CHAPTER 11

Memory-Hierarchy Tradeoffs

Although serial programming languages assume that programs are written for the RAM model, this model is rarely implemented in practice. Instead, the random-access memory is replaced with a hierarchy of memory units of increasing size, decreasing cost per bit, and increasing access time. In this chapter we study the conditions on the size and speed of these units when a CPU and a memory hierarchy simulate the RAM model. The design of memory hierarchies is a topic in operating systems.

A memory hierarchy typically contains the local registers of the CPU at the lowest level and may contain at succeeding levels a small, very fast, local random-access memory called a cache, a slower but still fast random-access memory, and a large but slow disk. The time to move data between levels in a memory hierarchy is typically a few CPU cycles at the cache level, tens of cycles at the level of a random-access memory, and hundreds of thousands of cycles at the disk level! A CPU that accesses a random-access memory on every CPU cycle may run at about a tenth of its maximum speed, and the situation can be dramatically worse if the CPU must access the disk frequently. Thus it is highly desirable to understand for a given problem how the number of data movements between levels in a hierarchy depends on the storage capacity of each memory unit in that hierarchy.

In this chapter we study tradeoffs between the number of storage locations (space) at each memory-hierarchy level and the number of data movements (I/O time) between levels. Two closely related models of memory hierarchies are used, the memory-hierarchy pebble game and the hierarchical memory model, which are extensions of those introduced in Chapter 10.

In most of this chapter it is assumed not only that the user has control over the I/O algorithm used for a problem but that the operating system does not interfere with the I/O operations requested by the user. However, we also examine I/O performance when the operating system, not the user, controls the sequence of memory accesses (Section 11.10). Competitive analysis is used in this case to evaluate two-level LRU and FIFO memory-management algorithms.

11.1 The Red-Blue Pebble Game

The red-blue pebble game models data movement between adjacent levels of a two-level memory hierarchy. We begin with this model to fix ideas and then introduce the more general memory-hierarchy game. Both games are played on a directed acyclic graph, the graph of a straight-line program. We describe the game and then give its rules.

In the red-blue game, (hot) red pebbles identify values held in a fast primary memory whereas (cold) blue pebbles identify values held in a secondary memory. The values identified with the pebbles can be words or blocks of words, such as the pages used by an operating system. Since the red-blue pebble game is used to study the number of I/O operations necessary for a problem, the number of red pebbles is assumed limited and the number of blue pebbles is assumed unlimited. Before the game starts, blue pebbles reside on all input vertices. The goal is to place a blue pebble on each output vertex, that is, to compute the values associated with these vertices and place them in long-term storage. These assumptions capture the idea that data resides initially in the most remote memory unit and the results must be deposited there.

RED-BLUE PEBBLE GAME

- (Initialization) A blue pebble can be placed on an input vertex at any time.
- (Computation Step) A red pebble can be placed on (or moved to) a vertex if all its immediate predecessors carry red pebbles.
- (Pebble Deletion) A pebble can be deleted from any vertex at any time.
- (Goal) A blue pebble must reside on each output vertex at the end of the game.
- (Input from Blue Level) A red pebble can be placed on any vertex carrying a blue pebble.
- (Output to Blue Level) A blue pebble can be placed on any vertex carrying a red pebble.

The first rule (**initialization**) models the retrieval of input data from the secondary memory. The second rule (a **computation step**) is equivalent to requiring that all the arguments on which a function depends reside in primary memory before the function can be computed. This rule also allows a pebble to move (or **slide**) to a vertex from one of its predecessors, modeling the use of a register as both the source and target of an operation. The third rule allows **pebble deletion**: if a red pebble is removed from a vertex that later needs a red pebble, it must be repebbled.

The fourth rule (the **goal**) models the placement of output data in the secondary memory at the end of a computation. The fifth rule allows data held in the secondary memory to be moved back to the primary memory (an **input operation**). The sixth rule allows a result to be copied to a secondary memory of unlimited capacity (an **output operation**). Note that a result may be in both memories at the same time.

The red-blue pebble game is a direct generalization of the pebble game of Section 10.1 (which we call the **red pebble game**), as can be seen by restricting the sixth rule to allow the placement of blue pebbles only on vertices that are output vertices of the DAG. Under this restriction the blue level cannot be used for intermediate results and the goal of the game becomes to minimize the number of times vertices are pebbled with red pebbles, since the optimal strategy pebbles each output vertex once.

A **pebbling strategy** $\mathcal{P}$ is the execution of the rules of the pebble game on the vertices of a graph. We assign a step to each placement of a pebble, ignoring steps on which pebbles are removed, and number the steps consecutively. The **space** used by a strategy $\mathcal{P}$ is defined as the maximum number of red pebbles it uses. The **I/O time**, T_2, of $\mathcal{P}$ on the graph G is the number of input and output (I/O) steps used by $\mathcal{P}$. The **computation time**, T_1, is the number of computation steps of $\mathcal{P}$ on G. Note that time in the red pebble game is the time to place red pebbles on input and internal vertices; in this chapter the former are called **I/O operations**.

Since accesses to secondary memory are assumed to require much more time than accesses to primary memory, a **minimal pebbling strategy**, $\mathcal{P}_{\min}$, performs the minimal number of I/O operations on a graph G for a given number of red pebbles and uses the smallest number of red pebbles for a given I/O time. Furthermore, such a strategy also uses the smallest number of computation steps among those meeting the other requirements. We denote by $T_1^{(2)}(S, G)$ and $T_2^{(2)}(S, G)$ the number of computation and I/O steps in a minimal pebbling of G in the red-blue pebble game with S red pebbles.

The minimum number of red pebbles needed to play the red-blue pebble game is the maximum number of predecessors of any vertex. This follows because blue pebbles can be used to hold all intermediate results. Thus, in the FFT graph of Fig. 11.1 only two red pebbles are needed, since one of them can be slid to the vertex being pebbled. However, if the minimum number of pebbles is used, many expensive I/O operations are necessary.

In Section 11.2 we generalize the red-blue pebble game to multiple levels and consider two variants of the model, one in which all levels including the highest can be used for intermediate storage, and a second in which the highest level cannot be used for intermediate storage. The second model (the **I/O-limited game**) captures aspects of the red-blue pebble game as well as the red pebble game of Chapter 10.

An important distinction between the pebble game results obtained in this chapter and those in Chapter 10 is that here lower bounds are generally derived for particular graphs, whereas in Chapter 10 they are obtained for all graphs of a problem.

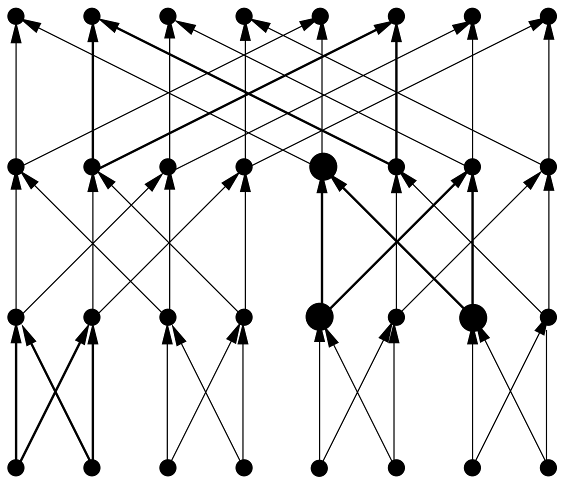


Figure 11.1 An eight-input FFT graph showing three two-input FFT subgraphs.

11.1.1 Playing the Red-Blue Pebble Game

The rules for the red-blue pebble game are illustrated by the eight-input FFT graph shown in Fig. 11.1. If $S = 3$ red pebbles are available to pebble this graph (at least $S = 4$ pebbles are needed in the one-pebble game), a pebbling strategy that keeps the number of I/O operations small is based on the pebbling of sub-FFT graphs on two inputs. Three such sub-FFT subgraphs are shown by heavy lines in Fig. 11.1, one at each level of the FFT graph. This pebbling strategy uses three red pebbles to place blue pebbles on the outputs of each of the four lowest-level sub-FFT graphs on two inputs, those whose outputs are second-level vertices of the full FFT graph. (Thus, eight blue pebbles are used.) Shown on a second-level sub-FFT graph are three red pebbles at the time when a pebble has just been placed on the first of the two outputs of this sub-FFT graph. This strategy performs two I/O operations for each vertex except for input and output vertices. A small savings is possible if, after pebbling the last sub-FFT graph at one level, we immediately pebble the last sub-FFT graph at the next level.

11.1.2 Balanced Computer Systems

A **balanced computer system** is one in which no computational unit or data channel becomes saturated before any other. The results in this chapter can be used to analyze balance. To illustrate this point, we examine a serial computer system consisting of a CPU with a random-access memory and a disk storage unit. Such a system is balanced for a particular problem if the time used for I/O is comparable to the time used for computation.

