1.3 Simulation

We have considered several models of computation. We will see now how the simplest of them – Turing Machine – can simulate all others: these powerful machines can compute no more functions than TM.

Church-Turing Thesis is a generalization of this conclusion: TMs can compute every function computable in any thinkable physical model of computation. This is not a math theorem because the notion of model is not formally specified. But the long history of studying ways to design real and ideal computing devices makes it very convincing. Moreover, this Thesis has a stronger ***Polynomial Time*** version which bounds the volume of computation required by that TM simulation by a polynomial of the volume used by the other models. Both forms of the Thesis play a significant role in foundations of Computer Science.

PKM Simulation of PPM. For convenience, we assume all PPM nodes have pointers to root. PPM configuration is represented in PKM with extra colors l, r, u used in a u-colored binary tree added to each node X so that all (unlimited in number) PPM pointers to X are reconnected to its leaves, and inverses, colored l, r, added to all pointers. The number of pointers increases at most 4 times. To simulate PPM, X gets a binary ***name*** formed by the l, r colors on its path through the root tree, and broadcasts it down its own tree. For pulling stage X extends its tree to double depth and merges (with combined colors) its own pointers to nodes with identical names. Then X re-colors its pointers as PPM program requires and rebalances its tree. This simulation of a PPM step takes polylogarithmic parallel time.

TM Simulation of PKM. We assume the PKM keeps a constant degree and a roughly balanced root tree (to yield short node names as described above). TM tape reflects its configuration as the list of all pointers sorted by source name, then by color. The TM's transition table reflects the PKM program. To simulate PKM's pulling stage TM creates a copy of each pointer and sorts copies by their sinks. Now each pointer, located at source, has its copy near its sink. So both components of 2-pointer paths are nearby: the special double-colored pointers can be created and moved to their sources by resorting on the source names. The re-coloring stage is straightforward, as all relevant pointers having the same source are located together. Once the root has no active pointers, the Turing machine stops and its tape represents the PKM output. If a PPM computes a function $f(x)$ in $t(x)$ steps, using $s(x)$ nodes, the simulating TM uses space $S = O(s \log s)$, ($O(\log s)$ bits for each of $O(s)$ pointers) and time $T = O(S^2)t$, as TM sorting takes quadratic time.

Squaring matters ! TM cannot outperform Bubble Sort. Is its quadratic overhead a big deal? In a short time all silicon gates on your PC run, say, $X{=}10^{23}{\sim}2^{2^{6.25}}$ clock cycles combined. Silicon parameters double almost annually. Decades may bring micron-thin things that can sail sunlight in space in clouds of great computing and physical (light beam) power. Centuries may turn them into a Dyson Sphere enveloping the solar system. Still, the power of such an ultimate computer is limited by the number of photons the Sun emits per second: $Y{\sim}2^{2^{7.25}}{=}X^2$. Giga-years may turn much of the known universe into a computer, but its might is still limited by its total entropy $2^{2^{8.25}}{=}Y^2$.

Faster PPM Simulations. Parallel Bubble-Sort on CA or Merge-Sort on sequential FCM take nearly linear time. Parallel FCM can do much better [Ofman 65]. It represents and updates pointer graphs as the above TM. All steps are straightforward to do locally in parallel polylog time except sorting of pointers. We need to create a fixed connection sorting network. Sophisticated networks sort arbitrary arrays of n integers in $O(\log n)$ parallel steps. We need only a simpler polylog method. Merge-Sort splits an array of two or more entries in two halves and sorts each recursively. Batcher-Merge combines two sorted lists in $O(\log n)$ steps.

Batcher Merge. A ***bitonic cycle*** is the combination of two sorted arrays (one may be shorter), connected by max-to-max and min-to-min entries. Entries in a contiguous half (***high-half***) of the cycle are $\geq$ than all entries in the other (***low***) half. Each half (with its ends connected) forms a bitonic cycle itself.

Shuffle Exchange graph links nodes in a 2^k-nodes array to their ***flips*** and ***shits***. The flip flips the highest bit of a node's address; the ***shift*** cycle-shifts that bit to the end, or switches 0^k address with 1^k.

We merge-sort two sorted arrays given as a bitonic cycle on such a graph as follows.
Comparing each entry with its flip (half-a-cycle away), and switching if wrongly ordered, fits the high and low halves into respectively the first and last halves of the array. (This rotates the array, and then its left and right halves.) We do so for each half recursively (decrementing k via graph's shift edges).