to binary search. When the data size of $\mathbf{P}^{r,s}$ is much larger than the available cache memory, binary search requires the repeated loading of subsets of $\mathbf{P}^{r,s}$ directly from RAM. As searching and disabling happens several billion times in large-scale simulations, this can introduce an excessive run-time overhead.

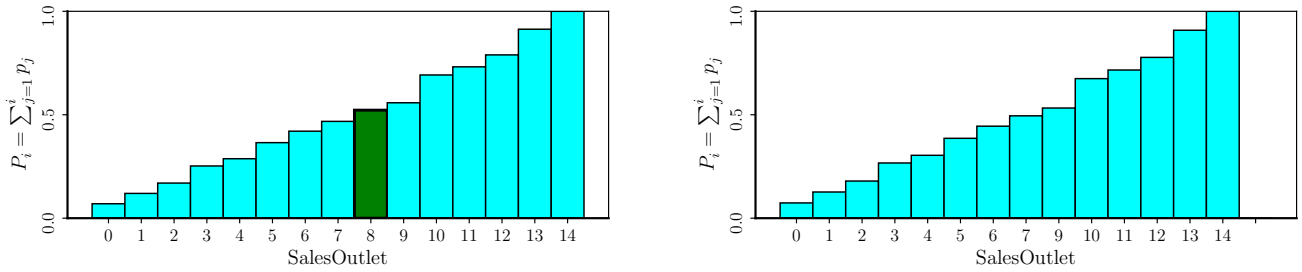
These excessive overheads can be significantly reduced by partitioning $\mathbf{P}^{r,s}$ and maintaining a table of the values at the partition boundaries. $\mathbf{P}^{r,s}$ is partitioned into m number of subsets of length $l(= {}^{N_s}/_{m})$ such that l elements fit into the available cache memory. Then a search table consisting of the $P_j^{r,s}$s at the boundaries of the divisions $\mathbf{Q} = \left\{ P_j^{r,s} \,|j \bmod l = 0 \right\}$ is generated. Now, the excessive memory-access overheads in the search can be eliminated by first searching the table for k such that $Q_k \leq x < Q_{k+1}$, and then searching $\mathbf{P}^{r,s}$ for the random value x in the sub-range $k \times l \leq j < (k+1) \times l$. Further, the sold-out sellers are disabled by updating $\mathbf{P}^{r,s}$ according to Equation 5, and $\mathbf{Q}$ is updated accordingly.

$$P_i^{r,s} = \begin{cases} P_i^{r,s} + p_j^{r,s}, & \text{for } 0 \leq i < j-1; \text{ if } j < {}^{N_s}/_{2} \\ P_i^{r,s} - p_j^{r,s}, & \text{for } j \leq i \leq N_s; \text{ if } j \geq {}^{N_s}/_{2}, \end{cases} \quad (5)$$

As shown in Fig. 7, this requires a maximum of ${}^{N_s}/_{2}$ elements of $\mathbf{P}^{r,s}$ to be updated and does not involve the heavy memory copy operation. It is also clear from Fig. 7 that the criterion for selection $P_{j-1}^{r,s} < x \leq P_j^{r,s}$ ensures that a disabled seller will not be selected. This implementation with improved search and disable is referred to as *advanced_update_distr* (Algorithm 3 in the supplementary file).

