

Parallel Algorithm Examples

Pangfeng Liu
National Taiwan University

April 18, 2020

Embarrassingly Parallel

- An embarrassingly parallel computation is a collection of tasks that require *none* or *little* communication among them. In other words, they are *independent*.
- This is “embarrassing” since nothing needs to be done to get good parallel performance.
- People doing parallel processing, e.g. me, are not fond of this kind of computation.

Monte Carlo Method

- The Monte Carlo method repeats a *random process* to compute the answer.
- We generate random numbers as input to the computation, so that the answer can be deduced from the random process. Note that all computations are *independent*.
- It is important to use different “random seed” with these tasks, so that the results from them are *statistically independent*.

Compute π

- Randomly throw darts into a square with a inscribed circle.
- Compute the number of darts that fall into and outside the circle.
- Compute the probability that a darts falls into the circle.
- Multiple the probability by 4 to approximate π .

Notes

- This is not a good example because people will not compute π this way.
- Nevertheless this is an easy-to-understand example to illustrate the independence of tasks.

Parameter Search

- Suppose we want to find a set of “best” parameters $x = (x_1, \dots, x_n)$, which maximize an objective function $y = f(x)$.
- We also assume that the f function is very complex so that we cannot deduce the values of $f(x)$'s for x 's that we have not yet computed the function values, from those function values that we have already computed.
- It is easy to dispatch the computation of $f(x)$ to processors to speed up the parameter search, since there is no dependency between these computations.

File Serving

- A web server serves static HTML files to clients.
- The requests of clients are independent, so the web server simply serves the files in parallel, maybe using multiple threads.
- If there are multiple web servers, they can also serve the files in parallel.

Summary

- Embarrassingly parallel computation has good speedup and efficiency, because the tasks are independent and do not require significant communication.
- It is trivial to dispatch tasks to processors if the tasks require roughly the same amount of time.
- It is non-trivial to dispatch tasks to processors if the tasks require very different amount of time. In this case “load balancing” is required.

Discussion

- Give an example of embarrassingly parallel computation.

Divide and Conquer

- Divide-and-conquer is a common parallel algorithm design technique.
- As in a sequential divide-and-conquer algorithm, the problem is first divided into sub-problems.
- Unlike a sequential divide-and-conquer algorithm, a parallel algorithm solves (conquer) the sub-problem *in parallel*.
- Some communication may be necessary since the sub-problems may have dependency on each other.
- Finally the answers from individual sub-problems are combined into the final answer.

Summation

- We want to sum n numbers, and n is very large.
- It is easy to see that we can apply divide-and-conquer technique to solve the problem in parallel with p processors.

Parallel Summation Algorithm

- 1 Partition the numbers so that each processor has roughly n/p numbers.
- 2 Each processor computes the sum of assigned numbers.
- 3 A processor collects all the partial sums from other processors and compute the final sum.

Analysis

- We assume that first step takes very little time.¹
- The second steps takes $O(\frac{n}{p})$ times.
- The third steps takes $O(p)$ times.

The time complexity is as follows.

$$O(\frac{n}{p} + p) \quad (1)$$

¹This is a the case for shared memory model, but not necessarily true for distributed memory model.

Discussion

- How do we minimize the $O(\frac{n}{p} + p)$ by choosing the right p ?

Communication

- Having one processor collect all the answer is not efficient.
- We partition the processor into two groups. Every processor in the first group sends its answer to the corresponding processors in the second group.
- We repeatedly do this until we have only one processor left, who should have the final answer.

Parallel Summation Algorithm

- 1 Partition the numbers so each processor has n/p numbers.
- 2 Each processor computes the sum of its numbers.
- 3 Use the recursive algorithm to compute the final sum.
- 4 This is similar to the “tree optimization” in synchronization.

Analysis

- We assume that first step takes very little time.
- The second steps takes $O(\frac{n}{p})$ time.
- The third steps takes $O(\log p)$ time because the depth of a complete binary tree of n nodes is about $O(\log n)$.

The final time complexity is as follows.

$$O(\frac{n}{p} + \log p) \quad (2)$$

Discussion

- Describe the difference between the previous two algorithms.