As shown in Section 11.5.2, multiplying two $n \times n$ matrices with a variant of the classical matrix multiplication algorithm requires a number of computations proportional to n^3 and a number of I/O operations proportional to $n^3/\sqrt{S}$, where S is the number of red pebbles or the capacity of the random-access memory. Let t_0 and t_1 be the times for one computation and I/O operation, respectively. Then the system is balanced when $t_0 n^3 \approx t_1 n^3/\sqrt{S}$. Let the **computational** and **I/O capacities**, C_{comp} and $C_{\text{I/O}}$, be the rates at which the CPU and disk can compute and exchange data, respectively; that is, $C_{\text{comp}} = 1/t_0$ and $C_{\text{I/O}} = 1/t_1$. Thus, balance is achieved when the following condition holds:

$$\frac{C_{\text{comp}}}{C_{\text{I/O}}} \approx \sqrt{S}$$

From this condition we see that if through technological advance the ratio $C_{\text{comp}}/C_{\text{I/O}}$ increases by a factor β, then for the system to be balanced the storage capacity of the system, S, must increase by a factor β^2.

Hennessy and Patterson [131, p. 427] observe that CPU speed is increasing between 50% and 100% per year while that of disks is increasing at a steady 7% per year. Thus, if the ratio $C_{\text{comp}}/C_{\text{I/O}}$ for our simple computer system grows by a factor of $50/7 \approx 7$ per year, then S must grow by about a factor of 49 per year to maintain balance. To the extent that matrix multiplication is typical of the type of computing to be done and that computers have two-level memories, a crisis is looming in the computer industry! Fortunately, multi-level memory hierarchies are being introduced to help avoid this crisis.

As bad as the situation is for matrix multiplication, it is much worse for the Fourier transform and sorting. For each of these problems the number of computation and I/O operations is proportional to $n \log_2 n$ and $n \log_2 n/\log_2 S$, respectively (see Section 11.5.3). Thus, balance is achieved when

$$\frac{C_{\text{comp}}}{C_{\text{I/O}}} \approx \log_2 S$$

Consequently, if $C_{\text{comp}}/C_{\text{I/O}}$ increases by a factor β, S must increase to S^β. Under the conditions given above, namely, $\beta \approx 7$, a balanced two-level memory-hierarchy system for these problems must have a storage capacity that grows from S to about S^7 every year.

11.2 The Memory-Hierarchy Pebble Game

The standard **memory-hierarchy game** (MHG) defined below generalizes the two-level red-blue game to multiple levels. The L-level MHG is played on directed acyclic graphs with p_l pebbles at level l, $1 \leq l \leq L-1$, and an unlimited number of pebbles at level L. When $L = 2$, the lower level is the red level and the higher is the blue level. The number of pebbles used at the $L-1$ lowest levels is recorded in the **resource vector** $\boldsymbol{p} = (p_1, p_2, \ldots, p_{L-1})$, where $p_j \geq 1$ for $1 \leq j \leq L-1$. The rules of the game are given below.

STANDARD MEMORY-HIERARCHY GAME

R1. (Initialization) A level-L pebble can be placed on an input vertex at any time.

R2. (Computation Step) A first-level pebble can be placed on (or moved to) a vertex if all its immediate predecessors carry first-level pebbles.

R3. (Pebble Deletion) A pebble of any level can be deleted from any vertex.

R4. (Goal) A level-L pebble must reside on each output vertex at the end of the game.

R5. (Input from Level l) For $2 \leq l \leq L$, a level-$(l-1)$ pebble can be placed on any vertex carrying a level-l pebble.

R6. (Output to Level l) For $2 \leq l \leq L$, a level-l pebble can be placed on any vertex carrying a level-$(l-1)$ pebble.

The first four rules are exactly as in the red-blue pebble game. The fifth and sixth rules generalize the fifth and sixth rules of the red-blue pebble game by identifying inputs from and outputs to level-l memory. These last two rules allow a level-l memory to serve as temporary storage for lower-level memories.

In the standard MHG, the highest-level memory can be used for storing intermediate results. An important variant of the MHG is the **I/O-limited memory-hierarchy game**, in which the highest level memory cannot be used for intermediate storage. The rules of this game are the same as in the MHG except that rule R6 is replaced by the following two rules:

I/O-LIMITED MEMORY-HIERARCHY GAME

R6. (Output to Level l) For $2 \leq l \leq L-1$, a level-l pebble can be placed on any vertex carrying a level-$(l-1)$ pebble.

R7. (I/O Limitation) Level-L pebbles can only be placed on **output** vertices carrying level-$(L-1)$ pebbles.

The sixth and seventh rules of the new game allow the placement of level-L pebbles only on output vertices. The two-level version of the I/O-limited MHG is the one-pebble game studied in Chapter 10. As mentioned earlier, we call the two-level I/O-limited MHG the **red pebble game** to distinguish it from the red-blue pebble game and the MHG. Clearly the multi-level I/O-limited MHG is a generalization of both the standard MHG and the one-pebble game.

The I/O-limited MHG models the case in which accesses to the highest level memory take so long that it should be used only for archival storage, not intermediate storage. Today disks are so much slower than the other memories in a hierarchy that the I/O-limited MHG is the appropriate model when disks are used at the highest level.

The **resource vector** $\boldsymbol{p} = (p_1, p_2, \dots, p_{L-1})$ associated with a pebbling strategy $\mathcal{P}$ specifies the number of l-level pebbles, p_l, used by $\mathcal{P}$. We say that p_l is the **space** used at level l by $\mathcal{P}$. We assume that $p_l \geq 1$ for $1 \leq l \leq L$, so that swapping between levels is possible. The **I/O time** at level l with pebbling strategy $\mathcal{P}$ and resource vector $\boldsymbol{p}$, $T_l^{(L)}(\boldsymbol{p}, G, \mathcal{P})$, $2 \leq l \leq L$, with both versions of the MHG is the number of inputs from and outputs to level l. The **computation time** with pebbling strategy $\mathcal{P}$ and resource vector $\boldsymbol{p}$, $T_1^{(L)}(\boldsymbol{p}, G, \mathcal{P})$, in the MHG is the number of times first-level pebbles are placed on vertices by $\mathcal{P}$. Since there is little risk of confusion, we use the same notation, $T_l^{(L)}(\boldsymbol{p}, G, \mathcal{P})$, in the standard and I/O-limited MHG for the number of computation and I/O steps.

The definition of a minimal MHG pebbling is similar to that for a red-blue pebbling. Given a resource vector $\boldsymbol{p}$, $\mathcal{P}_{\min}$ is a **minimal pebbling** for an L-level MHG if it minimizes the I/O time at level L, after which it minimizes the I/O time at level $L - 1$, continuing in this fashion down to level 2. Among these strategies it must also minimize the computation time. This definition of minimality is used because we assume that the time needed to move data between levels of a memory hierarchy grows rapidly enough with increasing level that it is less costly to repebble vertices at or below a given level than to perform an I/O operation at a higher level.

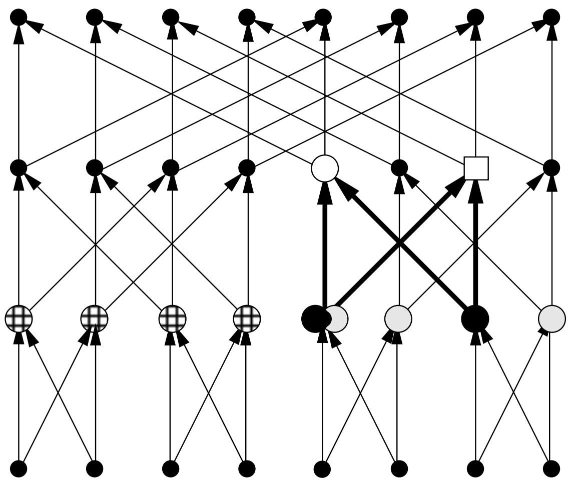


Figure 11.2 Pebbling an eight-input FFT graph in the three-level MHG.

11.2.1 Playing the MHG

Figure 11.2 shows the FFT graph on eight inputs being pebbled in a three-level MHG with resource vector $\boldsymbol{p} = (2, 4)$. Here black circles denote first-level pebbles, shaded circles denote second-level pebbles and striped circles denote third-level pebbles. Four striped, three shaded and two black pebbles reside on vertices in the second row of the FFT. One of these shaded second-level pebbles shares a vertex with a black first-level pebble, so that this black pebble can be moved to the vertex covered by the open circle without deleting all pebbles on the doubly covered vertex.

To pebble the vertex under the open square with a black pebble, we reuse the black pebble on the open circle by swapping it with a fourth shaded pebble, after which we place the black pebble on the vertex that was doubly covered and then slide it to the vertex covered by the open box. This graph can be completely pebbled with the resource vector $\boldsymbol{p} = (2, 4)$ using only four third-level pebbles, as the reader is asked to show. (See Problem 11.3.) Thus, it can also be pebbled in the four-level I/O-limited MHG using resource vector $\boldsymbol{p} = (2, 4, 4)$.

11.3 I/O-Time Relationships

The following simple relationships follow from two observations. First, each input and output vertex must receive a pebble at each level, since every input must be read from level L and every output must be written to level L. Second, at least one computation step is needed for each non-input vertex of the graph. Here we assume that every vertex in V must be pebbled to pebble the output vertices.

LEMMA 11.3.1 *Let α be the maximum in-degree of any vertex in $G = (V, E)$ and let $In(G)$ and $Out(G)$ be the sets of input and output vertices of G, respectively. Then any pebbling $\mathcal{P}$ of G with the MHG, whether standard or I/O-limited, satisfies the following conditions for $2 \le l \le L$:*

$$T_l^{(L)}(\boldsymbol{p}, G, \mathcal{P}) \ge |In(G)| + |Out(G)|$$
$$T_1^{(L)}(\boldsymbol{p}, G, \mathcal{P}) \ge |V| - |In(G)|$$

The following theorem relates the number of moves in an L-level game to the number in a two-level game and allows us to use prior results. The lower bound on the level-l I/O time is stated in terms of s_{l-1} because pebbles at levels $1, 2, \dots, l-1$ are treated collectively as red pebbles to derive a lower bound; pebbles at level l and above are treated as blue pebbles.