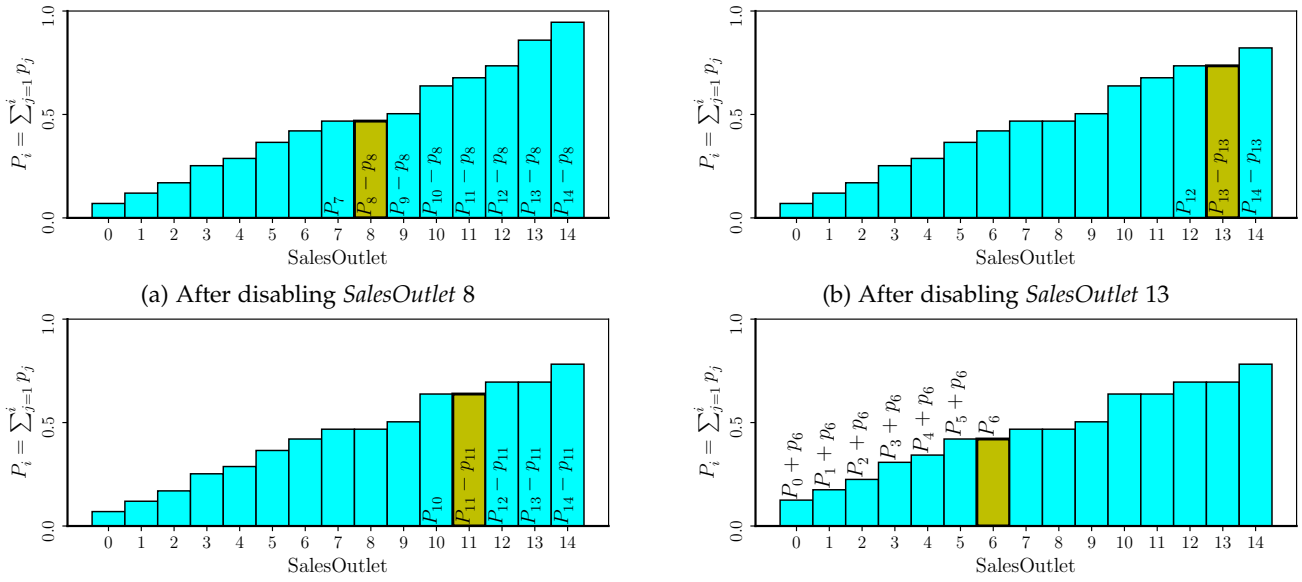
3.5.2 Data-oriented design

In the `buy()`, millions of customer agents visit several thousands of sellers of several tens of industries, exchanging a large volume of data. This is a memory-bound operation, and the size of customer objects and the memory-access patterns will have a notable impact on the run-time. In our



(a) Initial distribution (b) After deleting *SalesOutlet* 8, and re-normalizing

Fig. 6: Cumulative probability distribution $\mathbf{P}^{r,s}$ of *SalesOutlets* $o^{r,s}$ of sector s in rank r in case of *naive_update_distr*. The superscripts r, s of $P_j^{r,s}$ and $o_i^{r,s}$ are excluded for the sake of simplicity.



(a) After disabling *SalesOutlet* 8 (b) After disabling *SalesOutlet* 13

(c) After disabling *SalesOutlet* 11 (d) After disabling *SalesOutlet* 6

Fig. 7: Demonstration of *advanced_update_distr*'s updating of cumulative probability distribution $\mathbf{P}^{r,s}$.

implementation, for example, the household customer objects have at least 15 `double`, 2 `float`, and 4 `int` variables. However, of these, only 6 `double` variables are involved in the buying process. Moreover, to provide fair opportunities to all, the buyers are made to go shopping in a random sequence. The random accesses to a large array consisting of bloated objects cause poor cache performance and severely impact the overall performance of the code. To overcome this problem, new lightweight agents consisting of only `buy()`-related 6 variables of the customers are used as proxies for buyers. An array C is used to store all these lightweight buyers. Iterating through this lightweight array C improves the memory performance. For cache-friendly design, the array C is randomized before the start of the `buy()`. As element-wise randomizing is still a memory-intensive task, the Fisher-Yates algorithm [15] is utilized. As the array C consists of millions of elements, block-wise shuffling with a block size of a few thousand is sufficient to provide unbiased opportunities to the buyers. This implementation is referred to as *data-oriented_buy*.

3.5.3 Selling by sellers of big industries

The improvements presented in section 3.5.1 significantly improve the cache performance of maintaining and drawing random *SalesOutlets* $o_i^{r,s}$ from $\mathbf{P}^{r,s}$. However, it was observed that for the sectors with millions of sellers (i.e., $N_s \sim \mathcal{O}(10^6)$), these improvements are not sufficient. Some sectors of large economies like the eurozone have millions of *SalesOutlets*; 8 of the 62 sectors in the eurozone have more than 1 million *SalesOutlets*. The size of arrays containing millions of *SalesOutlets*, each having 6 `double`s, is much larger than the available L3 cache memory shared by multiple CPU cores. To overcome this issue, the *SalesOutlets* of large sectors are partitioned into n independent subsets with a maximum size of L, where $6 \times 8 \times L$ is less than L3 cache memory available to a CPU core. For the k^{th} subset, the cumulative distribution $\mathbf{P}^{r,s,k}$, where $\mathbf{P}^{r,s,k} = \{P_i^{r,s,k} | P_i^{r,s} = \sum_{j=k\times L}^{i} p_j^{r,s}; k \times L \leq i < ((k+1) \times L - 1)\}$, is generated. To choose a random seller $o_i^{r,s}$ from the subset k, a buyer draws a uniform random value x, $0 < x < P_{N_k}^{r,s,k}$, where $N_k (\leq L)$ is the number of sellers in the k^{th} subset. At the start of `buy()` of sector s, a randomly selected subset k is activated. The buyers visit sellers of the active subset, and after a random number of buy interactions, another subset

is activated. The optimal value of L depends on the specifications of the platform and can easily be determined by trial and error. $L = 100,000$ produced high-performance in the system we used for the simulations given in section 4. This implementation is referred to as *salesoutlets_subsets_buy*.

