Exercises for Advanced Algorithms

Solution by: Wentao Lu

Models of computation (20%)
NP-complete (55%)
Approximation algorithms (40%)
Linear programming (30%)
Computational geometry (30%)
Parallel models and algorithms (40%)

1 NP-complete

Provide a proof of NP-completeness for the two-machine scheduling problem by formulating a reduction from the knapsack problem and state its corresponding optimization problem.

- TWO-MACHINE SCHEDULING: Given the execution time $a_1, a_2 \ldots, a_n$ of n independent tasks and a deadline D, is it possible to schedule the n tasks on two identical machines in a non-preemptive fashion (meaning that once a task starts executing it must run to completion) such that all the task complete by the deadline D.
- KNAPSACK: Given n objects with volumes v_i and prices s_i, $1 \leq i \leq n$, a knapsack of volume V, and a constant $S > 0$, is there a subset $N' \subseteq \{1, \ldots, n\}$ such that $\Sigma_{i \in N'} v_i < V$ (the objects in N' can be placed in the knapsack) and $\Sigma_{i \in N'} s_i \geq S$ (their price exceeds the given limit S)?

SOLUTION:

The complete proof consists of two steps. Given any candidate solution (purported certificate) of this problem which divides the n independent tasks into two bins, we can easily compute the total execution time of tasks in each bin, and then check if both of them $\leq D$. Since the solution can be verified in linear time, by definition[1, p. 1049] it is an NP problem.

After Stephen Cook proved 3-SAT to be the first known NP-complete problem in 1971[2], the next year Richard Karp further proved that another 21 common computational problems are all NP-complete[3], one of which is the knapsack problem. To prove NP-hardness, we will show that the knapsack problem is reducible to the two-machine scheduling problem.

For every instance of the knapsack problem, construct $n + 2$ independent tasks with execution time $\{a_1, \ldots, a_{n+2}\}$ and a deadline $D = \Sigma_{i=1}^{n} a_i + 2$. Let

$$
\begin{aligned}
a_i &= v_i + s_i\,,\ 1 \leq i \leq n \\
a_{n+1} &= \Sigma_{i \in P} v_i + \Sigma_{i \in P} s_i + 2 \\
a_{n+2} &= \Sigma_{i=1}^{n} a_i - \Sigma_{i \in P} v_i - \Sigma_{i \in P} s_i + 2
\end{aligned}
$$

where P denotes the set of items placed in the knapsack. It is evident that this mapping takes polynomial time in the size of input. Now let's prove that this translation is indeed a reduction.

$\Rightarrow$ First, suppose that the knapsack problem has a true instance, which means there is indeed a subset of items $P \subseteq \{1, \ldots, n\}$ such that

$$\Sigma_{i \in P} v_i < V$$
$$\Sigma_{i \in P} s_i \geq S$$

Let

$$M_1 = \{a_i \mid i \in P\} \cup \{a_{n+2}\}$$
$$M_2 = \{a_i \mid i \notin P\} \cup \{a_{n+1}\}$$

Then we have

$$\text{the sum of } M_1 = \Sigma_{i \in P} v_i + \Sigma_{i \in P} s_i + \Sigma_{i=1}^{n} a_i - \Sigma_{i \in P} v_i - \Sigma_{i \in P} s_i + 2 = \Sigma_{i=1}^{n} a_i + 2 \quad (1)$$
$$\text{the sum of } M_2 = \Sigma_{i=1}^{n} a_i - \Sigma_{i \in P} v_i - \Sigma_{i \in P} s_i + \Sigma_{i \in P} v_i + \Sigma_{i \in P} s_i + 2 = \Sigma_{i=1}^{n} a_i + 2 \quad (2)$$

From (1) and (2), it's clear that we have $M_1 = M_2 = D$, also notice that $M_1 \cup M_2$ is the set of all tasks, therefore we have found a feasible schedule for two identical machines.

