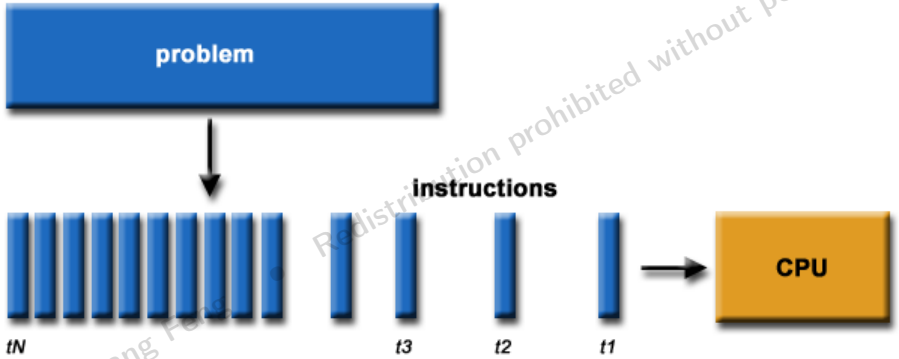
Quant Macro w/ Machine Learning
6: Parallel Computing

Zhigang Feng

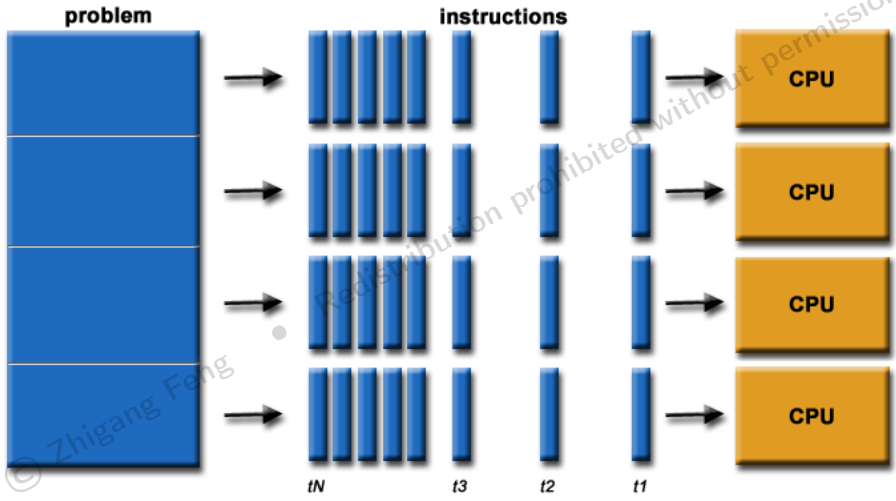
University of Nebraska
Zhongnan University of Economics and Law

November 4, 2025

Serial computing



Parallel computing



Benefit of parallel computing

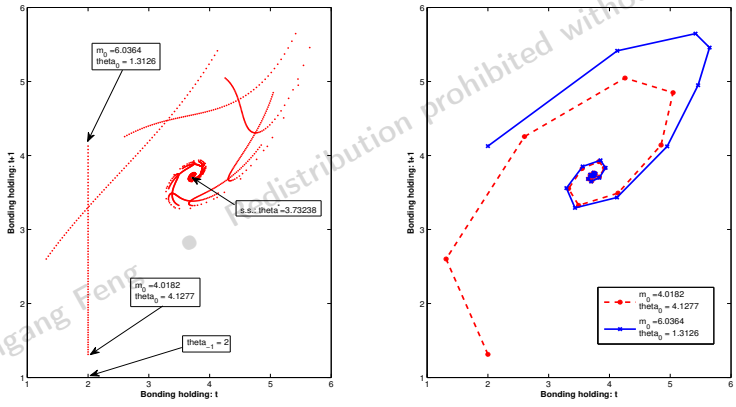
- Time Efficiency: Parallel computing can significantly reduce the amount of time needed to solve a problem. Tasks are divided and executed concurrently, leading to faster results. This is particularly advantageous in situations where real-time or near-real-time processing is needed.
- Cost Efficiency: Parallel computing allows for more efficient use of expensive resources. Instead of purchasing one high-performance supercomputer, multiple lower-cost machines can be used to achieve similar or better performance. The cost of electricity, cooling, and maintenance may also be lower.
- Parallel computing allows you to tackle problems that would be impossible or impractical to solve on a single machine. This is due to the increased computational power and combined memory resources of multiple processors. This can be particularly beneficial in fields like scientific computing, big data analysis, or machine learning, where large-scale problem-solving is common.