3.5.4 Big buyers `buy()`

Buyers in goods market comprise local and foreign firms, government agencies, investors, workers, and inactive households. Some of these agents, like government, wealthy investors, large firms, etc., have large buying powers (i.e., large financial wealth). Total demand for goods of sector s in rank r, $D^{r,s}$, is equal to $\sum_{i=1}^{{}^{B}N^{r,s}} C_i^{r,s}$, where $C_i^{r,s}$ is the consumption budget of i^{th} buyer and ${}^{B}N^{r,s}$ is the number of buyers of sector s. Because of non-uniform distribution of wealthy buyers over ranks, the demand $D^{r,s}$ is not evenly distributed among ranks, causing a large load imbalance in `buy()`. Making the large buyers buy in all the ranks is one effective way of making the demands $D^{r,s}$ uniform over the ranks, and thereby reducing the load imbalance. The threshold value of consumption budget $C_i^{r,s}$ to designate a buyer $B_i^{r,s}$ as a big buyer can be determined by measuring the computational performance. For the data sets used in this paper, all the firms with more than 100 workers and their investors, bank investor, and government are designated as big buyers. To distribute the buying action of a big buyer $B_i^{\bar{r},s}$ located in rank $\bar{r}$, we introduce a new agent *ProcurementDivision* $b_i^{r,s}$ in each rank r, and the government agent is made to buy in all ranks through its branches *LocalGovts*. At the start of `buy()`, $B_i^{\bar{r},s}$ distributes its consumption budget equally (i.e., ${C_i^{\bar{r},s}}/{P}$) to its *ProcurementDivisions* $b_i^{r,s}$ and collects the goods bought at the end of `buy()`. The communications among *ProcurementDivisions* and their owners are designed on the lines of those of *SalesOutlets* and their owners, as explained in 3.3.1. As demand becomes uniformly distributed, the load imbalance and run-time reduce, improving scalability. This implementation is referred to as *dstr_big_buy*.

4 Computational Performance

We developed a high performance version of the ABEM proposed by Poledna *et al.* [8] based on the strategies presented in the previous section. This section presents several numerical tests to quantitatively examine the effectiveness of those strategies in improving the performance of the code.

4.1 Problem settings

Two data sets are used in the numerical tests: a small economy with a total of 9.8 million agents, including 732,289 firms (herein referred to as *Small-Economy*), and a large economy with a total of 331 million agents, including 22.6 million firms (herein referred to as *Large-Economy*). Both the economies have 62 industrial sectors. *Small-Economy* and *Large-Economy* roughly represent the Austrian economy and the eurozone economy, respectively.

One simulaton period corresponds to one financial quarter. Each simulation is conducted for 13 time periods (one initialization period and 12 quarters ahead forecast) with different numbers of MPI processes in the Oakbridge-CX supercomputer of the University of Tokyo. Each computing node consists of Intel® Xeon® Platinum 8280 (2.7 GHz with 28 cores)×2 socket, and 192 GB memory with 281 GB/s bandwidth. The interconnection network is Intel® Omni-Path Host Fabric Interface (12.5 GBps).

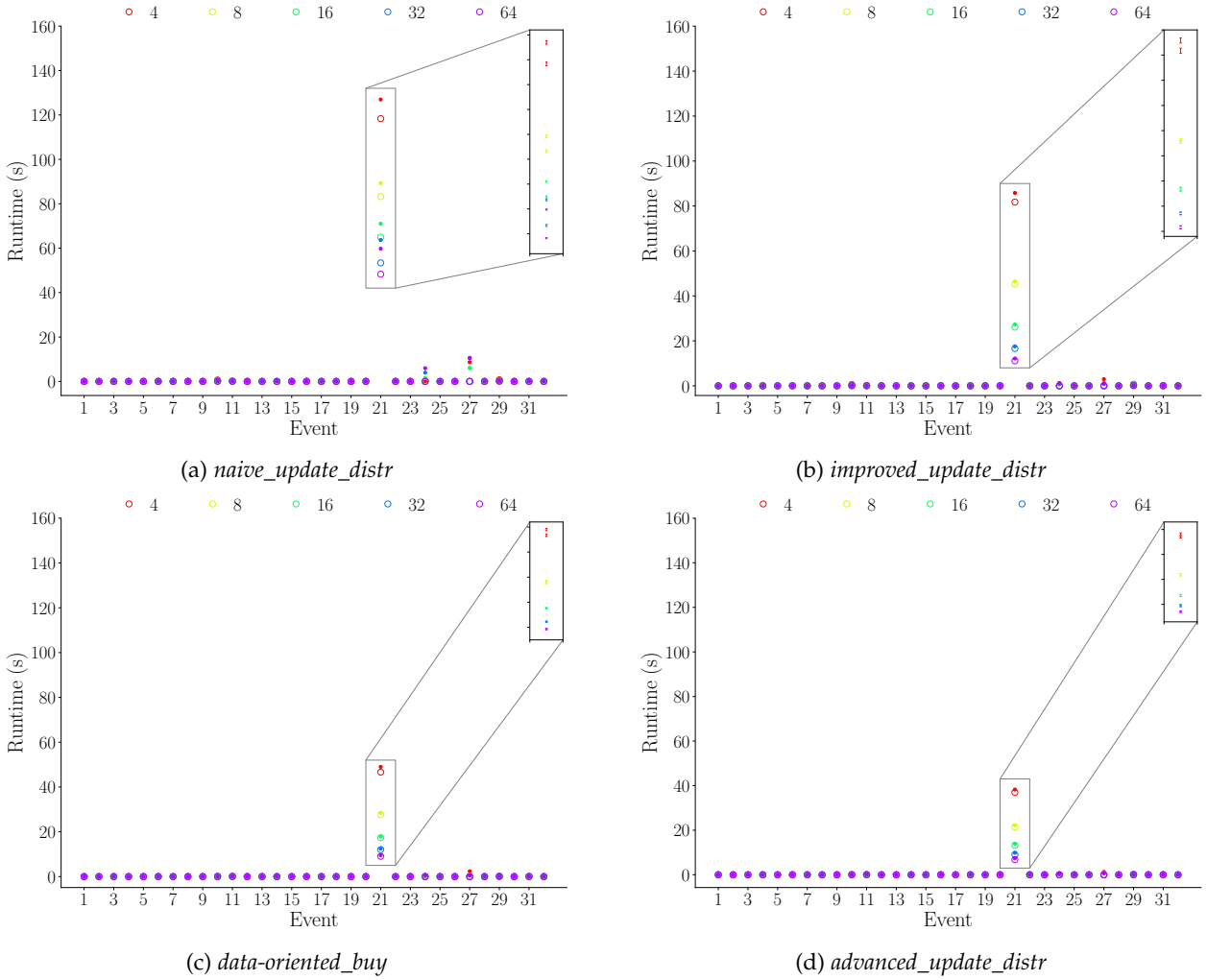
4.2 Performance of algorithmic improvements