THEOREM 11.3.1 *Let $s_l = \sum_{j=1}^{l-1} p_j$. Then the following inequalities hold for every L-level standard MHG pebbling strategy $\mathcal{P}$ for G, where $\boldsymbol{p}$ is the resource vector used by $\mathcal{P}$ and $T_1^{(2)}(S, G)$ and $T_2^{(2)}(S, G)$ are the number of computation and I/O operations used by a minimal pebbling in the red-blue pebble game played on G with S red pebbles:*

$$T_l^{(L)}(\boldsymbol{p}, G, \mathcal{P}) \ge T_2^{(2)}(s_{l-1}, G) \quad \textit{for } 2 \le l \le L$$

Also, the following lower bound on computation time holds for all pebbling strategies $\mathcal{P}$ in the standard MHG:

$$T_1^{(L)}(\boldsymbol{p}, G, \mathcal{P}) \ge T_1^{(2)}(s_1, G),$$

In the I/O-limited case the following lower bounds apply, where α is the maximum fan-in of any vertex of G:

$$T_l^{(L)}(\boldsymbol{p}, G, \mathcal{P}) \geq T_2^{(2)}(s_{l-1}, G) \quad \text{for } 2 \leq l \leq L$$
$$T_1^{(L)}(\boldsymbol{p}, G, \mathcal{P}) \geq T_2^{(2)}(s_{L-1}, G)/\alpha$$

Proof The first set of inequalities is shown by considering the red-blue game played with $S = s_{l-1}$ red pebbles and an unlimited number of blue pebbles. The S red pebbles and $s_{L-1} - S$ blue pebbles can be classified into $L-1$ groups with p_j pebbles in the jth group, so that we can simulate the steps of an L-level MHG pebbling strategy $\mathcal{P}$. Because there are constraints on the use of pebbles in $\mathcal{P}$, this strategy uses a number of level-l I/O operations that cannot be larger than the minimum number of such I/O operations when pebbles at level $l-1$ or less are treated as red pebbles and those at higher levels are treated as blue pebbles. Thus, $T_l^{(L)}(\boldsymbol{p}, G, \mathcal{P}) \geq T_2^{(2)}(s_{l-1}, G)$. By similar reasoning it follows that $T_1^{(L)}(\boldsymbol{p}, G, \mathcal{P}) \geq T_1^{(2)}(s_1, G)$.

In the above simulation, blue pebbles simulating levels l and above cannot be used arbitrarily when the I/O-limitation is imposed. To derive lower bounds under this limitation, we classify $S = s_{L-1}$ pebbles into $L-1$ groups with p_j pebbles in the jth group and simulate in the red-blue pebble game the steps of an L-level I/O-limited MHG pebbling strategy $\mathcal{P}$. The I/O time at level l is no more than the I/O time in the two-level I/O-limited red-blue pebble game in which all S red pebbles are used at level $l-1$ or less.

Since the number of blue pebbles is unlimited, in a minimal pebbling all I/O operations consist of placing of red pebbles on blue-pebbled vertices. It follows that if T I/O operations are performed on the input vertices, then at least T placements of red pebbles on blue-pebbled vertices occur. Since at least one internal vertex must be pebbled with a red pebble in a minimal pebbling for every α input vertices that are red-pebbled, the computation time is at least T/α. Specializing this to $T = T_2^{(2)}(s_{L-1}, G)$ for the I/O-limited MHG, we have the last result. ■

It is important to note that the lower bound to $T_1^{(2)}(S, G, \mathcal{P})$ for the I/O-limited case is not stated in terms of $|V|$, because $|V|$ may not be the same for each values of S. Consider the multiplication of two $n \times n$ matrices. Every graph of the standard algorithm can be pebbled with three red pebbles, but such graphs have about $2n^3$ vertices, a number that cannot be reduced by more than a constant factor when a constant number of red pebbles is used. (See Section 11.5.2.) On the other hand, using the graph of Strassen's algorithm for this problem requires at least $\Omega(n^{.38529})$ pebbles, since it has $O(n^{2.807})$ vertices.

We close this section by giving conditions under which lower bounds for one graph can be used for another. Let a **reduction** of DAG $G_1 = (V_1, E_1)$ be a DAG $G_0 = (V_0, E_0)$, $V_0 \subseteq V_1$ and $E_0 \subseteq E_1$, obtained by deleting edges from E_1 and coalescing the non-terminal vertices on a "chain" of vertices in V_1 into the first vertex on the chain. A **chain** is a sequence $v_1, v_2, \dots, v_r$ of vertices such that, for $2 \leq i \leq r-1$, v_i is adjacent to v_{i-1} and v_{i+1} and no other vertices.

LEMMA 11.3.2 *Let G_0 be a reduction of G_1. Then for any minimal pebbling $\mathcal{P}_{\min}$ and $1 \leq l \leq L$, the following inequalities hold:*

$$T_l^{(L)}(\boldsymbol{p}, G_1, \mathcal{P}_{\min}) \geq T_l^{(L)}(\boldsymbol{p}, G_0, \mathcal{P}_{\min})$$

Proof Any minimal pebbling strategy for G_1 can be used to pebble G_0 by simulating moves on a chain with pebble placements on the vertex to which vertices on the chain are coalesced and by honoring the edge restrictions of G_1 that are removed to create G_0. Since this strategy for G_1 may not be minimal for G_0, the inequalities follow. ■

11.4 The Hong-Kung Lower-Bound Method

In this section we derive lower limits on the I/O time at each level of a memory hierarchy needed to pebble a directed acyclic graph with the MHG. These results are obtained by combining the inequalities of Theorem 11.3.1 with a lower bound on the I/O and computation time for the red-blue pebble game.

Theorem 10.4.1 provides a framework that can be used to derive lower bounds on the I/O time in the red-blue pebble game. This follows because the lower bounds of Theorem 10.4.1 are stated in terms of T_I, the number of times inputs are pebbled with S red pebbles, which is also the number of I/O operations on input vertices in the red-blue pebble game. It is important to note that the lower bounds derived using this framework apply to every straight-line program for a problem.

In some cases, for example matrix multiplication, these lower bounds are strong. However, in other cases, notably the discrete Fourier transform, they are weak. For this reason we introduce a way to derive lower bounds that applies to a particular graph of a problem. If that graph is used for the problem, stronger lower bounds can be derived with this method than with the techniques of Chapter 10. We begin by introducing the S-span of a DAG.

DEFINITION 11.4.1 *Given a DAG $G = (V, E)$, the S-**span of** G, $\rho(S, G)$, is the maximum number of vertices of G that can be pebbled with S pebbles in the red pebble game maximized over all initial placements of S red pebbles. (The initialization rule is disallowed.)*

The following is a slightly weaker but simpler version of the Hong-Kung [136] lower bound on I/O time for the two-level MHG. This proof divides computation time into consecutive intervals, just as was done for the space–time lower bounds in the proofs of Theorems 10.4.1 and 10.11.1.

THEOREM 11.4.1 *For every pebbling $\mathcal{P}$ of the DAG $G = (V, E)$ in the red-blue pebble game with S red pebbles, the I/O time used, $T_2^{(2)}(S, G, \mathcal{P})$, satisfies the following lower bound:*

$$\lceil T_2^{(2)}(S, G)/S \rceil \rho(2S, G) \geq |V| - |In(G)|$$

Proof Divide $\mathcal{P}$ into consecutive sequential sub-pebblings $\{\mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_h\}$, where each sub-pebbling has S I/O operations except possibly the last, which has no more such operations. Thus, $h = \lceil T_2^{(2)}(S, G, \mathcal{P})/S \rceil$.

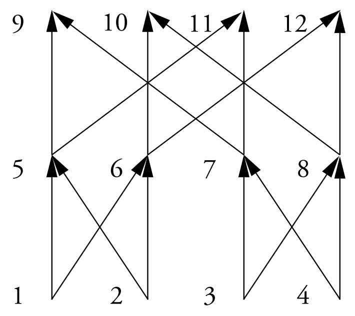
We now develop an upper bound Q to the number of vertices of G pebbled with red pebbles in any sub-pebbling $\mathcal{P}_j$. This number multiplied by the number h of sub-pebblings is an upper bound to the number of vertices other than inputs, $|V| - |In(G)|$, that must be pebbled to pebble G. It follows that

$$Qh \geq |V| - |In(G)|$$

The upper bound on Q is developed by adding S new red pebbles and showing that we may use these new pebbles to move all I/O operations in a sub-pebbling $\mathcal{P}_t$ to either the beginning or the end of the sub-pebbling without changing the number of computation steps or I/O operations. Thus, without changing them, we move all computation steps to a middle interval of $\mathcal{P}_t$, between the higher-level I/O operations.