Benefit of parallel computing

- Parallel computing is not confined to a single location. You can use resources distributed across different locations or even different continents. This can help improve the reliability of your system (through redundancy) and allow you to leverage geographically distributed resources, which can be more cost-effective or efficient depending on the task.
- Serial computing, where tasks are completed one after the other, has physical and practical limitations. Processor speed can only be increased so much before issues like overheating occur, and serial computing is inherently limited by the time it takes to execute each task sequentially.
- Parallel computing overcomes these limitations by enabling multiple tasks to be processed simultaneously. This scalability is one of the key advantages of parallel computing and is becoming increasingly important as data sizes and computational complexities continue to grow.

Application in Economics

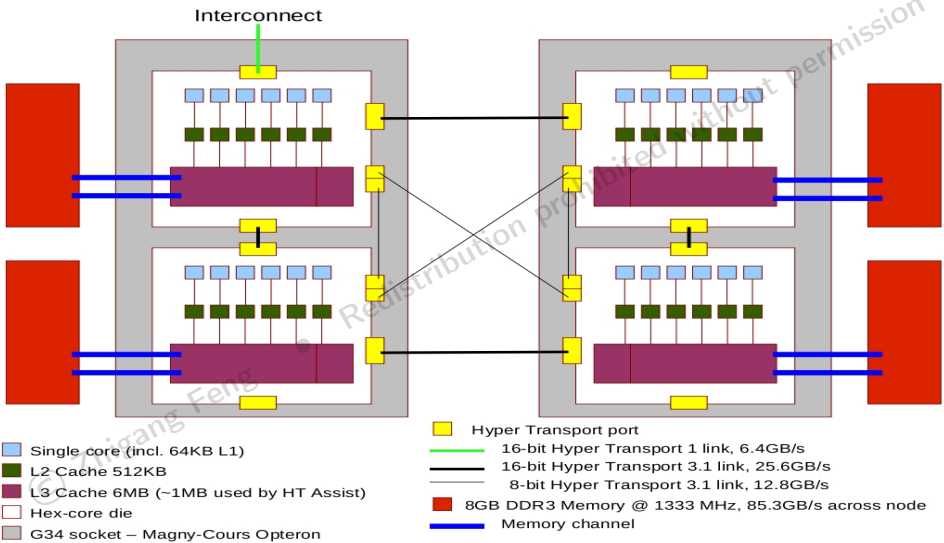
Figure: OLG with Indeterminacy



Application in Economics

- Macroeconomic models often require the computation of equilibrium or the solution of high-dimensional optimization problems. Parallel computing allows for the efficient simulation of these models, even when they include many sectors, regions, or agents.
- Cai and Lontzek (2019) solved their model in 8 hours using 110,688 cores in parallel on the Blue Waters supercomputer.