As mentioned earlier, `buy()` is the most time-consuming event, and its performance depends on how the active sellers are managed as the goods market progresses, as well as the uniformity of the demand distributed among various ranks. First, the performance analysis of algorithmic improvements discussed in sections 3.5.1 and 3.5.2 is presented. The strategies presented in section 3.5.1 aim to enhance the performance of `buy()` by managing the *SalesOutlets* of various industrial sectors in a cache-friendly manner, whereas those in 3.5.2 enhance the cache performance by using lightweight buyers. Four simulations corresponding to the *naive_update_distr*, *improved_update_distr*, *data-oriented_buy*, and *advanced_update_distr* were conducted with each of the two problems to check the effectiveness of the improvements. The code including all these improvements is referred to as *first_version* (see Algorithm 4 in the supplimentry file for the pseudo code of the kernel).

Fig. 8 presents the run-time details of the *Small_Economy*, averaged over 13 iterations, with different numbers of MPI-ranks. The centers of dots and circles represent the maximum and minimum run-time taken by each of the 32 events in the main loop, respectively; the distances between the centers indicate the load imbalances in respective events. Events related to *RecruitmentAgency* are not included in these simulations. As Fig. 8 shows, these algorithmic improvements have drastically reduced the run-time and load imbalance of the most time consuming event, event 21, the `buy()`. These numerical results confirm that the *first_version* has sufficient performance for practical applications for economies of a few ten million agents.

Exposing a severe performance bottleneck of the *first_version*, a single iteration of *Large-Economy* with 1024 MPI-ranks required around 5700 sec, which is excessively large: a 34-times-larger problem took 19,826 (${}^{5700}/_{18.40} \times {}^{1024}/_{16}$) times more computational resources than *Small-Economy*. Fig. 9 presents the run-times taken by the slowest and fastest ranks to finish `buy()` of each sector. As the figure shows, there is significant load imbalance in `buy()` of sectors 0, 26, 28, 29, 35, 43, 44, 45, 52, 55, and 61, which have more than 500,000 *SalesOutlets* (see Fig. 10). As 500,000 *SalesOutlets* do not fit into the available CPU cache, drawing a seller from the distribution and disabling a sold-out seller require repeated loading of data from RAM, which makes these operations very slow. It is to address this issue that we introduced subsets of *SalesOutlets*, as discussed in section 3.5.3. Fig. 11a illustrates the minimum and maximum run-times taken by 1024 MPI-ranks for one iteration when *first_version* is made to use *SalesOutlet* subsets with a maximum size of 100,000. As can be seen, the use of subsets has drastically reduced the load imbalances in all the sectors: imbalance in sector 26 drops from 2520 sec to just 73 sec.

The remaining imbalance comes from the non uniform distribution of demands among ranks. When the demands



(a) *naive_update_distr* (b) *improved_update_distr*

(c) *data-oriented_buy* (d) *advanced_update_distr*

Fig. 8: Mean run-times (of 13 iterations) of 32 events of *Small-Economy*. Zoomed views show the standard deviations of the event 21. The difference between the centers of dot and circle corresponding to an event denotes the load imbalance.

of big buyers are uniformly distributed among the ranks, as explained in the section 3.5.4, the load-imbalance reduces to a few seconds, as shown in Fig. 11b. We refer to this implementation of the code as *final_version* (see Algorithm 6 in the supplimentry file for the pseudo code of the kernel).