We now show how this may be done. Consider a vertex v carrying a red pebble at some time during $\mathcal{P}_t$ that is pebbled for the first time with a blue pebble during $\mathcal{P}_t$ (vertex 7 at step 11 in Fig. 11.3). Instead of pebbling v with a blue pebble, use a new red pebble to keep a red pebble on v. (This is equivalent to swapping the new and old red pebbles on v.) This frees up the original red pebble to be used later in the sub-pebbling. Because we attach a red pebble to v for the entire pebbling $\mathcal{P}_t$, all later output operations from v in $\mathcal{P}_t$ can be deleted except for the last such operation, if any, which can be moved to the end of the interval. Note that if after v is given a blue pebble in $\mathcal{P}$, it is later given a red pebble, this red pebbling step and all subsequent blue pebbling steps except the last, if any, can be deleted. These changes do not affect any computation step in $\mathcal{P}_t$.

Consider a vertex v carrying a blue pebble at the start of $\mathcal{P}_t$ that later in $\mathcal{P}_t$ is given a red pebble (see vertex 4 at step 12 in Fig. 11.3). Consider the first pebbling of this kind. The red pebble assigned to v may have been in use prior to its placement on v. If a new red pebble is used for v, the first pebbling of v with a red pebble can be moved toward the beginning of $\mathcal{P}_t$ so that, without violating the precedence conditions of G, it precedes all placements of red pebbles on vertices without pebbles. Attach this new red pebble to v during $\mathcal{P}_t$. Subsequent placements of red pebbles on v when it carries a blue pebble during $\mathcal{P}_t$, if any, are thereby eliminated.



Step	1	2	3	4	5	6	7	8	9	10	11	12	13
Pebble	R1	R2	R2	B	R2	R2	R1	B	R2	R2	B	R2	R2
Vertex	$\downarrow 1$	$\downarrow 2$	5	$\uparrow 5$	$\downarrow 2$	6	$\downarrow 3$	$\uparrow 6$	$\downarrow 4$	7	$\uparrow 7$	$\downarrow 4$	8

$\mathcal{P}_t$ (steps 10–13)

Step	14	15	16	17	18	19	20	21	22	23
Pebble	R1	R2	R2	R2	R2	R1	R2	R2	R2	R2
Vertex	$\downarrow 5$	$\downarrow 7$	9	$\downarrow 7$	11	$\downarrow 6$	$\downarrow 8$	10	$\downarrow 8$	12

Figure 11.3 The vertices of an FFT graph are numbered and a pebbling schedule is given in which the two numbered red pebbles are used. Up (down) arrows identify steps in which an output (input) occurs; other steps are computation steps. Steps 10 through 13 of the schedule $\mathcal{P}_t$ contain two I/O operations. With two new red pebbles, the input at step 12 can be moved to the beginning of the interval and the output at step 11 can be moved after step 13.

We now derive an upper bound to Q. At the start of the pebbling of the middle interval of $\mathcal{P}_t$ there are at most $2S$ red pebbles on G, at most S original red pebbles plus S new red pebbles. Clearly, the number of vertices that can be pebbled in the middle interval with first-level pebbles is largest when all $2S$ red pebbles on G are allowed to move freely. It follows that at most $\rho(2S, G)$ vertices can be pebbled with red pebbles in any interval. Since all vertices must be pebbled with red pebbles, this completes the proof. ■

Combining Theorems 11.3.1 and 11.4.1 and a weak lower limit on the size of $T_l^{(L)}(\boldsymbol{p}, G)$, we have the following explicit lower bounds to $T_l^{(L)}(\boldsymbol{p}, G)$.

COROLLARY 11.4.1 *In the standard MHG when $T_l^{(L)}(\boldsymbol{p}, G) \geq \beta(s_{l-1} - 1)$ for $\beta > 1$, the following inequality holds for $2 \leq l \leq L$:*

$$T_l^{(L)}(\boldsymbol{p}, G) \geq \frac{\beta}{\beta + 1} \frac{s_{l-1}}{\rho(2s_{l-1}, G)}(|V| - |\,In(G)|)$$

In the I/O-limited MHG when $T_l^{(L)}(\boldsymbol{p}, G) \geq \beta(s_{l-1} - 1)$ for $\beta > 1$, the following inequality holds for $2 \leq l \leq L$:

$$T_l^{(L)}(\boldsymbol{p}, G) \geq \frac{\beta}{\beta + 1} \frac{s_{L-1}}{\rho(2s_{L-1}, G)}(|V| - |\,In(G)|)$$

11.5 Tradeoffs Between Space and I/O Time

We now apply the Hong-Kung method to a variety of important problems including matrix-vector multiplication, matrix-matrix multiplication, the fast Fourier transform, convolution, and merging and permutation networks.

11.5.1 Matrix-Vector Product

We examine here the matrix-vector product function $f_{A\boldsymbol{x}}^{(n)} : R^{n^2+n} \mapsto R^n$ over a commutative ring $\mathcal{R}$ described in Section 6.2.1 primarily to illustrate the development of efficient multi-level pebbling strategies. The lower bounds on I/O and computation time for this problem are trivial to obtain. For the matrix-vector product, we assume that the graphs used are those associated with inner products. The inner product $\boldsymbol{u} \cdot \boldsymbol{v}$ of n-vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ over a ring $\mathcal{R}$ is defined by:

$$\boldsymbol{u} \cdot \boldsymbol{v} = \sum_{i=1}^{n} u_i \cdot v_i$$

The graph of a straight-line program to compute this inner product is given in Fig. 11.4, where the additions of products are formed from left to right.

The matrix-vector product is defined here as the pebbling of a collection of inner product graphs. As suggested in Fig. 11.4, each inner product graph can be pebbled with three red pebbles.

THEOREM 11.5.1 *Let G be the graph of a straight-line program for the product of the matrix A with the vector $\boldsymbol{x}$. Let G be pebbled in the* **standard MHG** *with the resource vector $\boldsymbol{p}$. There is a*

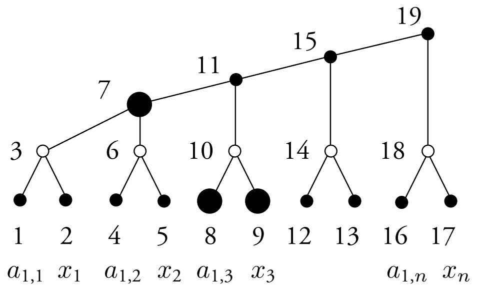


Figure 11.4 The graph of an inner product computation showing the order in which vertices are pebbled. Input vertices are labeled with the entries in the matrix A and vector $\boldsymbol{x}$ that are combined. Open vertices are product vertices; those above them are addition vertices.

pebbling strategy $\mathcal{P}$ of G with $p_l \geq 1$ for $2 \leq l \leq L-1$ and $p_1 \geq 3$ such that $T_1^{(L)}(\boldsymbol{p}, G, \mathcal{P}) = 2n^2 - n$, the minimum value, and the following bounds hold simultaneously:

$$n^2 + 2n \leq T_l^{(L)}(\boldsymbol{p}, G, \mathcal{P}) \leq 2n^2 + n$$

Proof The lower bound $T_l^{(L)}(\boldsymbol{p}, G, \mathcal{P}) \geq n^2+2n$, $1 \leq l \leq L$, follows from Lemma 11.3.1 because there are $n^2 + n$ inputs and n outputs to the matrix-vector product. The upper bounds derived below represent the number of operations performed by a pebbling strategy that uses three level-1 pebbles and one pebble at each of the other levels.

Each of the n results of the matrix-vector product is computed as an inner product in which successive products $a_{i,j}x_j$ are formed and added to a running sum, as suggested by Fig. 11.4. Each of the n^2 entries of the matrix A (leaves of inner product trees) is used in one inner product and is pebbled once at levels $L, L-1, \dots, 1$ when needed. The n entries in $\boldsymbol{x}$ are used in every inner product and are pebbled once at each level for each of the n inner products. First-level pebbles are placed on each vertex of each inner product tree in the order suggested in Fig. 11.4. After the root vertex of each tree is pebbled with a first-level pebble, it is pebbled at levels $2, \dots, L$.

It follows that one I/O operation is performed at each level on each vertex associated with an entry in A and the outputs and that n I/O operations are performed at each level on each vertex associated with an entry in $\boldsymbol{x}$, for a total of $2n^2 + n$ I/O operations at each level. This pebbling strategy places a first-level pebble once on each interior vertex of each of the n inner product trees. Such trees have $2n - 1$ internal vertices. Thus, this strategy takes $2n^2 - n$ computation steps. ■

As the above results demonstrate, the matrix-vector product is an example of an **I/O-bounded problem**, a problem for which the amount of I/O required at each level in the memory hierarchy is comparable to the number of computation steps. Returning to the discussion in Section 11.1.2, we see that as CPU speed increases with technological advances, a balanced computer system can be constructed for this problem only if the I/O speed increases proportionally to CPU speed.

The I/O-limited version of the MHG for the matrix-vector product is the same as the standard version because only first-level pebbles are used on vertices that are neither input or output vertices.

11.5.2 Matrix-Matrix Multiplication

In this section we derive upper and lower bounds on exchanges between I/O time and space for the $n \times n$ matrix multiplication problem in the standard and I/O-limited MHG. We show that the lower bounds on computation and I/O time can be matched by efficient pebbling strategies.

Lower bounds for the standard MHG are derived for the **family $\mathcal{F}_n$ of inner product graphs for $n \times n$ matrix multiplication**, namely, the set of graphs to multiply two $n \times n$ matrices using just inner products to compute entries in the product matrix. (See Section 6.2.2.) We allow the additions in these inner products to be performed in any order.