Hardware for parallel computing



Concepts and terminology

- Supercomputing / High Performance Computing (HPC)
  - Supercomputing, also known as High Performance Computing (HPC), is the use of parallel processing for running advanced application programs efficiently, reliably, and quickly.
  - These are typically used for solving advanced computation-intensive problems.
- CPU / Socket / Processor / Core
  - CPU (Central Processing Unit): The primary component of a computer that performs most of the processing inside the computer. It interprets and carries out the basic instructions that operate a computer.
  - Socket: The component that provides the mechanical and electrical connections between the processor and the motherboard.
  - Core: A processor core is an independent processing unit within the CPU that reads and executes program instructions

Concepts and terminology

- Shared Memory
  - Shared memory is a method where multiple processors in a system have access to the same memory. It's typically used in parallel computing for efficient data exchange between threads.
  - An example would be using shared memory in a multi-threaded programming model where threads can access and modify the same memory.
- Distributed Memory
  - In distributed memory systems, each processor has its own private memory (computational memory). Information is exchanged by passing messages between the processors.
  - This method can handle larger, more complex computations as you can add more nodes as needed.

Concepts and terminology

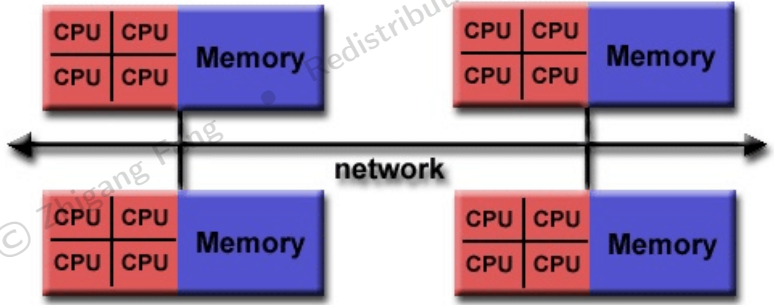
- Communications
  - In parallel computing, communication refers to how individual processes exchange data and coordinate their actions.
  - The choice of communication method has a significant impact on the performance of parallel programs.
- Synchronization
  - Synchronization is the coordination of threads to ensure the correct sequence of computation and access to shared resources.
  - It helps prevent issues like race conditions (A race condition occurs in a multi-threaded environment when two or more threads can access shared data and they try to change it at the same time) or deadlocks (a state in which each member of a group is waiting for another member, including itself, to take action, such as sending a message or more commonly releasing a lock).

Concepts and terminology

- Massively Parallel
  - Massively parallel refers to the use of a large number of processors (or separate computers) simultaneously.
  - This term is typically used to describe "supercomputers" that perform complex calculations involving large amounts of data.
- Embarrassingly Parallel
  - Embarrassingly parallel is a term used to describe a problem that can be naturally separated into a large number of independent tasks. Each task can be solved without needing to communicate with any others.
  - Examples include simulations or parameter sweeps.
- Scalability
  - Scalability in the context of parallel computing refers to a system's ability to maintain performance levels as the number of processors (or the load on the processors) is increased.

Parallel computer memory architectures

- Shared Memory:
  - Pros: easy to program, data sharing are fast.
  - Cons: lack of scalability between memory and CPUs.
- Distributed Memory:
  - Pros: Memory is scalable with the number of processors. Cost effectiveness.
  - More involved in programming.
  - Hybrid Distributed-Shared Memory