The *final_version* includes all the strategies presented in this paper and has 50 events in the main loop. These events include all the economic processes, including non-local hiring/firing and the initializing and finalizing of many non-blocking communications. The `buy()` is separated into firms' `buy()` and non-firm buyers' `buy()` to hide some communications.

4.3 Events and their load imbalances in the *final_version*

As demonstrated in the previous section, various strategies have significantly reduced the load imbalance in `buy()`. This section studies the performance of all the events of *final_version* against load imbalance. For this purpose, two simulations, one for *Small-Economy* and one for *Large-Economy*, were conducted, and the results are presented in Fig. 12.

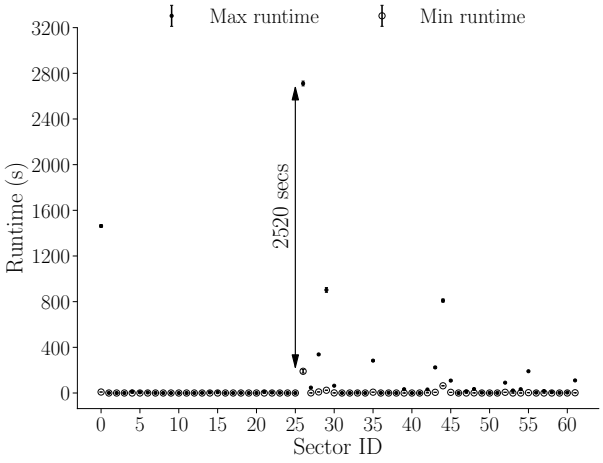


Fig. 9: *first_version*: Mean run-times along with standard deviations of `buy()` of each sector for 13 Monte Carlo simulations of *Large-Economy* using 1024 MPI-ranks.

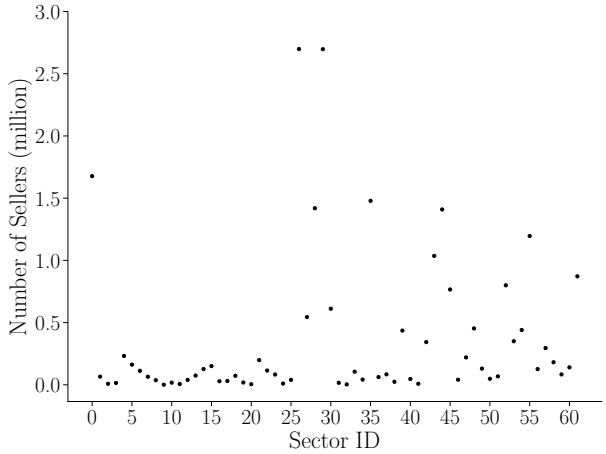


Fig. 10: Number of sellers in the sectors of the *Large-Economy*

As Fig. 12 shows, most of the events except 32, 34, 36, 41, and 43 take a negligible amount of time. Some of these events are initialization and finalization of non-blocking messages, and others are various economic processes. The most time-consuming event is event 34, which represents `buy()` of millions of non-firm buyers. As there are millions of random interactions between buyers and sellers causing millions of random memory accesses, event 34 takes a significant amount of time. Event 34 is followed by event 32, which is firm buyers' `buy()`. Apart from these events being the most time-consuming, there is a slight load imbalance in them. The reason for the load imbalance is the imbalance in the demand distributed among ranks, as discussed in the previous section. The load imbalance of event 32 is taken over to event 36, which is the finalization of the non-blocking communication posted in event 33 for the reduction of the quantity bought by distributed firm buyers. The imbalances of both events 32 and 34 are carried over to events 41 and 43, which finalize and post non-blocking communications for each of the *SalesOutlets*, respectively. Event 41 finalizes the non-blocking communication posted in event 39 for reducing the sales record of *SalesOutlets* at the master rank, and posts another non-blocking communication for scattering the reduced sales record back to the ranks containing owner firms. Event 43 finalizes the communication posted in event 41 and assigns the received sales records to the owner firms.

Furthermore, a comparison of Fig. 8d and 12a shows that, for *Small-Economy* simulations, *final_version* consumes approximately half of the time consumed by *first_version* to complete the most time-consuming event `buy()`. Even for *Large-Economy*, *final_version* takes only a few hundred seconds using just 16 MPI-ranks.

4.4 Scalability

Table 1a and 1b show the average run-time of 13 runs and the corresponding strong scalability of the *final_version*, with both the *Small-Economy* and the *Large-Economy*. The strong scalability is a standard performance measure in parallel computing and is defined as $(T_n/T_m)/(m/n)$, where T_n is the average run-time taken by n number of ranks, and $m \geq 2n$. As is evident from the tables, the code is quite fast and possesses a high, strong scalability for both small- and large-scale problems. Even large economies with hundreds of millions of agents can be simulated using small computational resources in a reasonably short time.