$\Leftarrow$ Conversely, suppose that the translated two-machine scheduling problem has a true instance, so the set of $n + 2$ independent tasks can be divided into two subsets, which can be scheduled on two machines M_1 and M_2, respectively, by the deadline $D = \Sigma_{i=1}^{n} a_i + 2$.

Since execution time must be nonnegative, and notice that $a_{n+1} + a_{n+2} = \Sigma_{i=1}^{n} a_i + 4 > D$, this means a_{n+1} and a_{n+2} must not be executed on the same machine. Without loss of generality, assume that a_{n+2} is scheduled on M_2, then the execution time of the rest of the tasks scheduled on M_2 must sum up to

$$D - a_{n+2} = \Sigma_{i=1}^{n} a_i + 2 - a_{n+2} = \Sigma_{i \in P} v_i + \Sigma_{i \in P} s_i$$

As per our reduction assumption, it imples that this subset $P \subseteq \{1, \ldots, n\}$ has the property

$$\Sigma_{i \in P} v_i < V$$
$$\Sigma_{i \in P} s_i \geq S$$

Thus, it makes a true instance of the knapsack problem as well.

Now we have KNAPSACK $\leq_p$ TWO-MACHINE SCHEDULING, which implies that the two-machine scheduling problem is at least as hard as the knapsack. This completes our proof that two-machine scheduling is also NP-complete.

The optimization variant of this is the so-called *makespan* problem, in our case with two machines denoted by $P2||C_{max}$[4], which can be stated as: Given the execution time $a_1, a_2 \ldots, a_n$ of n independent tasks, find a way to schedule the n tasks on two identical parallel machines in a non-preemptive fashion, so as to minimize the maximum task completion time. In other words, we would like to do our best to equally assign the n tasks to both machines, such that the total execution time on the machine that finishes last is minimized. This variant is NP-hard.[3]

2 Approximate Maximum Clique

Let $G = (V, E)$ be an undirected graph. For integer $k \geq 1$ define $G^{(k)} = (V^{(k)}, E^{(k)})$ such that $V^{(k)} = \{(v_1, v_2, \ldots, v_k) | v_i \in V, 1 \leq i \leq k\}$ and $((v_1, v_2, \ldots, v_k), (w_1, w_2, \ldots, w_k)) \in E^{(k)}$ iff either $(v_i, w_i) \in E$ or $v_i = w_i$ for all $1 \leq i \leq k$.

1. Prove $|C^{(k)}| = |C|^k$, where $C^{(k)}$ and C are the maximum cliques of $G^{(k)}$ and G, respectively.

2. Argue that the existence of an approximation algorithm for finding the maximum clique with a constant approximation ratio implies the existence of a polynomial-time approximation scheme for finding the maximum clique.

SOLUTION: .....

1. Let $V' = \{(v_1, v_2, \ldots, v_k) | v_i \in C, 1 \leq i \leq k\} \subseteq V^{(k)}$, where C is the maximum clique of G, so every pair of vertices (v_i, v_j) is adjacent, clearly V' is a clique of $G^{(k)}$. Since each element v_i in the k-tuple $(v_1, v_2, \ldots, v_k)$ comes from C, the size of V' is $|C|^k$. Therefore, the size of the maximum clique of $G^{(k)}$ is at least $|C|^k$. We will prove by induction that this is indeed the size of the maximum clique.

   In the base case, when $k = 1$, $G^{(1)} = G$, we have $|C^{(1)}| = |C|^1$. Now suppose that $|C^{(k)}| = |C|^k$ is true for $1 \leq i \leq k$. Let $C^{(k+1)} = \{(p_i, v_1, v_2, \ldots, v_k)\}$ where p_i is some vertex in G.

   Since the adjacency condition of $(v_1, v_2, \ldots, v_k)$ and $(w_1, w_2, \ldots, w_k)$ only requires the adjacency of v_i and w_i, the choice of vertices at index i is independent of vertices at any other index. Hence, if we remove the prefix vertex p_i from every tuple in $C^{(k+1)}$, the remaining k-tuples must form a maximum clique of $G^{(k)}$, otherwise we can always do better. By induction hypothesis, it has size $|C|^k$.