Observation

- The first term ($\frac{n}{p}$) is *computation*. We can *never* reduce this part.
- The second term ($\log p$) is about *communication*. We try our best to reduce this part.
- If we increase p , the computation time decreases and the communication time increases. That means we have more workers to share the workload, but we need to communicate more among more workers.

More Observations

- It is important to balance the load among processors, i.e. we want to send $(\frac{n}{p})$ data to each processor for processing.
- A shared memory implementation is significantly easier than a distributed memory implementation.
- The recursive (or tree like) communication pattern is much more complicated than a naive one, and requires much more complicated synchronization.

Analysis

Speedup

$$k = \frac{n}{\frac{n}{p} + \log p} \quad (3)$$

Efficiency

$$e = \frac{n}{n + p \log p} \quad (4)$$

Choice of p

- What is the best p in terms of speedup?
- Set $\frac{n}{p} = \log p$ and solve $p = \frac{n}{\log n}$.
- The minimum parallel execution time $\Theta(\log n)$ is achieved when $p = \Theta(\frac{n}{\log n})$.
- If we set $p = \sqrt{n}$, then the time will be $\Theta(\sqrt{n})$, which is much larger than the optimal $\Theta(\log n)$.

Discussion

- Use calculus to compute the optimal P value.
- What will happen if we set p to n ?
- Is this “theoretical” optimal p useful in practice?

Prefix Sum

Given n numbers (x_1, \dots, x_n) , we want to compute all prefix sums as follows.

$$s_k = \sum_{i=1}^k x_i \quad (5)$$

Compact An Array

- The prefix sum has various applications.
- If there are zeros and non-zeros in an array A and we only wish to keep the non-zeros in a new array B , then we can do a prefix sum on another array P with 0 and 1 to determine the positions of non-zeros in B .

Pact an Array

A	before packng	4	0	9	0	0	8	5	7
		1	0	1	0	0	1	1	1
P	0 and 1								

prefix sum

B	new index after packing	1	1	2	2	2	3	4	5
		4	9	8	5	7			

Fibonacci's Numbers

- The prefix sum has various applications, and it is not limited to summation.
- We all know Fibonacci's numbers.

$$f_i = \begin{cases} 0 & i = 0 \\ 1 & i = 1 \\ f_{i-1} + f_{i-2} & i \geq 2 \end{cases} \quad (6)$$

Fibonacci's Numbers

$$f_1 = f_1 \quad (7)$$

$$f_2 = f_0 + f_1 \quad (8)$$

$$\begin{pmatrix} f_1 \\ f_2 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} f_0 \\ f_1 \end{pmatrix} \quad (9)$$

$$\begin{pmatrix} f_2 \\ f_3 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} f_1 \\ f_2 \end{pmatrix} \quad (10)$$

$$= \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} f_0 \\ f_1 \end{pmatrix} \quad (11)$$

$$\begin{pmatrix} f_i \\ f_{i+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^i \begin{pmatrix} f_0 \\ f_1 \end{pmatrix} \quad (12)$$

Prefix Product

Given n matrices (x_1, \dots, x_n) , we want to compute all prefix products as follows.

$$s_k = \prod_{i=1}^k x_i \quad (13)$$

Discussion

- Give an example of prefix sum application.

Parallel Prefix Sum Algorithm

- Use the k -th processor to compute s_k .

Analysis

- A sequential algorithm can do this easily in $O(n)$ time.
- We assume that we use one processor per data, so $p = n$.
- The k -th processor requires $O(k)$ time.
- The parallel time is the maximum of all processor time, hence $O(n)$.
- The speedup is $O(1)$, and the efficiency is $(\frac{1}{p})$.
- Not very efficient.

Discussion

- What is wrong with the previous algorithm?

Parallel Prefix Sum Algorithm

- To avoid doing duplicated work, we again use the k -th processor to compute s_k , but we get the result from the $k - 1$ -th processor.

$$s_k = s_{k-1} + x_k \quad (14)$$

Analysis

- Now we do not duplicate work, but we need to *wait*.
- The k -th processor cannot compute its sum before receiving s_{k-1} .
- The result will go ripple-like from the first to the last processor like a wave-front.
- The parallel time is the maximum of all processor time, hence $O(n)$.
- The speedup is $O(1)$, and the efficiency is $(\frac{1}{n})$.
- Again not very efficient.

Discussion

- What is wrong with the previous algorithm?