The lower bounds on I/O time derived below for the I/O-limited MHG apply to all DAGs for matrix multiplication. Since these DAGs include graphs other than the inner product trees in $\mathcal{F}_n$, one might expect the lower bounds for the I/O-limited case to be smaller than those derived for graphs in $\mathcal{F}_n$. However, this is not the case, apparently because efficient pebbling strategies for matrix multiplication perform I/O operations only on input and output vertices, not on internal vertices. The situation is very different for the discrete Fourier transform, as seen in the next section.

We derive results first for the red-blue pebble game, that is, the two-level MHG, and then generalize them to the multi-level MHG. We begin by deriving an upper bound on the S-span for the family of inner product matrix multiplication graphs.

LEMMA 11.5.1 *For every graph $G \in \mathcal{F}_n$ the S-span $\rho(S, G)$ satisfies the bound $\rho(S, G) \leq 2S^{3/2}$ for $S \leq n^2$.*

Proof $\rho(S, G)$ is the maximum number of vertices of $G \in \mathcal{F}_n$ that can be pebbled with S red pebbles from an initial placement of these pebbles, maximized over all such initial placements. Let A, B, and C be $n \times n$ matrices with entries $\{a_{i,j}\}$, $\{b_{i,j}\}$, and $\{c_{i,j}\}$, respectively, where $1 \leq i, j \leq n$. Let $C = A \times B$. The term $c_{i,j} = \sum_k a_{i,k} b_{k,j}$ is associated with the root vertex in of a unique inner product tree. Vertices in this tree are either addition vertices, product vertices associated with terms of the form $a_{i,k} b_{k,j}$, or input vertices associated with entries in the matrices A and B. Each product term $a_{i,k} b_{k,j}$ is associated with a unique term $c_{i,j}$ and tree, as is each addition operator.

Consider an initial placement of $S \leq n^2$ pebbles of which r are in addition trees (they are on addition or product vertices). Let the remaining $S - r$ pebbles reside on input vertices. Let p be the number of product vertices that can be pebbled from these pebbled inputs. We show that at most $p + r - 1$ additional pebble placements are possible from the initial placement, giving a total of at most $\pi = 2p + r - 1$ pebble placements. (Figure 11.5

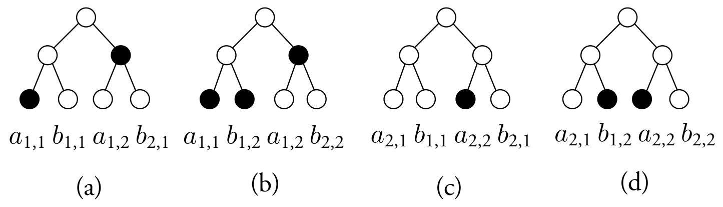

$a_{1,1}$ $b_{1,1}$ $a_{1,2}$ $b_{2,1}$
$a_{1,1}$ $b_{1,2}$ $a_{1,2}$ $b_{2,2}$
$a_{2,1}$ $b_{1,1}$ $a_{2,2}$ $b_{2,1}$
$a_{2,1}$ $b_{1,2}$ $a_{2,2}$ $b_{2,2}$
(a)
(b)
(c)
(d)

Figure 11.5 Graph of the inner products used to form the product of two 2×2 matrices. (Common input vertices are repeated for clarity.)

shows a graph G for a 2×2 matrix multiplication algorithm in which the product vertices are those just below the output vertices. The black vertices carry pebbles. In this example $r = 2$ and $p = 1$. While $p + r - 1 = 2$, only one pebble placement is possible on addition trees in this example.)

Given the dependencies of graphs in $\mathcal{F}_n$, there is no loss in generality in assuming that product vertices are pebbled before pebbles are advanced in addition trees. It follows that at most $p+r$ addition-tree vertices carry pebbles before pebbles are advanced in addition trees. These pebbled vertices define subtrees of vertices that can be pebbled from the $p + r$ initial pebble placements. Since a binary tree with n leaves has $n - 1$ non-leaf nodes, it follows that if there are t such trees, at most $p+r-t$ pebble placements will be made, not counting the original placement of pebbles. This number is maximized at $t = 1$. (See Problem 11.9.)

We now complete the proof by deriving an upper bound on p. Let $\mathcal{A}$ be the 0−1 $n \times n$ matrix whose (i, j) entry is 1 if the variable in the (i, j) position of the matrix A carries a pebble initially and 0 otherwise. Let $\mathcal{B}$ be similarly defined for B. It follows that the (i, j) entry, $\delta_{i,j}$, of the matrix product $\mathcal{C} = \mathcal{A} \times \mathcal{B}$, where addition and multiplication are over the integers, is equal to the number of products that can be formed that contribute to the (i, j) entry of the result matrix C. Thus $p = \sum_{i,j} \delta_{i,j}$. We now show that $p \leq \sqrt{S}(S-r)$.

Let $\mathcal{A}$ and $\mathcal{B}$ have a and b 1's, respectively, where $a+b = S-r$. There are at most a/α rows of $\mathcal{A}$ containing at least α 1's. The maximum number of products that can be formed from such rows is ab/α because each 1 in $\mathcal{B}$ combine with a 1 in each of these rows. Now consider the product of other rows of $\mathcal{A}$ with columns of $\mathcal{B}$. At most S such row-column inner products are formed since at most S outputs can be pebbled. Since each of them involves a row with at most α 1's, at most αS products of pairs of variables can be formed. Thus, a total of at most $p = ab/\alpha + \alpha S$ products can be formed. We are free to choose α to minimize this sum ($\alpha = \sqrt{ab/S}$ does this) but must choose a and b to maximize it ($a = (S-r)/2$ satisfies this requirement). The result is that $p \leq \sqrt{S}(S-r)$. We complete the proof by observing that $\pi = 2p + r - 1 \leq 2\sqrt{S}S$ for $r \geq 0$. ■

Theorem 11.5.2 states bounds that apply to the computation and I/O time in the red-blue pebble game for matrix multiplication.

THEOREM 11.5.2 *For every graph G in the family $\mathcal{F}_n$ of inner product graphs for multiplying two $n \times n$ matrices and for every pebbling strategy $\mathcal{P}$ for G in the* **red-blue pebble game** *that uses $S \geq 3$ red pebbles, the computation and I/O-time satisfy the following lower bounds:*

$$T_1^{(2)}(S, G, \mathcal{P}) = \Omega(n^3)$$
$$T_2^{(2)}(S, G, \mathcal{P}) = \Omega\left(\frac{n^3}{\sqrt{S}}\right)$$

Furthermore, there is a pebbling strategy $\mathcal{P}$ for G with $S \geq 3$ red pebbles such that the following upper bounds hold simultaneously:

$$T_1^{(2)}(S, G, \mathcal{P}) = O(n^3)$$
$$T_2^{(2)}(S, G, \mathcal{P}) = O\left(\frac{n^3}{\sqrt{S}}\right)$$

The lower bound on I/O time stated above applies for **every graph** *of a straight-line program for matrix multiplication in the* **I/O-limited red-blue pebble game**. *The upper bound on I/O time* *also applies for this game. The computation time in the I/O-limited red-blue pebble game satisfies the following bound:*

$$T_1^{(2)}(S, G, \mathcal{P}) = \Omega\left(\frac{n^3}{\sqrt{S}}\right)$$

Proof For the standard MHG, the lower bound to $T_1^{(2)}(S, G, \mathcal{P})$ follows from the fact that every graph in $\mathcal{F}_n$ has $\Theta(n^3)$ vertices and Lemma 11.3.1. The lower bound to $T_2^{(2)}(S, G)$ follows from Corollary 11.4.1 and Lemma 11.5.1 and the lower bound to $T_1^{(2)}(S, G, \mathcal{P})$ for the I/O-limited MHG follows from Theorem 11.3.1.

We now describe a pebbling strategy that has the I/O time stated above and uses the obvious algorithm suggested by Fig. 11.6. If S red pebbles are available, let $r = \lfloor\sqrt{S/3}\rfloor$ be an integer that divides n. (If r does not divide n, embed A, B and C in larger matrices for which r does divide n. This requires at most doubling n.) Let the $n \times n$ matrices A, B and C be partitioned into $n/r \times n/r$ matrices; that is, $A = [a_{i,j}]$, $B = [b_{i,j}]$, and $C = [c_{i,j}]$, whose entries are $r \times r$ matrices. We form the $r \times r$ submatrix $c_{i,j}$ of C as the inner product of a row of $r \times r$ submatrices of A with a column of such submatrices of B:

$$c_{i,j} = \sum_{q=1}^{r} a_{i,q} \times b_{q,j}$$

We begin by placing blue pebbles on each entry in matrices A and B. Compute $c_{i,j}$ by computing $a_{i,q} \times b_{q,j}$ for $q = 1, 2, \ldots, r$ and adding successive products to the running sum. Keep r^2 red pebbles on the running sum. Compute $a_{i,q} \times b_{q,j}$ by placing and holding r^2 red pebbles on the entries in $a_{i,q}$ and r red pebbles on one column of $b_{q,j}$ at a time. Use two additional red pebbles to compute the r^2 inner products associated with entries of $c_{i,j}$ in the fashion suggested by Fig. 11.4 if $r \geq 2$ and one additional pebble if $r = 1$. The maximum number of red pebbles in use is 3 if $r = 1$ and at most $2r^2 + r + 2$ if $r \geq 2$. Since $2r^2 + r + 2 \leq 3r^2$ for $r \geq 2$, in both cases at most $3r^2$ red pebbles are needed. Thus, there are enough red pebbles to play this game because $r = \lfloor\sqrt{S/3}\rfloor$ implies that $3r^2 \leq S$, the number of red pebbles. Since $r \geq 1$, this requires that $S \geq 3$.