   In order to make $C^{(k+1)}$ a clique of $G^{(k+1)}$, we must choose a set of prefixes $\{p_i\} \subseteq V$ such that every pair of vertices (p_i, p_j) is connected by an edge. Moreover, to make it a maximum clique, we also want to maximize the number of prefixes we choose. In other words, we are trying to find a maximum set of mutually adjacent vertices from G and prepend them to every k-tuple in $C^{(k)}$, so we should choose the set of prefix vertices $\{p_i\} = C$. Therefore, the size of $C^{(k+1)}$ is $|C|^k \times |C| = |C|^{k+1}$.

2. Given a c-approximation algorithm for finding the maximum clique, we can apply it on $G^{(k)}$ and obtain a clique of size n, while the true maximum clique of $G^{(k)}$ has size $|C|^k$.

   By definition of an approximation ratio, $|C|^k / n \leq c$, so we have $|C| / n^{1/k} \leq c^{1/k}$. For any $\varepsilon > 0$, we can choose a positive integer $k > \log_{1+\varepsilon} c$, then $|C| / n \leq |C| / n^{1/k} \leq c^{1/k} < 1 + \varepsilon$. Since it is polynomial in the size of input, so we have a polynomial-time approximation scheme for finding the maximum clique.

3 Linear Inequality Feasibility

Given a set of m linear inequalities over n variables, the linear inequality feasibility problem asks whether there exists a set of the instances that satisfies all the linear inequalities simultaneously.

1. Prove that we can use an algorithm for linear programming to solve linear inequality feasibility problems. The number of variables and constraints used in the linear programming problem must be polynomial in n and m.

2. Prove that we can use an algorithm for the linear inequality feasibility problem to solve linear programming problems. The number of variables and linear inequalities must be polynomial in the number of variables and constraints of the linear program.

SOLUTION:

1. Given a set of linear inequalities, we can simply formulate the equality feasibility problem as a linear program. First, we can set the objective function to be a constant such as 0 because there's nothing to maximize or minimize in this case. Next, we need to transform those linear inequalities into their slack form, and then apply the simplex algorithm. If the linear program returns a feasible solution, it implies that all the linear inequalities can be satisfied at the same time. Otherwise, those linear inequalities are not feasible. The variables and constraints in the linear program are exactly the same as in the equality feasibility problem so they are polynomial in n and m.

2. Given a linear program in standard form, we want to maximize the objective function $\Sigma_{j=1}^{n} c_j x_j$ with n variables, subject to m constraints $\Sigma_{j=1}^{n} a_{ij} x_j \leq b_i, 1 \leq i \leq m$. Suppose we have an algorithm for the linear inequality feasibility problem, then we can apply it on the m constraints of the linear program. If the algorithm fails to find a solution for the linear inequality feasibility problem, it means that those inequalities can not be satisfied, so the linear program should return infeasible. If the algorithm does find an assignment of the n variables that satisfies all the linear inequalities, then the linear program is also feasible.

   In order to solve the linear program and maximize the objective function, in each step we first calculate the value of our objective function using the assignment values returned by that algorithm, then we update the linear program by introducing a new constraint. For example, if the algorithm finds an assignment $\overline{x}$ which gives an objective function value of $\Sigma_{j=1}^{n} c_j x_j = k$, we will add another constraint $\Sigma_{j=1}^{n} c_j x_j > k$ to the original linear program. As long as this updated linear program is feasible, it must be equivalent to the original linear program because both the objective function and all other constraints are not affected. If we iteratively run this step, we are getting closer to the optimal solution. We will repeat this step until the algorithm detects that the updated constraints are not feasible anymore, in which case we stop and return the last calculated objective function value as the optimal solution.