(a) *Small-Economy* simulations

MPI ranks	Run-time (s)	Scalability (%)
4	22.92	
8	12.03	95.26
16	6.57	91.55
32	3.93	83.58
64	2.53	77.66

(b) *Large-Economy* simulations

MPI ranks	Run-time (s)	Scalability (%)
16	483.75	
32	241.23	100.26
64	152.95	78.85
128	108.95	70.19
256	80.56	67.62

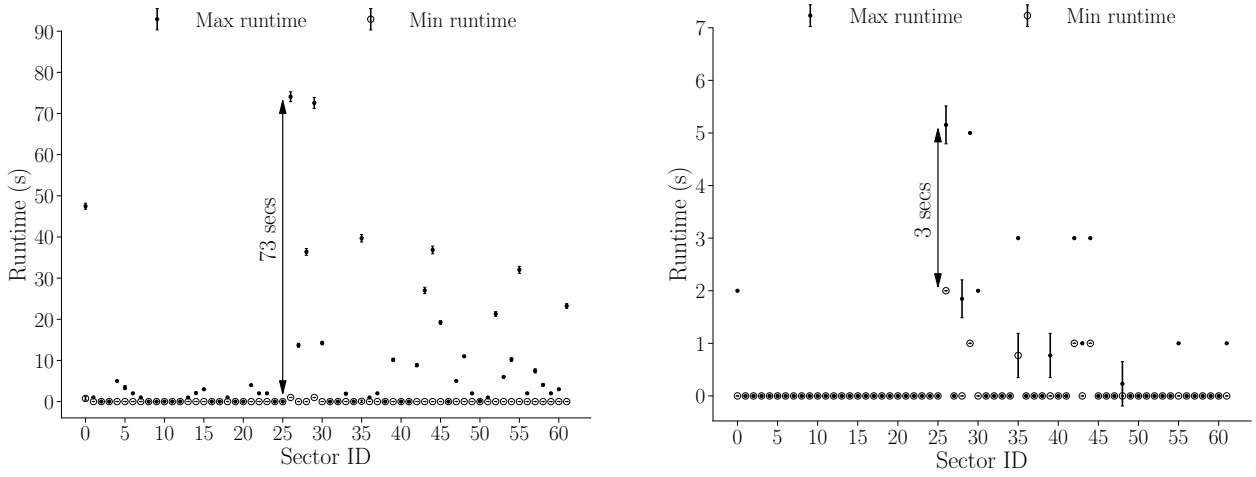
TABLE 1: Run-time and strong scalability for *Small-Economy* and *Large-Economy* simulations

5 MPI + OpenMP Hybrid Implementation

The MPI-only implementation discussed so far provides a fast and scalable parallel implementation with a reasonably low requirement for computational resources. Due to the dense and random nature of the interactions, it is difficult to achieve good scalability using shared-memory parallelization. However, a hybrid implementation can be utilized to reduce the memory requirements and the computation time when per node memory is small. This section presents an MPI + OpenMP hybrid implementation targeting the optimal use of computing nodes with limited memory.

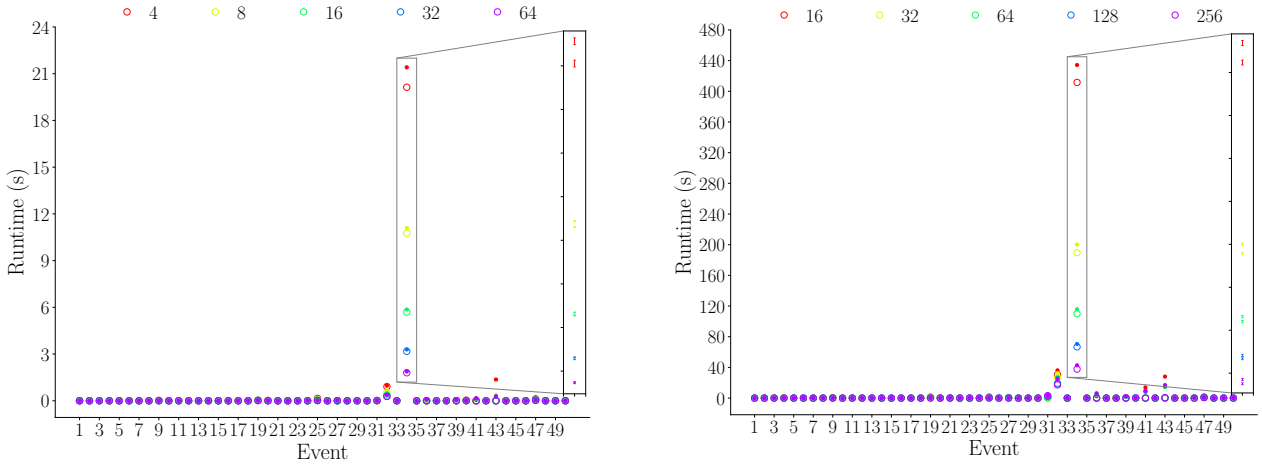
As explained in section 3.3.1, the nature of goods market is such that all the *SalesOutlets* must be copied in all the MPI-ranks. In large economies, there can be several million *SalesOutlets*; for example, there are 22 million *SalesOutlets* (each with 6 `double` variables) in *Large-Economy*, requiring 1.056 GB ($6 \times 8 \times 22 \times 10^6$ Bytes) memory per MPI-rank. In modern computing nodes with many-core CPUs, this large memory requirement may lead to a waste of computational resources, unless CPU cores are equipped with a large enough memory. As an example, the A64FX CPU, developed for the Fugaku supercomputer, has 48 CPU cores and only 32 GB of on-chip high-bandwidth memory. In such cases, less than half of the CPU cores can be utilized in MPI-only implementation.