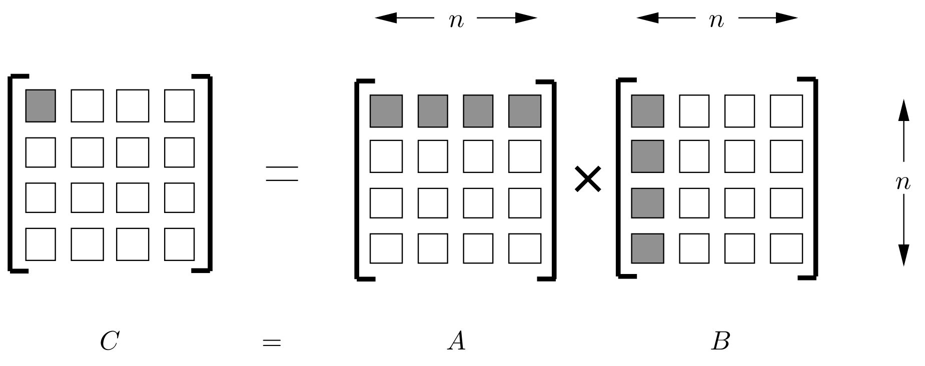


Figure 11.6 A pebbling schema for matrix multiplication based on the representation of a matrix in terms of block submatrices. A submatrix of C is computed as the inner product of a row of blocks of A with a column of blocks of B.

This algorithm performs one input operation on each entry of $a_{i,q}$ and $b_{q,j}$ to compute $c_{i,j}$. It also performs one output operation per entry to compute $c_{i,j}$ itself. Summing over all values of i and j, we find that n^2 output operations are performed on entries in C. Since there are $(n/r)^2$ submatrices $a_{i,q}$ and $b_{q,j}$ and each is used to compute n/r terms $c_{u,v}$, the number of input operations on entries in A and B is $2(n/r)^2 r^2 (n/r) = 2n^3/r$. Because $r = \lfloor \sqrt{S/3} \rfloor$, we have $r \geq \sqrt{S/3} - 1$, from which the upper bound on the number of I/O operations follows. Since each product and addition vertex in each inner product graph is pebbled once, $O(n^3)$ computation steps are performed.

The bound on $T_2^{(2)}(S, G, \mathcal{P})$ for the I/O-limited game follows from two observations. First, the computational inequality of Theorem 10.4.1 provides a lower bound to T_I, the number of times that input vertices are pebbled in the red-pebble game when only red pebbles are used on vertices. This is the I/O-limited model. Second, the lower bound of Theorem 10.5.4 on T (actually, T_I) is of the form desired. ■

These results and the strategy given for the two-level case carry over to the multi-level case, although considerable care is needed to insure that the pebbling strategy does not fragment memory and lead to inefficient upper bounds.

Even though the pebbling strategy given below is an I/O-limited strategy, it provides bounds on time in terms of space that match the lower bounds for the standard MHG.

THEOREM 11.5.3 *For every graph G in the family $\mathcal{F}_n$ of inner product graphs for multiplying two $n \times n$ matrices and for every pebbling strategy $\mathcal{P}$ for G in the* **standard MHG** *with resource vector $\boldsymbol{p}$ that uses $p_1 \geq 3$ first-level pebbles, the computation and I/O time satisfy the following lower bounds, where $s_l = \sum_{j=1}^{l} p_j$ and k is the largest integer such that $s_k \leq 3n^2$:*

$$T_1^{(L)}(\boldsymbol{p}, G, \mathcal{P}) = \Omega\left(n^3\right)$$

$$T_l^{(L)}(\boldsymbol{p}, G, \mathcal{P}) = \begin{cases} \Omega\left(n^3/\sqrt{s_{l-1}}\right) & \textit{for } 2 \leq l \leq k \\ \Omega\left(n^2\right) & \textit{for } k+1 \leq l \leq L \end{cases}$$

Furthermore, there is a pebbling strategy $\mathcal{P}$ for G with $p_1 \geq 3$ such that the following upper bounds hold simultaneously:

$$T_1^{(L)}(\boldsymbol{p}, G, \mathcal{P}) = O(n^3)$$

$$T_l^{(L)}(\boldsymbol{p}, G, \mathcal{P}) = \begin{cases} O\left(n^3/\sqrt{s_{l-1}}\right) & \textit{for } 2 \leq l \leq k \\ O\left(n^2\right) & \textit{for } k+1 \leq l \leq L \end{cases}$$

In the I/O-limited MHG the upper bounds given above apply. The following lower bound on the I/O time applies to **every graph** *G for $n \times n$ matrix multiplication and every pebbling strategy $\mathcal{P}$, where $S = s_{L-1}$:*

$$T_l^{(L)}(\boldsymbol{p}, G, \mathcal{P}) = \Omega\left(n^3/\sqrt{S}\right) \quad \textit{for } 1 \leq l \leq L$$

Proof The lower bounds on $T_l^{(L)}(\boldsymbol{p}, G, \mathcal{P})$, $2 \leq l \leq L$, follow from Theorems 11.3.1 and 11.5.2. The lower bound on $T_1^{(L)}(\boldsymbol{p}, G, \mathcal{P})$ follows from the fact that every graph in $\mathcal{F}_n$ has $\Theta(n^3)$ vertices to be pebbled.

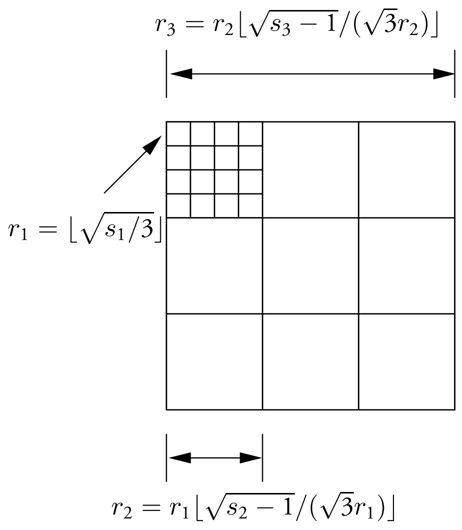

$r_3 = r_2\lfloor\sqrt{s_3 - 1}/(\sqrt{3}r_2)\rfloor$
$r_1 = \lfloor\sqrt{s_1/3}\rfloor$
$r_2 = r_1\lfloor\sqrt{s_2 - 1}/(\sqrt{3}r_1)\rfloor$

Figure 11.7 A three-level decomposition of a matrix.

We now describe a multi-level recursive pebbling strategy satisfying the upper bounds given above. It is based on the two-level strategy given in the proof of Theorem 11.5.2. We compute C from A and B using inner products.

Our approach is to successively block A, B, and C into $r_i \times r_i$ submatrices for $i = k, k-1, \dots, 1$ where the r_i are chosen, as suggested in Fig. 11.7, so they divide on another and avoid memory fragmentation. Also, they are also chosen relative to s_i so that enough pebbles are available to pebble $r_i \times r_i$ submatrices, as explained below.

$$r_i = \begin{cases} \left\lfloor\sqrt{s_1/3}\right\rfloor & i = 1 \\ \\ r_{i-1}\left\lfloor\sqrt{(s_i - i + 1)}/(\sqrt{3}r_{i-1})\right\rfloor & i \geq 2 \end{cases}$$

Using the fact that $b/2 \leq a\lfloor b/a\rfloor \leq b$ for integers a and b satisfying $1 \leq a \leq b$ (see Problem 11.1), we see that $\sqrt{(s_i - i + 1)/12} \leq r_i \leq \sqrt{(s_i - i + 1)/3}$. Thus, $s_i \geq 3r_i^2 + i - 1$. Also, $r_k^2 \leq n^2$ because $s_k \leq 3n^2$.

By definition, s_l pebbles are available at level l and below. As stated earlier, there is at least one pebble at each level above the first. From the s_l pebbles at level l and below we create a reserve set containing one pebble at each level except the first. This reserve set is used to perform I/O operations as needed.

Without loss of generality, assume that r_k divides n. (If not, n must be at most doubled for this to be true. Embed A, B, and C in such larger matrices.) A, B, and C are then blocked into $r_k \times r_k$ submatrices (call them $a_{i,j}$, $b_{i,j}$, and $c_{i,j}$), and these in turn are blocked into $r_{k-1} \times r_{k-1}$ submatrices, continuing until 1×1 submatrices are reached. The submatrix $c_{i,j}$ is defined as

$$c_{i,j} = \sum_{q=1}^{r_k} a_{i,q} \times b_{q,j}$$

As in Theorem 11.5.2, $c_{i,j}$ is computed as a running sum, as suggested in Fig. 11.4, where each vertex is associated with an $r_k \times r_k$ submatrix. It follows that $3r_k^2$ pebbles at level k or less (not including the reserve pebbles) suffice to hold pebbles on submatrices $a_{i,q}$, $b_{q,j}$ and the running sum. To compute a product $a_{i,q} \times b_{q,j}$, we represent $a_{i,q}$ and $b_{q,j}$ as block matrices with blocks that are $r_{k-1} \times r_{k-1}$ matrices. Again, we form this product as suggested in Fig. 11.4, using $3r_{k-1}^2$ pebbles at levels $k-1$ or lower. This process is repeated until we encounter a product of $r_1 \times r_1$ matrices, which is then pebbled according to the procedure given in the proof of Theorem 11.5.2.

