Parallel Algorithm Principles

Pangfeng Liu
National Taiwan University

February 26, 2022

Principles

There are three basic principles in improving the efficiency of parallel computing.

- Even partition
- Communication reduction
- Efficient Implementation

Partition

- Partition is an essential parallel algorithm design technique.
- As in a sequential divide-and-conquer algorithm, the problem is first partitioned (divided) into sub-problems.
- Unlike a sequential divide-and-conquer algorithm, a parallel algorithm solves (conquers) the sub-problem *in parallel*.
- Some communication may be necessary since the sub-problems may depend on other tasks or may need to transfer data among themselves.
- Finally, we combine the answers from individual sub-problems into the final answer.

Partition

- Partition is the first step in a divide-and-conquer algorithm.
- One can partition the data and the process *data partitioning*.
- Or one can partition the main loop of the computation, and it is *loop* partitioning.
- The partition has a significant impact on the overall performance.

Discussion

- Give an example of divide-and-conquer computation.

Partition Principles

There are two important issues in partitioning.

- Even workload distribution
- Proper granularity

Even Workload Distribution

- We want to distribute the workload among processors to minimize the maximum workload among all processors.
- The execution time of a parallel program is the execution time of the *slowest* processor involved, which is usually the processor that has the maximum workload.
  - This is the *makespan* of the execution time. Note that usually, we are interested in the makespan of the execution, not the sum or the average of the execution time of processors.

Idle v.s. Busy

- Uneven distribution of workload leaves some processor idle while others are busy.
- If everyone is busy all the time, then the workload is evenly distributed.

Workload Estimation

- To distribute the workload evenly, one needs to predict the workload accurately.
- For data-parallel computation, one can associate the computation with the data. If we further assume that the computation workload on every data is about *the same*, then we can estimate the workload by counting the *number* of data each processor is assigned.

Workload Estimation

- For task parallel computation, we must predict the workload of sub-problems.
- It is difficult to estimate the workload of tasks, so profiling or programmer intervention is necessary.

Discussion

- Give an example to illustrate the importance of even workload distribution.

Granularity

- The granularity is the basic unit in partitioning.
- For data-parallel computation, it indicates the smallest chunk of data while assigning data chunks to processors.
- For task-parallel computation, it indicates the smallest chunk of task while assigning tasks to processors.
  - Recall that we can always *refine* a step of our algorithm into finer steps.

The Size

- It is always easier to balance the workload if the granularity is *small* because it is always easier to distribute a set of objects evenly if we can *cut them into small pieces*.
- There will be much more overhead in assigning these chunks to processors because the mapping table will be larger and in scheduling and synchronizing the processors because the number of these operations will increase.
- More details on the communication later.

Fine and Coarse

- *Fine grain* parallelism partitions data/task into very small pieces, then assigns them to processors for processing.
  - Suitable for system that can spawn a large number of threads with low cost, e.g., GPU.
- *Coarse grain* parallelism partitions data/task into very large pieces, then assigns them to processors for processing.
  - Suitable for system that can only spawn a limited number of threads, and the thread creation is expensive, e.g., CPU.

Discussion

- Give an example to illustrate the importance of granularity in partitioning workload.

Communication Reduction

- Communication is inevitable because multiple processors are working on the *same* problem.
- Communication is overhead – it does not appear in a sequential computation.
- Communication should be reduced.

Principles

- There are two basic principles to reduce communication.
  - Low synchronization overheads
  - Data locality

Synchronization

- The synchronization is inevitable in parallel and distributed computing because we want to coordinate the processors.
  - Barrier synchronization
  - Before/after synchronization
  - Access synchronization

Barrier Synchronization

- A computation may proceed in *stages* – all processors needs to finish a stage before going to the next stage.
  - This is usually called a *barrier* synchronization. For example, all processors must combine their partial answer into the final answer.
  - This usually involves *all* processors.

Before/After Synchronization

- In task parallelism one computation may need to precede another.
  - You need to cook dinner before you can eat it. This may be referred to as *before/after* synchronization.
  - This usually involves two processors – one processor finishes a computation, then notifies the other processor to proceed.

