CSE Qualifying Exam: High-Performance Computing

Spring 2020

Instructions

- Please answer three of the following four questions. All questions are graded on a scale of 10. If you answer all four, all answers will be graded and the three lowest scores will be used in computing your total.
- Please write clearly and concisely, explain your reasoning, and show all work. Points will be awarded for clarity as well as correctness.
- Unless otherwise specified, assume a distributed memory model of computation where a message of m words can be sent in $\mathcal{O}(\tau + \mu m)$ time, where τ denotes the latency and μ denotes the per word transfer time. You may assume each processor can send and receive one message within the same parallel communication step.

Problem 1

Suppose we have p interconnected processors, each with its own memory. Consider n particles where the position of each particle in 3-D space is known. Space is partitioned into p cells. The positions of the particles lying in the kth cell are stored only on processor k. If a pair of particles i and j are separated by a distance d less than a given distance r, we must compute the equal-and-opposite force on particle i and on particle j. This force depends on d and is used to update the position of particles i and j.

(a) Assuming a uniform random distribution of particles in a bounded 3-D domain, derive an asymptotic lower bound on the interprocess communication volume required to update the positions of all n particles.

(b) Give an algorithm that attains this lower bound.

Problem 2

The Hilbert space filling curve is a way to convert coordinates (x, y) into an index h such that two points with consecutive indices are always near each other in the plane.

We can write a Hilbert index as an integer in base 4, $h = h_1 h_2 \cdots h_n$, where each $h_i \in \{0, 1, 2, 3\}$.

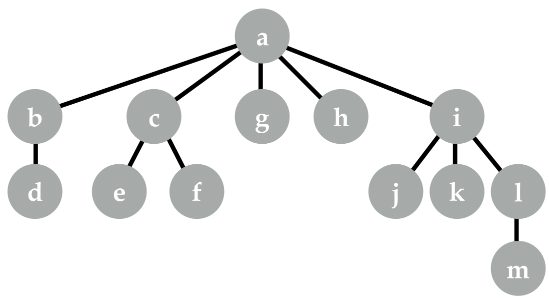
Decoding a Hilbert index can be done recursively: we start with $x_0 = y_0 = 0$, and we have four different update rules:

- if $h_i = 0$, $x_i \leftarrow y_{i-1}/2$, $y_i \leftarrow x_{i-1}/2$;
- if $h_i = 1$, $x_i \leftarrow (1 + x_{i-1})/2$, $y_i \leftarrow y_{i-1}/2$;
- if $h_i = 2$, $x_i \leftarrow (1 + x_{i-1})/2$, $y_i \leftarrow (1 + y_{i-1})/2$;
- if $h_i = 3$, $x_i \leftarrow (1 - y_{i-1})/2$, $y_i \leftarrow (2 - x_{i-1})/2$.

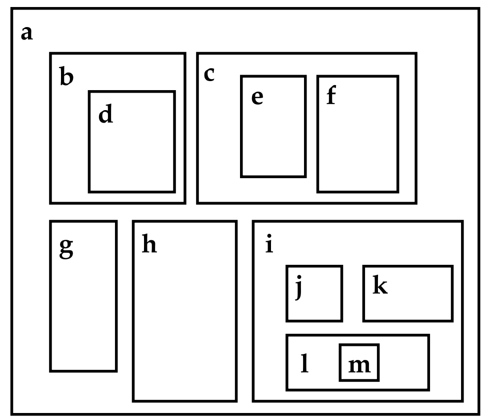
Write a parallel algorithm for decoding a Hilbert index. Your algorithm should run in $O(\log n)$ time.

Problem 3

Parallel HTML rendering. When a web browser downloads a web page, it receives it in HTML format. Let's assume a simplified form of HTML: the downloaded document is represented as a tree of page elements, as shown in Figure 1a. Each vertex is a *page element*; and page elements may be *nested*. If v is an element, let $P[v]$ be its *parent* and let $C[v]$ be the set of its children. For instance, in Figure 1a, $C[a] = \{b, c, g, h, i\}$ and $P[d] = b$.



(a) Raw HTML, viewed as a tree of "page elements"



(b) Illustration of a rendered HTML page

Figure 1: Parallel HTML rendering problem

Given such a tree, the browser needs to *render* it, meaning to lay it out physically on the output device, such as a screen or printed page. Figure 1b shows a hypothetical rendering. When fully rendered, every element v has known width and height and a known absolute position on the output device. Let's use S_v as a shorthand for this "state" of v. The problem is that the input tree (Figure 1a) initially holds only the nesting relationships and the content of each element; it does *not* have any size or position information, other than the nesting structure. In this scenario, we say $S_v = \emptyset$, to denote the initially unknown state.

However, suppose you have two special operations that can help resolve S_v.

- $S_v = \textbf{place}(S_a, S_b)$: Let a and b be any pair of direct siblings, that is $P[a] = P[b] = v$, with known dimensions in S_a and S_b. Then this function will decide how to place these together (e.g., next to each other, on top of each other, or some other arrangement) and returns a new combined state, S_v, that captures this arrangement and their *relative* positions. You can think of the combined state as a kind of "virtual child" of v, so that one may compose placements as $\textbf{place}(\textbf{place}(S, T), U)$ = $\textbf{place}(\textbf{place}(S, U), T)$.
- $S_v''' = \textbf{merge}(S_v', S_v'')$: Given two partial renderings of v, as might be produced by $\textbf{place}(\cdot, \cdot)$, this operation reconciles them into a single rendering. That is, S_v' might represent a partial rendering with some of the children of v while S_v'' does so for a different set of v's children. This operation will combine them. Similar to the above, you may further assume that $\textbf{merge}(S', \textbf{merge}(S'', S'''))$ = $\textbf{merge}(\textbf{merge}(S', S''), S''')$.

Lastly, assume that for any leaf w (i.e., $C[w] = \emptyset$), the size is easy to compute by a call to $\textbf{place}(\emptyset, w)$ = $\textbf{place}(w, \emptyset)$. You may also assume that the above operations cost $\mathcal{O}(1)$ time each. If you need more assumptions, state them clearly.

Please answer the following questions.

a. (70%) Give an efficient parallel algorithm to render the page, that is, to fully resolve S_v for all v. "Efficient" in this case means with a total work that is as asymptotically close as you can manage to $\mathcal{O}(n)$ and a span or depth that is polylogarithmic in n. Analyze your algorithm.

b. (30%) Critique the assumptions of this problem. That is, which ones pose the "biggest threat" to the efficiency or correctness of your approach, and why?

Problem 4

Scheduling.

a. *Scheduling for a Single Processor:* Consider n jobs $\{j_1, j_2, \ldots, j_n\}$ that will arrive for execution on a single processor, one after each other. Let their arrival times be $\{a_1, a_2, \ldots, a_n\}$, respectively, with $a_i \leq a_{i+1}$. Also let $\{t_1, t_2, \ldots, t_n\}$ be the respective execution times on that processor. If the processor is free when job j_i arrives, job starts without any wait and executes for t_i unit and it exits. Otherwise, job waits in the queue for its turn for execution (in the order it was received). Assume a_i's and t_i's are distributed evenly across p processors. Design and efficient parallel algorithm to determine the waiting time incurred by each job. (60%)

b. *Scheduling for Multiple Processors:* Now consider, we have m identical processors that jobs can be executed. Discuss the changes you would do to your algorithm; how they effect the execution time of your algorithm. (40%)