Let's now determine the number of I/O and computation steps at each level. Since all non-input vertices of G are pebbled once, the number of computation steps is $O(n^3)$. I/O operations are done only on input and output vertices. Once an output vertex has been pebbled at the first level, reserve pebbles can be used to place a level-L pebble on it. Thus one output is done on each of the n^2 output vertices at each level.

We now count the I/O operations on input vertices starting with level k. $n \times n$ matrices A, B, and C contain $r_k \times r_k$ matrices, where r_k divides n. Each of the $(n/r_k)^2$ submatrices $a_{i,q}$ and $b_{q,j}$ is used in (n/r_k) inner products and at most r_k^2 I/O operations at level k are performed on them. (If most of the s_k pebbles at level k or less are at lower levels, fewer level-k I/O operations will be performed.) Thus, at most $2(n/r_k)^2(n/r_k)r_k^2 = 2n^2/r_k$ I/O operations are performed at level k. In turn, each of the $r_k \times r_k$ matrices contains $(r_k/r_{k-1})^2$ $r_{k-1} \times r_{k-1}$ matrices; each of these is involved in (r_k/r_{k-1}) inner products each of which requires at most r_{k-1}^2 I/O operations. Since there are at most $(n/r_{k-1})^2$ $r_{k-1} \times r_{k-1}$ submatrices in each of A, B, and C, at most $2n^3/r_{k-1}$ I/O operations are performed at level $k-1$. Continuing in this fashion, at most $2n^3/r_l$ I/O operations are performed at level l for $2 \leq l \leq k$. Since $r_l \geq \sqrt{(s_i - i + 1)/12}$, we have the desired conclusion.

Since the above pebbling strategy does not place pebbles at level 2 or above on any vertex except input and output vertices, it applies in the I/O-limited case. The lower bound follows from Lemma 11.3.1 and Theorem 11.5.2. ■

11.5.3 The Fast Fourier Transform

The fast Fourier transform (FFT) algorithm is described in Section 6.7.3 (an FFT graph is given in Fig. 11.1). A lower bound is obtained by the Hong-Kung method for the FFT by deriving an upper bound on the S-span of the FFT graph. In this section all logarithms have base 2.

LEMMA 11.5.2 *The S-span of the FFT graph $F^{(d)}$ on $n = 2^d$ inputs satisfies $\rho(S,G) \leq 2S \log S$ when $S \leq n$.*

Proof $\rho(S,G)$ is the maximum number of vertices of G that can be pebbled with S red pebbles from an initial placement of these pebbles, maximized over all such initial placements. G contains many two-input FFT (butterfly) graphs, as shown in Fig. 11.8. If v_1 and v_2 are the output vertices in such a two-input FFT and if one of them is pebbled, we

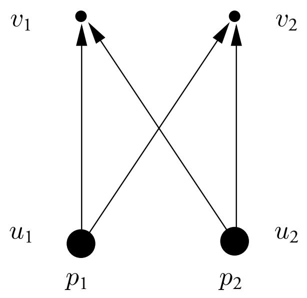


Figure 11.8 A two-input butterfly graph with pebbles p_1 and p_2 resident on inputs.

obtain an upper bound on the number of pebbled vertices if we assume that both of them are pebbled. In this proof we let $\{p_i \mid 1 \le i \le S\}$ denote the S pebbles available to pebble G. We assign an integer cost $num(p_i)$ (initialized to zero) to the ith pebble p_i in order to derive an upper bound to the total number of pebble placements made on G.

Consider a matching pair of output vertices v_1 and v_2 of a two-input butterfly graph and their common predecessors u_1 and u_2, as suggested in Fig. 11.8. Suppose that on the next step we can place a pebble on v_1. Then pebbles (call them p_1 and p_2) must reside on u_1 and u_2. Advance p_1 and p_2 to both v_1 and v_2. (Although the rules stipulate that an additional pebble is needed to advance the two pebbles, violating this restriction by allowing their movement to v_1 and v_2 can only increase the number of possible moves, a useful effect since we are deriving an upper bound on the number of pebble placements.)

After advancing p_1 and p_2, if $num(p_1) = num(p_2)$, augment both by 1; otherwise, augment the smaller by 1. Since the predecessors of two vertices in an FFT graph are in disjoint trees, there is no loss in assuming that all S pebbles remain on the graph in a pebbling that maximizes the number of pebbled vertices. Because two pebble placements are possible each time $num(p_i)$ increases by 1 for some i, $\rho(S, G) \le 2\sum_{1 \le i \le S} num(p_i)$.

We now show that the number of vertices that contained pebbles initially and are connected via paths to the vertex covered by p_i is at least $2^{num(p_i)}$. That is, $2^{num(p_i)} \le S$ or $num(p_i) \le \log_2 S$, from which the upper bound on $\rho(S, G)$ follows. Our proof is by induction. For the base case of $num(p_i) = 1$, two pebbles must reside on the two immediate predecessors of a vertex containing the pebble p_i. Assume that the hypothesis holds for $num(p_i) \le e - 1$. We show that it holds for $num(p_i) = e$. Consider the first point in time that $num(p_i) = e$. At this time p_i and a second pebble p_j reside on a matching pair of vertices, v_1 and v_2. Before these pebbles are advanced to these two vertices from u_1 and u_2, the immediate predecessors of v_1 and v_2, the smaller of $num(p_i)$ and $num(p_j)$ has a value of $e - 1$. This must be p_i because its value has increased. Thus, each of u_1 and u_2 has at least 2^{e-1} predecessors that contained pebbles initially. Because the predecessors of u_1 and u_2 are disjoint, each of v_1 and v_2 has at least $2^e = 2^{num(p_i)}$ predecessors that carried pebbles initially. ■

This upper bound on the S-span is combined with Theorem 11.4.1 to derive a lower bound on the I/O time at level l to pebble the FFT graph. We derive upper bounds that match to within a multiplicative constant when the FFT graph is pebbled in the standard MHG. We develop bounds for the red-blue pebble game and then generalize them to the MHG.

THEOREM 11.5.4 *Let the FFT graph on $n = 2^d$ inputs, $F^{(d)}$, be pebbled in the* **red-blue pebble game** *with S red pebbles. When $S \geq 3$ there is a pebbling of $F^{(d)}$ such that the following bounds hold simultaneously, where $T_1^{(2)}(p_1, F^{(d)})$ and $T_2^{(2)}(p_1, F^{(d)})$ are the computation and I/O time in a minimal pebbling of $F^{(d)}$:*

$$T_1^{(2)}(S, F^{(d)}) = \Theta(n \log n)$$
$$T_2^{(2)}(S, F^{(d)}) = \Theta\left(\frac{n \log n}{\log S}\right)$$

Proof The lower bound on $T_1^{(2)}(S, F^{(d)})$ is obvious; every vertex in $F^{(d)}$ must be pebbled a first time. The lower bound on $T_2^{(2)}(S, F^{(d)})$ follows from Corollary 11.4.1, Theorem 11.3.1, Lemma 11.5.2, and the obvious lower bound on $|V|$. We now exhibit a pebbling strategy giving upper bounds that match the lower bounds up to a multiplicative factor.

As shown in Corollary 6.7.1, $F^{(d)}$ can be decomposed into $\lceil d/e \rceil$ stages, $\lfloor d/e \rfloor$ stages containing 2^{d-e} copies of $F^{(e)}$ and one stage containing 2^{d-k} copies of $F^{(k)}$, $k = d - \lfloor d/e \rfloor e$. (See Fig. 11.9.) The output vertices of one stage are the input vertices to the next. For example, $F^{(12)}$ can be decomposed into three stages with $2^{12-4} = 256$ copies of $F^{(4)}$ on each stage and one stage with 2^{12} copies of $F^{(0)}$, a single vertex. (See Fig. 11.10.) We use this decomposition and the observation that $F^{(e)}$ can be pebbled level by level with $2^e + 1$ level-1 pebbles without repebbling any vertex to develop our pebbling strategy for $F^{(d)}$.