Access Synchronization

- Many processor may need to access a shared variable in a shared memory multiprocessor.
  - Not an issue for distributed memory multicomputer since the computers do not share memory.
- If the memory access is not synchronized properly, race condition may occur.

Synchronization Mechanism

- Many parallel programming environments provide a mechanism for program to specify synchronization *explicitly*.
- The synchronization should be efficient.
- The synchronization should be scalable, i.e., it should be efficient even if the number of processors involved is large.

Discussion

- Give an example for each synchronization described earlier.

Synchronization Mechanism

- One can use message passing or shared memory to implement barrier synchronization within the same computer.
- One can use signal inter-process communication to implement before/after synchronization within the same computer.
- One can use busy waiting or semaphore to implement the critical section for accessing shared variables.
- If processors of different computers need to synchronize, they need to use a network protocol.

Synchronization Optimization

- We should reduce the number of stages.
- The synchronization should be efficient.
- We should carefully choose the granularity to balance synchronization and workload distribution overhead.
  - A fine-grain parallel computation is hard to synchronize but easy to have an even workload.
  - A coarse-grain parallel computation is easy to synchronize but hard to have an even workload.

Discussion

- Describe the inter-process communication (IPC) mechanism that you are aware of.

Data Locality

- *Locality* is a trend for a program to access data/instruction in *proximity*.
- When a program access a data/instruction, it is very likely it will access the same data/instruction in the dear future, or it will access the data/instruction nearby in the near future.
- Computer architecture explores locality for performance.

Temporal Locality

- When a program access a data/instruction, it is very likely to access the same data/instruction shortly.
- If we *cache* this data/instruction in fast storage, then it is very likely we will be able to access the data fast.
- Data/instruction are cached in data/instruction cache for performance.
- CPU first tries to get the data from the cache.
- If the CPU finds it in the cache, it uses the data; otherwise, it gets it from memory.
- There could be several levels of caches.

Performance

- The performance comes from the difference in accessing speed to memory and cache and the probability of finding the data/cache in the cache.
- If we can find the data/instruction in cache with high probability, i.e., with a high cache hit rate; then the performance will improve.
- If the temporal locality is good, which means the CPU will use the same data/instruction again shortly, then we have good performance.

Shortly?

What do we mean *shortly*?

- The capacity of cache is minimal.
- When we access a data/cache, we have to place it into the cache for possible later references.
- If the cache is *full*, then we have to remove some data/instructions to make space for the incoming ones.
- *Shortly* means when we want to access the data/instruction we placed into cache *again*, it will still be there, i.e., before we remove it to make room for other data.

Other Applications

- Hard disks maintain a small cache for data stored in the disk.
- Operating system maintains disk cache for frequently accessed data on disk.
- A translation lookaside buffer (TLB) is a cache for frequently accesses items in the page table.

Discussion

- Give an example of temporal locality.

Spacial Locality

- When a program access a data/instruction, it will access the data/instruction *nearby* in the near future.
- If we *cache* the near by data/instruction in a fast storage, then it is very likely we will be able to access the nearby data/instruction fast.
- Parallel processing focuses on *spacial data locality*.

Cache Line

- Modern computer architecture does not cache data individually; instead, it caches data/instruction in the unit of a cache line.
- A cache line consists of consecutive data/instruction in memory.
- A cache line automatically caches nearby data/instructions and improves spatial locality.
- Parallel programmers preserve data locality in a much higher *data level* when partitioning the data into chunks for processing.

Data Level Locality

- When we assign data to processors for processing, we not only want to distribute them evenly, we also want to preserve *spacial data locality*.
- That means when we want to process a data, the *required* data is *nearby*.
  - What is required data?
  - What is "near by"?

Required Data

- When we process a data, we usually need *other* data.
- For example, when we want to compute vector C, which is the sum of two vectors A and B.
- We need A_i and B_i to compute C_i, then A_i and B_i are required data of C_i.

Owner

- We usually follow a *owner computes* rule.
- If a processor is the owner of data, then it is responsible for the computation of this data.
- The rule is simple.
- On rare occasion we will not follow the *owner computes* rule.