A Better Algorithm

- There are $\log n$ stages.
- In the i -stage every element adds the element 2^i to the left to itself.
 - In the first stage every element adds the element to its left to itself.
 - In the second stage every element adds the element two elements to its left to itself.

Parallel Prefix Sum Algorithm

```
for  $i \leftarrow 0$  TO  $\log n - 1$  do  
  for  $k \leftarrow 2^i$  TO  $n$  do  
     $x[k] \leftarrow x[k] + x[k - 2^i]$  {The  $k$ -th processor receives a partial sum  
    from the  $k - 2^i$ -th processor.}  
  end for  
end for
```

Parallel Prefix

2	1	7	4	2	3	1	5
2	3	8	11	6	5	4	6
2	3	10	14	14	16	10	11
2	3	10	14	16	19	20	25

Analysis

- There are $\log n$ steps, since $p = n$ now.
- A processor does a sum and receives a message, so the time is $O(1)$.
- The total parallel time is $O(\log n)$, which is much better than $O(n)$ in previous approaches.
- The speedup is $O(\frac{n}{\log n})$, and the efficiency is $(\frac{1}{\log n})$.

Discussion

- Prove that the algorithm is correct.
- What is the possible problem with the previous algorithm?

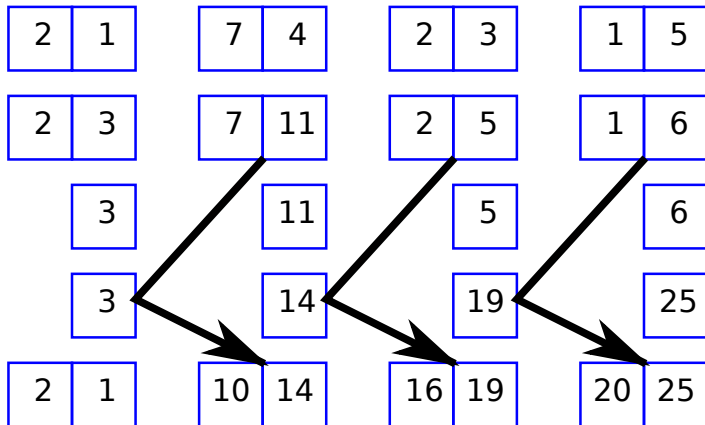
Improvement

- It is not practical to use one processor per data, since in practice the number of data is much more than the number of processors.
- We will assume that n is much larger than p , hence we need to partition the data among processors.
- Now each processor will compute the prefix sum of its data first, then use the previous algorithm to “patch” things up.

Improved Algorithm

- 1 Partition data among processors.
- 2 Each processor computes its prefix sum.
- 3 Use the previous algorithm to compute the prefix sum of the *last* elements from all processors.
- 4 Use the prefix sum from the last elements to patch up the answers.

Parallel Prefix



Analysis

- A sequential algorithm can do this easily in $O(n)$ time.
- The first step does not take time.
- Both the second and the fourth step take $O(\frac{n}{p})$ time.
- The third step takes $O(\log p)$, as discussed before.

$$T_p = O\left(\frac{n}{p} + \log p\right) \quad (15)$$

- Similar optimization can find good p .

Final Notes

- What did we compute?

$$s_4 = (((x_1 + x_2) + x_3) + x_4) \quad (16)$$

$$s_4 = ((x_1 + x_2) + (x_3 + x_4)) \quad (17)$$

Discussion

- What property the operation must have in order for this algorithm to work?
- Does “+” have this property?
- Does “maximum” have this property?
- Does matrix multiplication have this property?

Sorting

- To sort keys between 1 and n in order.
- We try the bucket sort first.

Bucket Sort

- We need b buckets.
- We need to know the range of keys, and we assume that the keys are *evenly* distributed in this range.
- We use an array element to record whether a key *appears* in the input.
- We do not compare!

Bucket Sort

- 1 Scan the keys and record its appearance in the corresponding buckets.
- 2 Read the keys from the buckets.