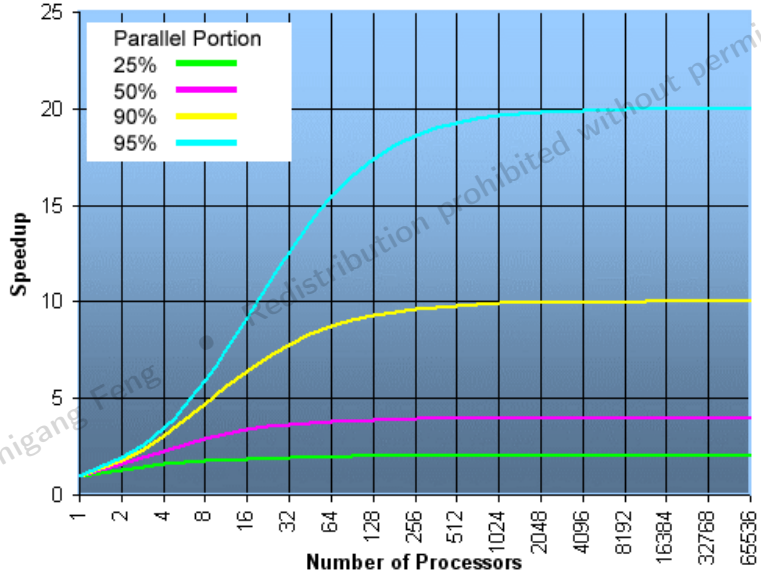


Limits and costs of parallel computing

- Amdahl's Law: a principle from computer science which offers a way to find the maximum improvement to an overall system when only a part of that system can be improved. It is named after the computer scientist Gene Amdahl, and is commonly used in parallel computing to predict the theoretical maximum speedup using multiple processors.

$$speedup = \frac{1}{\frac{P}{\#CPU} + S}$$

Limits and costs of parallel computing



An example: evaluate function at grid points

- Serial:

```
do j = 1,n
  do i = 1,n
    a(i,j) = fcn(i,j)
  end do
end do
```

- Parallel:

```
do j = mystart, myend
  do i = 1,n
    a(i,j) = fcn(i,j)
  end do
end do
```

Basic MPI operations

- Before any MPI routine is called, the MPI environment must be initialized by MPI_Init. It should be noted that all MPI programs must call MPI_Init and it must be called before any other MPI routine. MPI_Finalize is used to terminate the MPI environment. No more MPI routines may be called after MPI_Finalize.
  MPI_Init, MPI_Finalize
- To find out the total number of processes
  MPI_Comm_size

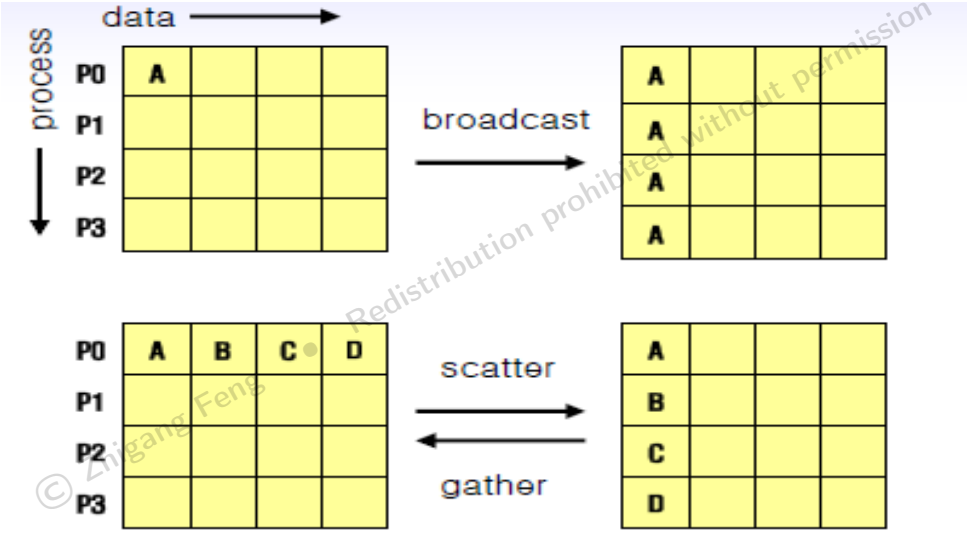
Basic MPI operations

- MPI_Comm_rank returns the rank of the calling process within the specified communicator.
  MPI_Comm_rank
- MPI_Gather is a collective communication routine that gathers data from all processes and delivers it to the root process.
  MPI_Gather
- MPI_Bcast is a collective communication routine that broadcasts a message from the process with rank "root" to all other processes within the communicator.
  MPI_Bcast