   Whenever the algorithm returns a feasible assignment so that all constraints are satisfied, it represents a point that's somewhere in the simplex. The basic idea is to update that point until it reaches a local maximum, which is also a global maximum because the simplex is convex. Since we are only adding 1 more constraint in each step, we can optimistically assume that the number of linear inequalities is polynomial in the number of constraints. If that's not true, we can modify the constraint to be $\Sigma_{j=1}^{n} c_j x_j > k + \alpha$, where α is a reasonable step size to choose in different trials. Unless the original linear program has an unbounded solution, we will eventually converge to an optimal solution.

4 Ghostbusters and Ghosts

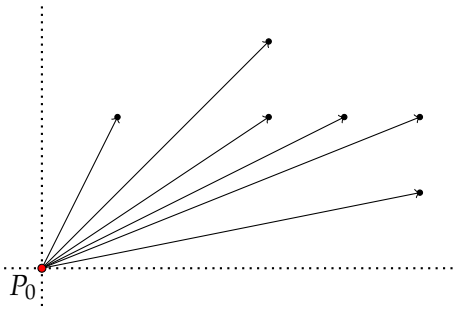
A group of n ghostbusters is battling n ghosts. Each buster carries a proton pack, which shoots a stream at a ghost, eradicating it. A stream goes in a straight line and terminates when it hits the ghost. The ghostbusters decide upon the following strategy: They will pair off with the ghosts, forming n ghostbuster-ghost pairs, and then simultaneously each ghostbuster will shoot a stream at his chosen ghost. As we all know, it is *very* dangerous to let streams cross, and so the busters

must choose pairings for which no streams will cross. Assume that the position of each buster and each ghost is a fixed point in the plane and that no three positions are colinear.

1. Argue that there exists a line passing through one Ghostbuster and one ghost such that the number of Ghostbusters on one side of the line equals the number of ghosts on the same side. Describe how to find such a line in $O(n \log n)$ time.

2. Give an $O(n^2 \log n)$-time algorithm to pair Ghostbusters with ghosts such that no streams cross.

SOLUTION: .....



Example 1: Graham Scan

1. First we pick the leftmost bottom point P_0 as the origin. For the remaining points, we can easily sort them by polar angle in counterclockwise order. Concretely, the cross product $(P_1 - P_0) \times (P_2 - P_0)$ gives us a comparison relationship between any two points P_1 and P_2, which can be applied to any comparison-based sorting algorithm such as merge sort.

   Given the sorted points by polar angle with respect to P_0, we then visit them one by one in this order, and count the number of ghostbusters and ghosts along the way. We would stop at a point P_k when the following two conditions are met:

   - Either P_0 is a ghostbuster and P_k is a ghost, or P_0 is a ghost and P_k is a ghostbuster.
   - The number of ghostbusters and ghosts visited so far are equal (including P_0 and P_k).

   It follows that the line passing through P_0 and P_k is the solution. Since the number of ghostbusters and ghosts are equal, we are guaranteed to find such a line. It takes $O(n \log n)$ to sort and $O(n)$ to scan, so the running time is $O(n \log n)$ in total.

2. Now we have paired one ghostbuster with one ghost in $O(n \log n)$ time, applying this algorithm recursively on both sides of this line will pair all the ghostbusters with ghosts.

   That is, for the subset of points on each side, we choose the origin with the minimum coordinate, sort the remaining points on this side and then scan through them until we find a new line. During each recursion step, work done on one side of the line is independent from the other, so that no lines will ever cross. We need to find n lines, so this divide-and-conquer approach runs in $O(n^2 \log n)$ time.