Placement

- If the length of the vector is 32, and we have two processors, how do we assign data to processors?
- Intuitively, we can place the first 16 elements of A, B, and C to one processor and the rest to the other.
- The workload of computing C is even because each processor computes 16 elements for C.
- When a processor computes a A_i, it can get all the required data within its memory.

Wrong Placement

- If the length of the vector is 32, and we have two processors.
- We place the first 16 elements of A, B, and the last 16 elements of C to one processor and the rest to the other.
- The workload of computing C is even because each processor will compute 16 elements for C.
- When a processor computes a A_i, it *cannot* get any required data within its memory.
- Is this good?

Nearby

- *Nearby* means in the same processor.
- We can access the required data within the processor of the same processor by *memory bandwidth*.
- We can only access the required data within the processor of other processors by *network bandwidth*.
- Memory bandwidth is *much much larger* than network bandwidth.

Local v.s. Remote

- We use *Local memory* to indicate the memory of the same processor, and *remote memory* as the memory of other processors.
- We conclude that *Local memory* is much much faster than *remote memory*.
- This distinction applies only to distributed memory multicomputer.

Goal

- If *most* of the required data is nearby; then we have good performance.
- That is, we want to ensure that most of the required data are nearby, i.e., in local memory, when we assign data to processors for computation.
- Note that we say *most* because sometimes it is impossible to partition data so that all data access is local.

Discussion

- Give an example of spacial locality.

Matrix Multiplication

- We multiple matrix A and B and get C.
- The required data of C_{ij} is the i'th row of A and j'th column of B.
- If we insist that the required data must be in local memory, then everything will be in one processor!
- This is against the principle of *even workload distribution*.

Proof

- C_{ij} has to be in the same processor as the i'th row of A and j'th column of B.
- C_{kl} has to be in the same processor as the k'th row of A and l'th column of B.
- Then C_{kj} has to be in the same processor as the k'th row of A and j'th column of B.

Proof

- This implies C_{ij} and C_{kj} have to be in the same processor because they are in the same processor as the j'th column of B.
- Similarly C_{kj} and C_{kl} have to be in the same processor because they are in the same processor as the k'th row of A.
- We conclude that C_{ij} must be in the same processor as C_{kl}, for any i, j, k, and l.
- Finally, all data will be in the same processor, which is bad.

Best Effort

- If *most* of the required data is in local memory; then we have good performance.
- We would like to increase the percentage of access to local memory, which is the best effort.
- We need to carefully partition data to preserve locality.

Communication-to-Computation Ratio

- Another metric to understand the data locality is the computation-to-communication ratio.
- The amount of computation is roughly the same throughout different data partitioning.
- The amount of communication is proportional to remote data because local data do not incur communication.
- If the communication-to-computation ratio is small, we have small communication overheads, which means we have good data locality.

Discussion

- Give an example of good locality and another example of bad locality for the same problem, due to different partitioning methods.

Surface to Volume Ratio

- Sometimes we use a *surface-to-volume* ratio to explain communication-to-computation ratio.
- We now consider the entire data as an object, and data partitioning is a way to cut the object into pieces.

Neighbors

- In many computations, the required data are those *neighboring* data.
  - In an array, the neighboring data for an array element have indices differing from the element by 1.
  - In a graph, the neighboring data are those nodes that are adjacent to the node.
  - In a graphic computation, the neighboring data for a pixel are those that are adjacent to it.

Neighbors

- In a table for dynamic programming, the value of an element is usually determined by those elements that have indices differing from the element by 1.
- In a page ranking algorithm, the value of a node is a function of the neighboring nodes.
- In a graphic relaxing problem, the new value of a pixel is a function of the eight neighbors.

Discussion

- Give an example of computation that uses neighbors.

Pieces

- We can use the *volume* of a piece to represent the number of data in a piece, which in turn represents the amount of *computation*.
  - We assume that amount of workload is about the same for all data.
- We can also use the *surface area* of a piece to represent the number of *required data* in a piece, which in turn represents the amount of *communication*.
  - We assume that the required data are on the surface of the pieces.

Surface-to-volume Ratio

- Now we can easily relate the computation-to-communication ratio to the surface-to-volume ratio.
- We want to have small computation-to-communication ratio, then we must partition data into pieces that have small surface-volume-ratio.
  - Surface area is communication.
  - Volume is computation.