Parallel example - Pseudo code

```
MPI_Init
MPI_Comm_size
MPI_Comm_rank
    if I am MASTER
      do until no more jobs
        calculate a(i,j) = fcn(i,j)
      end do
      COLLECT DATA FROM WORKER
MPI_Gather
      DISTRIBUTE DATA TO WORKER
MPI_Bcast
    else if I am WORKER
      do until no more jobs
        calculate a(i,j) = fcn(i,j), send results to MASTER
      end do
endif
MPI_Finalize
```

Basic Operators



A canonical optimal growth model

- The household solves the following dynamic optimization problem

$$\max_{\{c_t, k_{t+1}\}_{t=0}^{\infty}} \sum_{t=0}^{\infty} \beta^t \log c_t$$

$$c_t + k_{t+1} = A k_t^{\alpha} + (1-\delta) k_t$$

$$\beta \in (0,1),\ A > 0,\ \alpha \in (0,1),\ \delta \in [0,1]$$

$$k_t,\ c_t \geq 0,\ k_0 \text{ given},\ t = 0, 1, 2, ...$$

- Recursive formulation

$$v(k) = \max_{k_+} \log c + \beta v(k_+)$$

$$(k, k_+) \in \Omega$$

$$\Omega := \{(k, k_+) | A k^{\alpha} + (1-\delta)k - c - k_+ \geq 0,\ k \in R^+,\ k_+ \in R^+\}$$

- Equilibrium: value function $v(k)$, policy function $k_+ = g(k)$.

Numerical algorithm

1. Discritization of the state space:

$$K = \bigcup_i K^i,$$

   we approximate $v(k)$ with $\{k_i \to v(k_i)\}_{i=1}^{N_k}$.
2. Initial step: set an accuracy level ϵ and an initial guess $v^0(k)$.
3. Value function evaluation: let

$$v^{(n+1)}(k_i) = \max_{k_+} \log\left[Ak^\alpha + (1-\delta)k - k_+\right] + \beta v^{(n)}(k_+)$$

4. End of iteration: if $\| v^{(n+1)} - v^{(n)} \| \leq \epsilon$ then stop; else $v^{(n)} = v^{(n+1)}$ ans return to step 3.

Parallelization - Pseudo code

do until convergence reached
  computation
    all CPU
      do until no more jobs
        calculate element: $V^1(k_i) = \max u(c(k_i)) + \beta V^0(k_+)$
        send results to MASTER
      end do
  updating
    if I am MASTER
      collect dada for V_1 *from WORKERs*
      set $V_0 = V_1$ *and distribute* V_0 *to WORKERs*
end do

Sample code

```
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &n_proc);
do {
  for (int iv = 0; iv < Nk; iv++) { vzero[iv] = -vone[iv]; }
  for (INT IK = 0; IK < NK/N_PROC; IK++) {
      k0 = k_vec[RANK*NK/N_PROC + IK];
      minimizor(value_function);
      argmax_k1 = xmin;
      msg[ik] = value_function(tmp_k1);
  }
      to be continued..
} while (!checkEnd());
```

Sample code

```
do {
        continued..
   MPI_Barrier(MPI_COMM_WORLD);
   stat = MPI_Gather(msg, N_msg, MPI_DOUBLE, msg_Array,
          N_msg, MPI_DOUBLE, root, MPI_COMM_WORLD);
   if (rank == root){
     for (int row = 0; row < Nk; row++) {
       vone[ik] = msg_Array[ik];
     }
   }
   stat = MPI_Bcast (vone, Nk, MPI_DOUBLE, root,
          MPI_COMM_WORLD);
} while (!checkEnd());
```

Parallel Computing in PyTorch

- Data Parallelism: Parallelizing the data input, where the data is divided into smaller chunks and each chunk is processed independently.
- Model Parallelism: The model is split across the computing units, each computing unit calculates different parts of the same model.
- Hybrid Parallelism: A combination of both data and model parallelism.