The use of MPI + OpenMP hybrid parallelization to share the *SalesOutlets* assigned to an MPI-rank among multiple threads is an effective means of reducing the wastage of computational resources in memory-deprived many-core computing nodes. As buyers buy from one sector at a time and buying from two sectors is two independent operations, OpenMP threads can be utilized to make buyers buy



(a) *first_version + salesoutlets_subsets_buy* (b) *final_version*

Fig. 11: Mean run-times along with standard deviations of `buy()` of each sector for 13 Monte Carlo simulations of *Large-Economy* using 1024 MPI-ranks.



(a) *Small-Economy* simulation (b) *Large-Economy* simulation

Fig. 12: Mean runtimes (of 13 iterations) for each event. Zoomed views show the standard deviations of the event 34. The difference between the centers of dot and circle corresponding to an event denotes the load imbalance.

from different sectors concurrently, and thereby utilize the unused CPU cores. However, this requires all the buyers assigned to an MPI-rank to be copied to all the corresponding OpenMP threads, which, as a downside, increases the memory consumption. The trade-off between MPI-only and hybrid implementation depends on the number of MPI ranks and OpenMP threads. As an example, consider simulation of *Large-Economy* with n_M MPI-ranks, each with n_O OpenMP threads. Altogether, the 300 million buyers, each with 2 `int` and 8 `double` variables, in *Large-Economy* require 21.6 GB ($9 \times 8 \times 300 \times 10^6$ Bytes) memory. Therefore, the hybrid implementation requires $(21.6 \times n_O + 1.056 \times n_M)$ GB in order to store sellers and buyers. When $n_M = 56$ and $n_O = 1$, the total memory requirement is 80.736 GB, whereas for $n_M = 28$ and $n_O = 2$, it is 72.77 GB. This shows that n_M and n_O have to be decided based on the specifications of the computing nodes, as well as on the amount of run-time reduction brought by the OpenMP threads.

To check the performance of the hybrid implementation, *Large-Economy* simulations were conducted. Table 2 presents the average run-time of 13 runs and corresponding strong scalability. The implementation possesses a strong scalability of up to 80%. According to our tests, the distributed memory implementation is predominant in the hybrid implementation, and the implementation works best with a large number of MPI-ranks and 2 to 8 OpenMP threads.

TABLE 2: Run-time and strong scalability of MPI+OpenMP hybrid implementation for *Large-Economy* simulations

MPI ranks	OpenMP threads	Run-time (s)	Scalability (%)
56	1	157.15	
	2	96.72	81.24
	3	79.81	80.79
	4	74.45	80.40

6 Concluding Remarks

A scalable HPC implementation of an ABEM capable of performing 1:1 scale simulations of large economies is presented. The challenges posed by dense, random, and dynamic interaction graphs are addressed by partitioning the agents based on a representative employer-employee interaction graph to assign balanced loads to MPI-ranks. By mimicking real-life solutions, the unknown large number of MPI messages required in the original ABEM is drastically reduced to a handful of well-organized communications overlapped with computations, thereby gaining a significant parallel performance. The performance of the goods market, the most time-consuming event, is enhanced by maintaining the sellers' cumulative probability distribution in a cache-friendly data structure. This implementation with the cache-friendly goods market is sufficient for simulating medium-size economies with a maximum of 100,000 sellers per sector. The cache performance of the goods market is further improved by using seller subsets to simulate large economies with millions of sellers in a sector. All these strategies make the implementation highly scalable and capable of simulating large economies within a few minutes utilizing small computational resources. Further, an MPI + OpenMP hybrid implementation has been developed to utilize all the computational resources with low per-core memory capacity. It is demonstrated that the hybrid implementation requires a relatively small amount of memory and possesses a strong scalability of up to 80%.

Acknowledgments

This work was supported by JSPS Kakenhi grant 18H01675. Parts of the results were obtained using K computer at the RIKEN Center for Computational Science (hp150283), and the Oakbridge-CX supercomputer at the Univ. of Tokyo.