Discussion

- Give an example of surface-to-volume ratio. If an object has a large surface-to-volume ratio, is it easier, or harder, to coll down? How does that relate to communication costs?

An Example

- We are given a matrix of 32 by 32 by 32, and we would like to update each cell to be the average of its six neighbors with 8 processors.
- We have two choices.
  - We cut the matrix into eight 16 by 16 by 16 cubes.
  - We cut the matrix into eight 4 by 32 by 32 slates.

Cubes

- The volume of a cube is 16 by 16 by 16 $= 4k$.
- The surface area of a cube is $6 \times 16 \times 16 = 1.5k$.
- The surface to volume ratio is $1.5/4 = 3/8$.
- This means for the computation on each data the processors needs to access remote memory 3/8 times.

Slates

- The volume of a slate is 4 by 32 by 32 $= 4k$.
- The surface area of a slate is $2 \times 32 \times 32 + 4 \times 32 \times 4 = 2.5k$.
- The surface to volume ratio is $2.5/4 = 5/8$.
- This means for the computation on each data the processors needs to access remote memory $5/8$ times, which is more than the $3/8$ while cutting into cubes.

Lessons

- The surface-to-volume ratio is a reasonable estimate of the communication-to-computation ratio.
- It is intuitive to partition the data into chunks to minimize the surface, i.e., communication.
- For example, if we partition the data into a checkerboard pattern, the surface-to-volume ratio will be huge, and data locality will be poor.

Discussion

- Describe the difference in sizes of similar animals that live in tropical or Arctic area.

Efficiency

- How to synchronize processors efficiently?
  - Global synchronization
  - Point-to-point synchronization
- How to transfer data efficiently?
  - Batch mode message passing
  - Overlap communication with computation
  - Explore memory hierarchy

Global Synchronization

- Reduction
  - Every processor has a value for the solution of its sub-problem, and we want to compute the *sum* of these values.
  - Every processor has a value for the solution of its sub-problem, and we want to compute the *minimum* of these values.
  - A reduction also serves as a barrier synchronization.
- Barrier synchronization
  - One can think of a barrier synchronization as a special form of reduction in which no value is exchanged.

Tree Optimization

- We can ask a processor to coordinate the synchronization.
  - Inherent sequential and the coordinator is the bottleneck.
- Or we can organize the process as a tree.
  - We partition the processors into two subsets.
  - Two subsets recursively synchronize themselves *in parallel*.
  - Finally the two subsets synchronize with each other.
  - More details in lectures later.

Two Party Synchronization

- In a multiprocessor environment, the critical section or semaphore may not be the best synchronization solution.
- Unlike in a uni-processor environment, the critical section or semaphore overheads are very high in a multiprocessor environment.
- We sometimes prefer spin-locks in a multiprocessor environment, e.g., Linux kernel data structure.

Transfer Efficiency

- In many low level parallel programming environment, (e.g. OpenCL, CUDA, or MPI) the programmers can explicit control how data is transferred among professors.
- In these environments the programmer can apply the following techniques to improve data transfer efficiency.
  - Batch mode message sending
  - Overlap computation with communication
  - Explore memory hierarchy

Batch Mode

- Many message passing system is on top of network protocol like TCP/IP.
- These protocol has a fixed start-up overhead, e.g., to establish a connection in TCP/IP.
- If we send a large number of data through a connection, then the start-up overhead is amortized among the data begin transferred, which means we should transfer data in large quantity.

Overlap Communication with Computation

- It is beneficial to have many threads so that when a thread is waiting for data, other threads can use CPU resources for computation.
- For example, in GPU, many running threads can hide memory latency, i.e., when a thread is waiting for memory, other threads can use ALU for computations.
- This requires many threads and a flexible scheduler to schedule them.
- This relieves the burden of cache.
- More details in later lectures.

Explore Memory Hierarchy

- In some parallel programming environments (e.g., CUDA and OpenCL), the programmer can move data with the memory hierarchy.
- The processing units of GPU have fast and small local memory and share a slow and large global memory.
- CUDA and OpenCL programmers must *explicitly* move the data between the global and local memory to achieve performance.
- This is a tedious and error-prone process.
- More details on later lectures.