5 Divide and Conquer Prefix Computations

We use again divide and conquer method to perform prefix computation (with $\circ$ as binary operator) on a sequence S of size n. This time the running time will be $O(\log \log n)$ time using

$O(n/\log\log n)$ processors. This algorithm would be optimal, faster than the one mentioned in the book, but also using more processors. To obtain such an algorithm we proceed as follows:

a Divide the sequence S into $n^{1/2}$ subsequences of size $n^{1/2}$ each. Perform prefix computation on each sequence recursively using $n^{1/2}$ processors. Let the result of the computation on the i-th sequence be $s(i,1), s(i,2), \ldots, s(i,n^{1/2})$.

b Perform a prefix computation on the sequence $s(1,n^{1/2}), s(2,n^{1/2}), \ldots, s(n^{1/2}-1,n^{1/2})$ using n processors. Let the result of this computation be $s'(1,n^{1/2}), s'(2,n^{1/2}), \ldots, s'(n^{1/2}-1,n^{1/2})$.

c For all $1 \leq i < n^{1/2}$ use the binary operator $\circ$ to combine $s'(i,n^{1/2})$ with all the elements of $s(i+1,1), s(i+1,2), \ldots, s(i+1,n^{1/2})$ using $n^{1/2}$ processors per subsequence.

1. Show that the algorithm described above uses n processors to run in $O(\log\log n)$ time.

2. Show how the number of processors can be reduced to $O(n/\log\log n)$ while maintaining the $O(\log\log n)$ running time, thus achieving the optimal cost $O(n)$.

3. Discuss the implications of the result obtained in Question 2 on related computations such as array packing.

SOLUTION: .....

1. In step 1 we recursively perform prefix computation on each subsequence of size $n^{1/2}$ using $n^{1/2}$ processors. Suppose our recursive algorithm on the entire sequence S has running time $t(n)$, then the recursion call on each subsequence takes $t(n^{1/2})$. Since this is done in parallel, this step takes $t(n^{1/2})$.

   In step 2, the sequence $s(1,n^{1/2}), s(2,n^{1/2}), \ldots, s(n^{1/2}-1,n^{1/2})$ contains every last element of a subsequence except the first one, so there are only $n^{1/2}-1$ elements, but we have n processors available. We can take advantage of all of them by letting $n^{1/2}$ processors work on each element simultaneously, that is, for some element $s(i,n^{1/2})$, we use $n^{1/2}$ processors in parallel to perform binary operations on $s(i,n^{1/2})$ with all other elements $s(j,n^{1/2})$, where $j \neq i, 1 \leq j \leq n^{1/2}-1$. Since we have n processors so that all elements can update in parallel, this step takes $O(1)$. However, this does not work for all binary operations, in fact, the binary operation must be both associative and commutative, such as addition, multiplication and union, but not concatenation because concurrent writes would mess up the order. Besides, this speed up is possible only with the Combining CRCW PRAM model such that concurrent reads and writes can be effectively reconciled.

   In step 3, for each subsequence we use $n^{1/2}$ processors to combine its elements with the last element in the preceding subsequence. Each element can be mapped to a processor, so this single operation is done in parallel, this step takes $O(1)$ time.

   Now combining these results we have $t(n) = t(n^{1/2}) + O(1)$. To derive the running time from this equation, we substitute n with e^m, so $t(e^m) = t(e^{m/2}) + O(1)$. Let $s(m) = t(e^m)$, then $s(m) = s(m/2) + O(1)$. According to the Master's Theorem, this implies that

$$s(m) = O(\log m) \quad \Rightarrow \quad t(n) = t(e^m) = s(m) = O(\log m) = O(\log\log n)$$

2. If we want to reduce the number of processors to $O(n/\log\log n)$, the previous partition will not work, we need a new partition. Besides, we want to follow the previous three steps, but need to make some modifications.

In concrete, we will divide S into $n/\log\log n$ subsequences or groups, each group has size $\log\log n$. Since the number of groups is equal to the number of processors, each group can only have one processor, so in step 1, prefix computation must be executed sequentially within each group, while all groups do this in parallel. Hence, this step takes $O(\log\log n)$.