Bucket Sort

Example 1: Bucket sort

```

1 void bucketsort(int array[], int n, int b)
2 {
3     int i, j = 0;
4     int *bucket = calloc(b + 1, sizeof(int));
5     for (i = 0; i < n; i++)
6         bucket[array[i]]++;
7     for (i = 0; i <= b; i++)
8         while(bucket[i]--)
9             array[j++] = i;
10 }

```

2

²<http://www.eecs.ucf.edu/courses/cop3502h/spr2007/sorting3.pdf>



Bucket Sort

array

2	1	7	4	2	3	1	5
---	---	---	---	---	---	---	---

bucket

2	2	1	1	1	0	1	0	0	0
1	2	3	4	5	6	7	8	9	10

array

1	1	2	2	3	4	5	7
---	---	---	---	---	---	---	---

Analysis

- The first step takes $O(n)$ time.
- The second step takes $O(n + b)$ time.
- The total complexity is $O(n + b)$.

Discussion

- Why the second step takes $O(n + b)$, not $O(nb)$ time?

Parallel Bucket Sort

- 1 Every processor has exactly one bucket.
- 2 Every processor scans all keys to record keys corresponding to its bucket.
- 3 Read the keys from the buckets.

Bucket Sort

array

2	1	7	4	2	3	1	5
---	---	---	---	---	---	---	---

bucket

2	2	1	1	1	0	1	0	0	0
1	2	3	4	5	6	7	8	9	10

array

1	1	2	2	3	4	5	7
---	---	---	---	---	---	---	---

Parallel Bucket Sort

- Every processor takes $O(n)$ time just to scan data.
- Exactly how to “read the keys from the bucket”?

Discussion

- What is wrong with the previous algorithm?

Parallel Bucket Sort

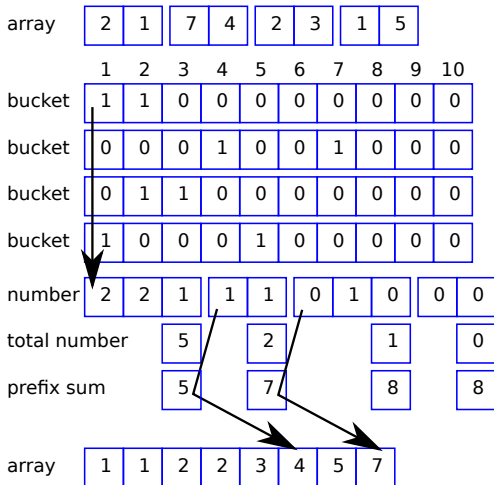
- Every processor reads only $\frac{n}{p}$ keys, and record the appearance into its own set of buckets.
- Now it only takes $O(\frac{n}{p})$ time.

Read Keys

How do we read the keys out in the second step?

- 1 Every processor remembers the number of keys it places in every bucket.
- 2 Each processor computes the number of keys that should be in its bucket.
- 3 Use the parallel prefix sum algorithm to know the starting position each bucket should start.
- 4 Actually read from the bucket.

Bucket Sort



Analysis

- 1 The first step takes $O(\frac{n}{p})$ time.
- 2 The second step takes $O(b)$ because each processor needs to add b numbers.
- 3 The third step takes $O(\log p)$.
- 4 the fourth step takes $O(\frac{n}{p})$.³

³Assuming the keys are evenly distributed among processors.

Final Time Complexity

- 1 The “read to bucket” takes $O(\frac{n}{p})$ time.
- 2 The “read from bucket” takes $O(\frac{n}{p} + \log p + b)$ time.
- 3 The total time complexity is $O(\frac{n}{p} + \log p + b)$.
- 4 The speedup is $O(\frac{n+b}{\frac{n}{p} + \log p + b})$, which is $O(\frac{n+b}{\log n + b})$ when we set p to $\frac{n}{\log n}$.

Discussion

- What is wrong with the previous algorithm?

Improvement

- The bucket sort uses (and wastes) a lot of memory.
- We will try a “quicker” sort that also uses the divide-and-conquer techniques.
- Again we assume that the keys are evenly distributed and we know the range.

Sequential “Quicker” Sort

Sort the keys recursively as follows.

- Partition keys into g groups according to $g - 1$ pivots.
- Individually sort the keys in each group recursively.
- Concatenate all the keys from the g groups.

Analysis

- The first step takes $O(n)$ time.
- Let the time to sort n keys be $T(n)$.

$$T(n) = gT\left(\frac{n}{g}\right) + n \quad (18)$$

- It is easy to see that $T(n) = O(n \log_g n) = O(n \log n)$.