Parallel Computing in PyTorch

- PyTorch provides utilities for both Data Parallelism and Model Parallelism.
- The PyTorch tensors can be distributed across devices and machines to parallelize the computations.
- PyTorch supports GPU acceleration using NVIDIA's CUDA, making computations significantly faster.

Using GPU with PyTorch

- PyTorch makes it easy to switch between CPU and GPU. To move tensors or models to GPU, we use the .to method.
- The device where a tensor is currently located can be accessed with .device attribute.
- Example of how to move tensors between devices.

```
import torch
# Check if CUDA is available and set the device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# Define a tensor
tensor_cpu = torch.zeros(2, 2)
# Check the current device of tensor
print(tensor_cpu.device) # This will output: cpu
# Move the tensor to GPU if it's available
tensor_gpu = tensor_cpu.to(device)
# Check the current device of tensor
print(tensor_gpu.device) # This will output: cuda:0 if GPU is available, else cpu
```

Data Parallelism in PyTorch

- PyTorch's nn.DataParallel class allows us to wrap any module and it will be parallelized over batch dimension. This enables simultaneous computation over multiple GPUs.
- An example code for using nn.DataParallel.

```
# Check if multiple GPUs are available
if torch.cuda.device_count() > 1:
  print("Let's use", torch.cuda.device_count(), "GPUs!")
  # dim = 0 [30, xxx] -> [10, ...], [10, ...], [10, ...] on 3 GPUs
  model = nn.DataParallel(model)
# Define your input data
input_data = torch.randn(30, 10)
# Use the model
output = model(input_data)
```

- In this example, if multiple GPUs are available, we wrap our model with nn.DataParallel, which will distribute the computations across all GPUs. The input data is automatically split on the first dimension and sent to each GPU, where a separate process computes the forward pass. Then, the results are collected back onto the first GPU.

Model Parallelism in PyTorch

- PyTorch allows you to split your model into various parts and run these parts on different devices by using nn.Module.
- An example code for implementing model parallelism.

```
class Model(nn.Module):
  def __init__(self, device1, device2):
    super(Model, self).__init__()
    self.device1 = device1
    self.device2 = device2
    self.layer1 = nn.Linear(10, 10).to(device1)
    self.layer2 = nn.Linear(10, 10).to(device2)
  def forward(self, x):
    # Forward pass on the first device
    x = self.layer1(x.to(self.device1))
    # Move tensor to the next device
    x = x.to(self.device2)
    # Forward pass on the second device
    x = self.layer2(x)
    return x
```

Model Parallelism in PyTorch

- In this example, we manually split our model across two GPUs. The first layer of the model is placed on cuda:0 and the second layer on cuda:1. During the forward pass, the input data is first moved to cuda:0 where it goes through layer1. The output of layer1 is then moved to cuda:1 where it goes through layer2.
- While this does allow for larger models that don't fit on a single GPU, it doesn't provide any performance improvement unless the computation time for the two parts of the model is roughly equal. Also, the process of moving data between GPUs can introduce significant overhead, which might even lead to slower computations compared to using a single GPU.

Distributed Data Parallelism in PyTorch

- DistributedDataParallel (DDP) is a wrapper that allows us to easily parallelize computations across multiple GPUs.
  - DDP wraps around any PyTorch module and replicates it across multiple GPUs.
  - Each GPU is assigned a portion of the input data and performs forward and backward passes independently.
  - The gradients calculated on each GPU are then synchronized across all GPUs.
- As part of the torch.nn.parallel package in PyTorch, DDP reduces the amount of computation per GPU, allows us to train larger models and batches. It uses collective communications from torch.distributed package for synchronizing gradients and buffers.

Working with DistributedDataParallel

- DistributedDataParallel (DDP) is a special form of Data Parallelism.
- While Data Parallelism splits the input data across multiple GPUs within a single machine, DDP extends this concept to work across multiple machines, each with multiple GPUs.
- Just like in Data Parallelism, in DDP, each GPU computes the entire model but on a different subset of the data.