In step 2, the sequence will be $s(1, \log\log n), s(2, \log\log n), \ldots, s(n/\log\log n - 1, \log\log n)$ because we only have $n/\log\log n$ groups, processors and last elements. To perform prefix computation on this sequence, we will adopt the algorithm in Question 1. As we have shown in Question 1, that algorithm runs in $O(\log\log n)$ time on n elements using n processors, so here with $n/\log\log n$ elements and processors, it will take $O(\log\log(n/\log\log n))$ time to run. By removing the trivial terms, we have

$$O(\log\log(n/\log\log n)) = O(\log(\log n - \log(\log\log n))) = O(\log\log n)$$

In step 3, for the same reason as in step 1, the combining operation has to be executed sequentially for each group due to the lack of processors. As a result, each group takes $O(\log\log n)$ to run, while different groups run in parallel with each other. This step takes $O(\log\log n)$.

In summation, the total running time is $3 \times O(\log\log n) = O(\log\log n)$ which is identical to the previous one, but the cost is optimal as we are now using less than n processors.

3. The result obtained in Question 2 has several implications.

   We have seen that it is possible to devise an optimal parallel algorithm that takes a suboptimal algorithm as an intermediate step. The divide and conquer approach allows us to divide a problem into many subproblems which can be solved in parallel, and the way we partition the data can sometimes make a difference. In addition, different PRAM models may lead to different optimal algorithms to solve the same problem, we are free to choose among them depending on the scenario. This gives us a lot more flexibility from a practical perspective since the number of processors and hardware is often fixed in real world. For example, compared to the other optimal algorithm discussed in the class notes, this one is also optimal but much faster, so it can be very useful if we are strict on the running time and have sufficient processors that support CRCW PRAM models. It has also proved the possibility of applications being highly scalable based on commutative binary operations such as prefix sum. For example, the array packing problem uses addition as the underlying binary operation, so that as long as the hardware requirement is satisfied, it can easily scale up to boost performance, especially when the input array size is huge.

6 Descending in a Hypercube

Let $N = 2^g$ data be stored in the processors of a hypercube, one value per processor (so that the value x_i is stored in $P_i, 0 \leq i < N$). A technique with wide applicability to hypercube algorithms is called DESCEND and consists of g iterations. During the j-th iteration a basic binary operation OPERATION(P_i, P_l) is performed on data in cores whose indices differ by 2^j. Here is the pseudocode for this process, with $i_{g-1}i_{g-2}\ldots i_1 i_0$ the binary representation of index i:

Algorithm 1 DESCEND

```
1: for j ← g − 1 to 0 do                                          ▷ iterate over each bit
2:     for i ← 0 to 2^g − 1 do                                    ▷ iterate over each core
3:         if i_j = 0 then OPERATION(P_i, P_{i+2^j})   } in parallel   ▷ operate on adjacent cores
```

If OPERATION requires constant time then DESCEND runs in $O(\log N)$ time. The dual to DESCEND is ASCEND, in which j in the outer loop varies from 0 to $g-1$.

1. Show how DESCEND can be used to obtain an algorithm that computes the sum of all the values x_i and places the result in P_0.

2. Use either ASCEND or DESCEND to broadcast the datum held by some processor P_k to all the other processors.

3. Suggest other problems that can be solved using these paradigms.

SOLUTION: .....