Discussion

- Explain what will happen when there are only two groups in the “quicker” sort.

Parallel Quicker Sort

Every processor manages a group, which will store a range of keys.

- ① Every processor reads only $\frac{n}{p}$ keys, and puts them into the corresponding bucket.
- ② Each processor sorts the keys in its bucket.
- ③ Read all keys from processors.

Pact an Array

array

2	1	7	4	2	3	1	5
---	---	---	---	---	---	---	---

group

2	1	1	3	2
---	---	---	---	---

1	1	2	2	3
---	---	---	---	---

group

4	5			
---	---	--	--	--

4	5			
---	---	--	--	--

group

7				
---	--	--	--	--

7				
---	--	--	--	--

group

--	--	--	--	--

--	--	--	--	--

array

1	1	2	2	3	4	5	7
---	---	---	---	---	---	---	---

Analysis

- The first step takes $O(\frac{n}{p})$ time, without considering the synchronization.
- The second step takes $O(\frac{n}{p} \log \frac{n}{p})$, assuming that the keys are evenly distributed.
- The third step takes $O(\log p)$ from previous analysis on prefix sum.
- The total time is $O(\frac{n}{p} \log \frac{n}{p} + \log p)$.
- The speedup is $O(\frac{n \log n}{\frac{n}{p} \log \frac{n}{p} + \log p})$.

Discussion

- What synchronization mechanism do we need for the first step in the previous algorithm?
- Why we did not need it in the previous parallel bucket sort?

Exchange Sort

- We consider a recursive “exchange” sort that is much more suitable for distributed memory multicomputers.
- This algorithm is very suitable for hypercube.
- We do not require that the keys are distributed uniformly, since we can argue that the performance is *statistically* acceptable.

Exchange Sort

- Divide the processors into two groups of equal size.
- Each processor “exchanges” keys with its corresponding processor in the other group according to a pivot – smaller keys go to a group of processors and bigger keys go to the other group.
- Recursively do the same for both groups.

Exchange Sort

- 1 Find a pivot.
- 2 Divide the processors into two groups of equal size.
- 3 Each processor “exchanges” keys with its corresponding processor in the other group.
- 4 Recursively do the same for both groups.
- 5 Finally each processor sorts its keys.

Pivot

- How to find a pivot?
- This is like the argument sequential quick sort, we just randomly pick one, which is good enough.
- Use a binomial trial to argue that the tree depth of a quick sort is bounded by $O(\log p)$ with high probability.

Analysis

- We focus on the number of “movements” as the cost, since basically no computation is involved.
- In each level of the exchange a processor exchanges at most $\frac{n}{p}$ keys.
- From previous argument the depth of the tree is bounded by $O(\log p)$, so the cost of exchange is $O(\frac{n}{p} \log p)$.
- Finally each processor still needs to sort its key with $O(\frac{n}{p} \log \frac{n}{p})$ time.
- The final complexity is $O(\frac{n}{p}(\log p + \log \frac{n}{p})) = O(\frac{n}{p} \log n)$.

Discussion

- What is the theoretical speedup of this algorithm?
- Find out the definition of hypercube and why is this algorithm suitable for hypercube.

Matrix Multiplication

- Multiple two $n \times n$ matrices A , and B and place the result into C .

$$A \times B = C \quad (19)$$

- We assume that the matrix is dense.

Sequential Matrix Multiplication

- For the interest of simplicity we use the standard $O(n^3)$ algorithm, instead of the Strassen⁴ algorithm.
- The time complexity is $O(n^3)$.

⁴http://en.wikipedia.org/wiki/Strassen_algorithm

Parallel Matrix Multiplication

- We use p processors to compute the n^3 elements in C .
- Each processor simply computes the answer and no communication among them is necessary for a shared memory implementation.

Analysis

- Each processor computes $\frac{n^2}{p}$ elements.
- Each elements takes $O(n)$ time to computes.
- The parallel time is $O(\frac{n^3}{p})$.
- The speedup is $\frac{n^3}{\frac{n^3}{p}} = p$. It seems to an embarrassingly parallel computation and nothing can be improved.

Discussion

- Why no communication among processors is necessary for a shared memory implementation of the previous algorithm during the computation stage?