References

[1] C. Deissenberg, S. Van Der Hoog, and H. Dawid, "Eurace: A massively parallel agent-based model of the european economy," *Applied Mathematics and Computation*, vol. 204, no. 2, pp. 541–552, 2008.
[2] F. D. Farmer, J., "The economy needs agent-based modelling," *Nature*, vol. 460, pp. 685–686, 2020.
[3] S. Chen, *Agent-Based Computational Economics: How the idea originated and where it is going*, ser. Routledge Advances in Experimental and Computable Economics.
[4] D. Helbing, "The futurict knowledge accelerator: Unleashing the power of information for a sustainable future," 2010.
[5] "Eurace: An agent-based software platform for european economic policy design with heterogeneous interacting agents: new insights from a bottom up approach to economic modelling and simulation," https://cordis.europa.eu/project/id/035086.
[6] "Symphony: Orchestrating information technologies and global systems science for policy design and regulation of a resilient and sustainable global economy," https://cordis.europa.eu/project/id/611875.
[7] "The futurict knowledge accelerator:creating socially interactive information technologies for a sustainable future," https://cordis.europa.eu/project/id/284709.
[8] S. Poledna, M. G. Miess, and C. H. Hommes, "Economic forecasting with an agent-based model," *Available at SSRN 3484768*, 2020.
[9] X. Rubio-Campillo, "Pandora: a versatile agent-based modelling platform for social simulation," *Proceedings of SIMUL*, pp. 29–34, 2014.
[10] N. Collier and M. North, "Parallel agent-based simulation with repast for high performance computing," *Simulation*, vol. 89, no. 10, pp. 1215–1235, 2013.
[11] R. K. Standish, "Going stupid with ecolab," *Simulation*, vol. 84, no. 12, pp. 611–618, 2008.
[12] S. Coakley, L. Chin, M. Holcomb, C. Greenough, and D. Worth, "Flexible large-scale agent modelling environment (flame). university of sheffield and rutherford appleton laboratories, stfc, license: Lesser gpl v3," 2012.
[13] M. Lalith, A. Gill, S. Poledna, M. Hori, I. Hikaru, N. Tomoyuki, T. Koyo, and T. Ichimura, "Distributed memory parallel implementation of agent-based economic models," in *International Conference on Computational Science*. Springer, 2019, pp. 419–433.
[14] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.
[15] R. Durstenfeld, "Algorithm 235: random permutation," *Communications of the ACM*, vol. 7, no. 7, p. 420, 1964.

Amit Gill Amit Gill received a B.Tech. in Civil Eng. from the Indian Institute of Technology Roorkee, Roorkee, India in 2014. He obtained his M.E. in Civil Eng. from the Univ. of Tokyo in 2018 and is currently pursuing a Ph.D. there. His research interests lie in HPC and integrated large-scale simulations of disasters and economy.

Madegedara Lalith Lalith Wijerathne received a B.E. in Civil Eng. from the Univ. of Peradeniya, Peradeniya, Sri Lanka in 1999. He obtained his M.E. and Ph.D. degrees in Civil Eng. from the Univ. of Tokyo, Japan in 2000, and 2005, respectively. He is currently an associate professor with the Earthquake Research Institute (ERI) and the Dept. of Civil Eng. of the Univ. of Tokyo. His research interests lie in HPC, agent-based modeling, large-scale simulations of crack propagation phenomena, and integrated simulations of earthquake disasters and their aftermath.

Sebastian Poledna Sebastian Poledna holds a magister degree in physics and in economics and business administration. In 2016, he received his doctorate in physics from the Univ. of Vienna. He currently leads the research group on Exploratory Modeling of Human-natural Systems at the International Institute for Applied Systems Analysis (IIASA) and is a research fellow of the Institute for Advanced Studies Vienna (IHS). He was previously a research fellow at the Institute for Advanced Study of the University of Amsterdam and a visiting researcher at the ERI of the Univ. of Tokyo.

Muneo Hori Muni Hori received a B.E. degree in Civil Eng. from the Univ. of Tokyo, Japan in 1984 and an M.E. in Civil Eng. from Northwestern University, Evanston, Illinois, USA, in 1985. He obtained his Ph.D. in applied mechanics and engineering sciences from the Univ. of California, San Diego, USA in 1987. He served as a professor of Civil Eng. at the Univ. of Tokyo, Japan, until 2018 and is currently working as Director-General of Research Institute for Value-Added-Information Generation at the JAMSTEC, Yokohama, Japan. His research interests lie in large-scale integrated earthquake simulations, social science simulations, meta-modeling of structures for rational numerical computation, tera models of Earth, and fusion of remote sensing and simulation.

Kohei Fujita Kohei Fujita received his B.E., M.E., and Ph.D. degrees in Civil Eng. from The Univ. of Tokyo, Japan in 2010, 2012, and 2014, respectively. He is currently an assistant professor with the ERI and the Dept. of Civil Eng. at the Univ. of Tokyo. His research interests lie in HPC and finite-element solvers in large scale earthquake simulations.

Tsuyoshi Ichimura Tsuyoshi Ichimura received his B.E., M.E., and Ph.D. degrees in Civil Eng. from the Univ. of Tokyo, Japan in 1998, 1999, and 2001, respectively. He is currently a professor with the ERI of the Univ. of Tokyo, Japan. His research interests lie in numerical algorithms enhanced by linear algebra, parallel computing and advanced computer architectures, finite element solvers, urban disaster simulations, crust deformation analysis, and AI applications in disaster simulations.