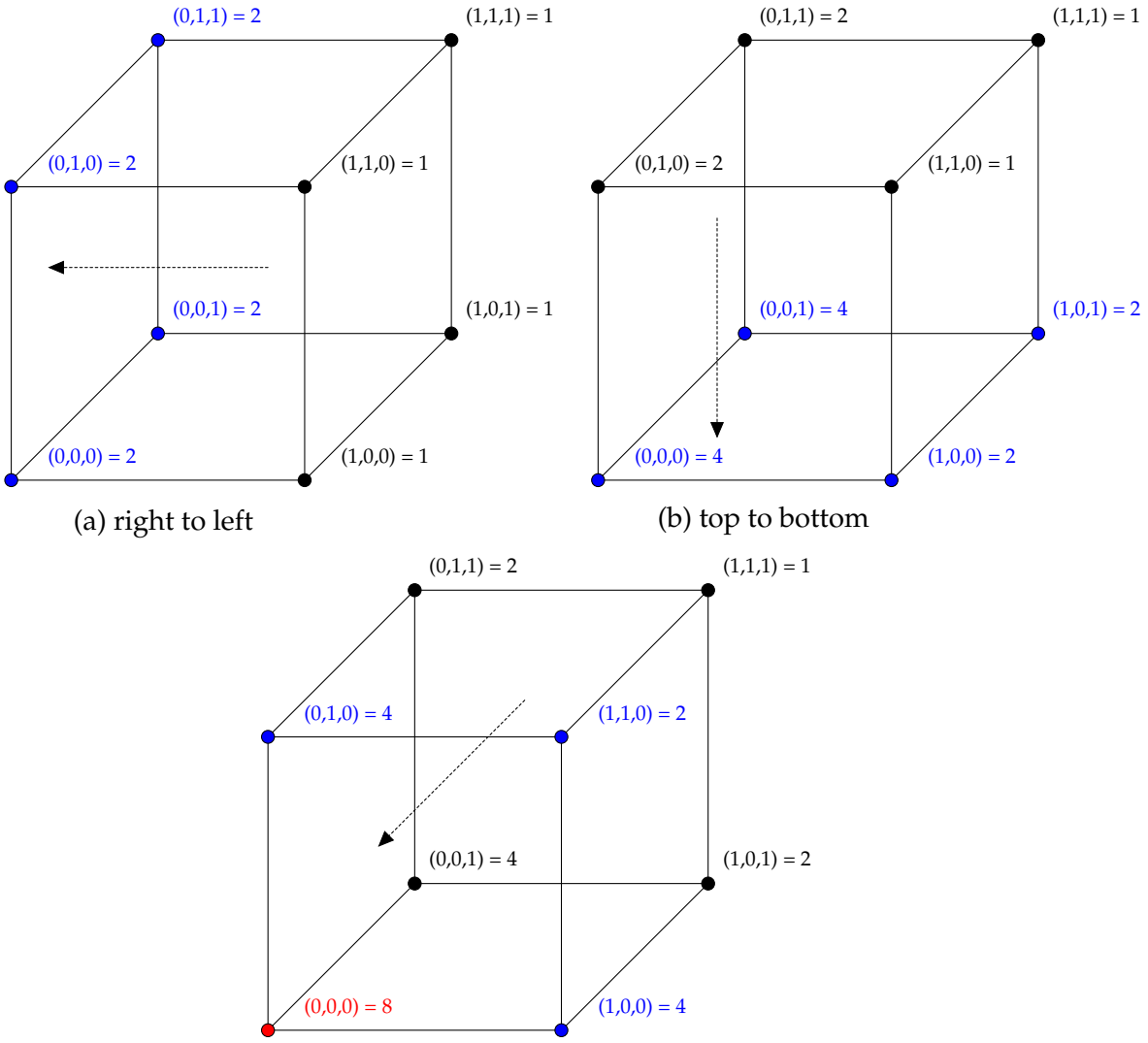
If two integers differ by 2^k, then in their binary representation, all bits must be the same except the k-th bit where they differ by 1. That is, in the j-th iteration of the DESCEND algorithm, the binary operation is applied on any pair of processors whose indices differ only at the j-th bit. A bit can be either 0 or 1, only 2 possible values. Therefore, in each iteration, all processors are divided into halves by some bit, one half will exchange data with the other half via the binary operation. After g iterations, we would have divided the set of processors bit by bit in every possible way, so that every processor has talked to all the other processors exactly once, either directly or indirectly.

1. Since the addition operation is commutative, it doesn't matter if we loop from the highest bit or the lowest bit, so we don't mind whether the DESCEND or ASCEND algorithm is being used. In order to place the final result in P_0 whose index is a bunch of zeros in binary format, during the j-th step we must store the sum of P_i and P_{i+2^j} into P_i because $i_j = 0$. Then after g iterations, the value in each processor will be added exactly once to P_0. So we define

   OPERATION(P_i, P_{i+2^j}) to be $\boxed{P_i = P_i + P_{i+2^j}}$

   As an example, we can graphically illustrate how it works in a 3 dimensional hypercube. In the figure below, processors $P_0, P_1, \ldots, P_7$ correspond to the 8 corner vertices of the cube, which are represented in their binary format as $(0,0,0), (0,0,1), \ldots, (1,1,1)$. For simplicity, suppose that initially each processor has value $x_i = 1$, so we expect P_0 to be 8 in the end.

   In the first iteration the algorithm looks at the highest bit, which is 0 for vertices on the left side and 1 for vertices on the right side. So as to perform the binary operation, values on the right side are added to those on the left. Vertices whose values have been updated are colored blue. Likewise, the second round looks at the middle bit, so values on the upper side are added to those on the lower side, and so forth. When the algorithm finishes, $P_0(0,0,0)$ has been involved in all iterations and colored blue for three times, hence it has the sum of all values and we color it red. In the end, we managed to compute the sum of 8 numbers in merely 3 steps.



(a) right to left

(b) top to bottom

(c) back to front

Example 2: Parallel sum in a 3-hypercube

2. No matter which bit we start from, the DESCEND or ASCEND algorithm always divides the processors by a different dimension in each round, then the binary operation is performed on pairs of processors, one from each group. Suppose that we use the ASCEND algorithm that loops from the 0-th bit, further assume that P_k has the datum set $\{x\}$ while other processors have no data ($\emptyset$) initially. In order to broadcast, we can simply define

   OPERATION(P_i, P_{i+2^j}) to be $\boxed{P_i \bigcup P_{i+2^j}}$

   In the first round, P_k sends $\{x\}$ to its counterpart $\bar{P}_k$ in the other group via the binary operation, this implies that P_k and $\bar{P}_k$ can only differ by 1 at the 0-th bit. On the other hand, the rest of the processors are empty, so the binary operation has no effect on them. When this round finishes, now 2 processors know about $\{x\}$. In the second round, we have a new partition of processors. Since P_k and $\bar{P}_k$ differs only at the 0-th bit but now we have divided the processors by another bit, they cannot be counterparts this time or in any future rounds. Hence, they will both have a

new counterpart who can receive the datum $\{x\}$. When this round finishes, now 4 processors know about $\{x\}$, and so forth...

Given that any two processors can be counterparts in only one round, we will always have new pairs of counterparts in each round. As a result, after the j-th iteration, 2^j processors will know about $\{x\}$, and finally the datum will be propagated across the entire interconnection network.

3. The DESCEND and ASCEND paradigm can be a powerful tool in many scenarios.

   If the binary operation is set intersection and each processor has a set, we can find the common values and remove duplicates. If the binary operation is set union, we can also find all distinct values or aggregate the data. For other binary operations such as the max()/min() function or multiplication, we can compute the maximum, minimum and product of numbers in an array or other data structures. It is also possible to use logical gates in parallel, where each processor handles a condition that is either true or false. For instance, we can check if the constraints of a linear program are violated, or if an interpretation of the 3-SAT formula is satisfiable.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009, ISBN: 978-0-262-03384-8. [Online]. Available: http://mitpress.mit.edu/books/introduction-algorithms.

[2] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, 1971, pp. 151–158. DOI: 10.1145/800157.805047. [Online]. Available: https://doi.org/10.1145/800157.805047.

[3] R. M. Karp, "Reducibility among combinatorial problems," in *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, 1972, pp. 85–103. DOI: 10.1007/978-1-4684-2001-2_9. [Online]. Available: https://doi.org/10.1007/978-1-4684-2001-2_9.

[4] P. Schuurman and G. J. Woeginger, "Approximation schemes - a tutorial," in *Lectures on Scheduling*, 2000.