Given S red pebbles, our pebbling strategy is based on this decomposition with $e = d_0 = \lfloor \log_2(S - 1) \rfloor$. Since $S \geq 3$, $d_0 \geq 1$. Of the S red pebbles, we actually use only $S_0 = 2^{d_0} + 1$. Since $S_0 \leq S$, the number of I/O operations with S_0 red pebbles is no

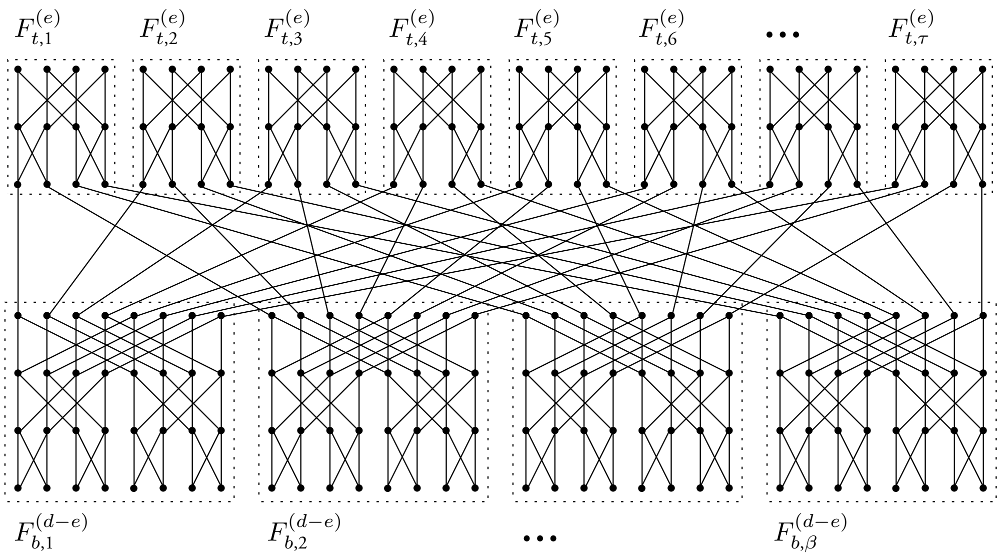


Figure 11.9 Decomposition of the FFT graph $F^{(d)}$ into $\beta = 2^e$ bottom FFT graphs $F^{(d-e)}$ and $\tau = 2^{d-e}$ top $F^{(e)}$. Edges between bottom and top sub-FFT graphs identify common vertices between the two.

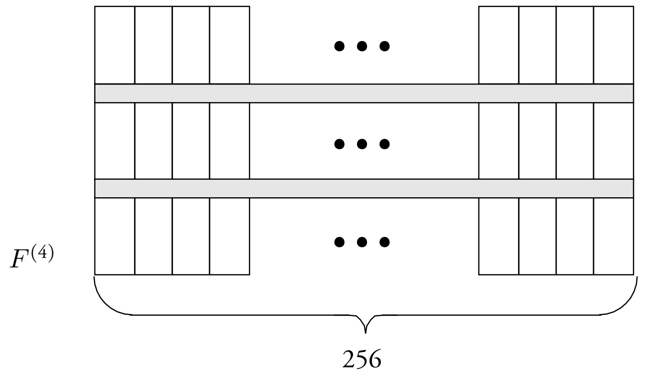


Figure 11.10 The decomposition of an FFT graph $F^{(12)}$ into three stages each containing 256 copies of $F^{(4)}$. The gray areas identify rows of $F^{(12)}$ in which inputs to one copy of $F^{(4)}$ are outputs of copies of $F^{(4)}$ at the preceding level.

less than with S red pebbles. Let $d_1 = \lfloor d/d_0 \rfloor$. Then, $F^{(d)}$ is decomposed into d_1 stages each containing 2^{d-d_0} copies of $F^{(d_0)}$ and one stage containing 2^{d-t} copies of $F^{(t)}$ where $t = d - d_0 d_1$. Since $t \le d_0$, each vertex in $F^{(t)}$ can be pebbled with S_0 pebbles without re-pebbling vertices. The same applies to $F^{(d_0)}$.

The pebbling strategy for the red-blue pebble game is based on this decomposition. Pebbles are advanced to outputs of each of the bottom FFT subgraphs $F^{(t)}$ using $2^t+1 \le S_0$ red pebbles, after which the red pebbles are replaced with blue pebbles. The subgraphs $F^{(d_0)}$ in each of the succeeding stages are then pebbled in the same fashion; that is, their blue-pebbled inputs are replaced with red pebbles and red pebbles are advanced to their outputs after which they are replaced with blue pebbles.

This strategy pebbles each vertex once with red pebbles with the exception of vertices common to two FFT subgraphs which are pebbled twice. It follows that $T_1^{(L)}(S, F^{(d)}) \le 2^{d+1}(d+1) = 2n(\log_2 n + 1)$. This strategy also executes one I/O operation for each of the 2^d inputs and outputs to $F^{(d)}$ and two I/O operations for each of the 2^d vertices common to adjacent stages. Since there are $\lceil d/d_0 \rceil$ stages, there are $\lceil d/d_0 \rceil - 1$ such pairs of stages. Thus, the number of I/O operations satisfies $T_2^{(L)}(S, F^{(d)}) \le 2^{d+1}\lceil d/d_0 \rceil \le 2n(\log_2 n/(\log_2 S/4) + 1) = O(n \log n/\log S)$. ■

The bounds for the multi-level case generalize those for the red-blue pebble game. As with matrix multiplication, care must be taken to avoid memory fragmentation.

THEOREM 11.5.5 *Let the FFT graph on $n = 2^d$ inputs, $F^{(d)}$, be pebbled in the* **standard MHG** *with resource vector $\mathbf{p}$. Let $s_l = \sum_{j=1}^{l} p_j$ and let k be the largest integer such that $s_k \le n$. When $p_1 \ge 3$, the following lower bounds hold for all pebblings of $F^{(d)}$ and there exists a pebbling $\mathcal{P}$ for* *which the upper bounds are simultaneously satisfied:*

$$T_l^{(L)}(\boldsymbol{p}, F^{(d)}, \mathcal{P}) = \begin{cases} \Theta(n \log n) & l = 1 \\ \Theta\left(\frac{n \log n}{\log s_{l-1}}\right) & 2 \leq l \leq k \\ \Theta(n) & k+1 \leq l \leq L \end{cases}$$

Proof Proofs of the first two lower bounds follow from Lemma 11.3.1 and Theorem 11.5.4. The third follows from the fact that pebbles at every level must be placed on each input and output vertex but no intermediate vertex. We now exhibit a pebbling strategy giving upper bounds that match (up to a multiplicative factor) these lower bounds for all $1 \leq l \leq L$. (See Fig. 11.9.)

We define a non-decreasing sequence $\boldsymbol{d} = (d_0, d_1, d_2, \dots, d_{L-1})$ of integers used below to describe an efficient multi-level pebbling strategy for $F^{(d)}$. Let $d_0 = 1$ and $d_1 = \lfloor \log(s_1 - 1) \rfloor \geq 1$, where $s_1 = p_1 \geq 3$. Define m_r and d_r for $2 \leq r \leq L-1$ by

$$m_r = \left\lfloor \frac{\lfloor \log \min(s_r - 1, n) \rfloor}{d_{r-1}} \right\rfloor$$
$$d_r = m_r d_{r-1}$$

It follows that $s_r \geq 2^{d_r} + 1$ when $s_r \leq n+1$ since $a\lfloor b/a \rfloor \leq b$. Because $\lfloor \log a \rfloor \geq (\log a)/2$ when $a \geq 1$ and also $a\lfloor b/a \rfloor \geq b/2$ for integers a and b when $1 \leq a \leq b$ (see Problem 11.1), it follows that $d_r \geq \log(\min(s_r - 1, n))/4$. The values d_l are chosen to avoid memory fragmentation.

Before describing our pebbling strategy, note that because we assume at least one pebble is available at each level in the hierarchy, it is possible to perform an I/O operation at each level. Also, pebbles at levels less than l can be used as though they were at level l.

Our pebbling strategy is based on the decomposition of $F^{(d)}$ into FFT subgraphs $F^{(d_k)}$, each of which is decomposed into FFT subgraphs $F^{(d_{k-1})}$, and so on, until reaching FFT subgraphs $F^{(1)}$ that are two-input, two-output butterfly graphs. To pebble $F^{(d)}$ we apply the strategy described in the proof of Theorem 11.5.4 as follows. We decompose $F^{(2)}$ into d_2/d_1 stages, each containing $2^{d_2 - d_1}$ copies of $F^{(1)}$, which we pebble with $s_1 = p_1$ first-level pebbles using this strategy. By the analysis in the proof of Theorem 11.5.4, 2^{d_2+1} level-2 I/O operations are performed on inputs and outputs to $F^{(d_2)}$ as well as another 2^{d_2+1} level-2 I/O operations on the vertices between two stages. Since there are d_2/d_1 stages, a total of $(d_2/d_1)2^{d_2+1}$ level-2 I/O operations are performed. We then decompose $F^{(3)}$ into d_3/d_2 stages each containing $2^{d_3 - d_2}$ copies of $F^{(2)}$. We pebble $F^{(3)}$ with s_2 pebbles at level 1 or 2 by pebbling copies of $F^{(2)}$ in stages, using $(d_3/d_2)2^{d_3+1}$ level-3 I/O operations and using $(d_3/d_2)2^{d_3 - d_2}$ times as many level-2 I/O operations as used by $F^{(2)}$. Let $n_2^{(3)}$ be the number of level-2 I/O operations used to pebble $F^{(3)}$. Then $n_2^{(3)} = (d_3/d_1)2^{d_3+1}$.

Continuing in this fashion, we pebble $F^{(r)}$, $1 \leq r \leq k$, with s_{r-1} pebbles at levels l or below by pebbling copies of $F^{(r-1)}$ in stages, using $(d_r/d_{r-1})2^{d_r+1}$ level-r I/O operations and using $(d_r/d_{r-1})2^{d_r - d_{r-1}}$ as many level-j I/O operations for $1 \leq j \leq r-1$. Let $n_j^{(r)}$ be the number of level-j I/O operations used to pebble $F^{(r)}$. By induction it follows that $n_j^{(r)} = (d_r/d_j)2^{d_r+1}$.

For $r \geq k$, the number of pebbles available at level r or less is at least $2^d + 1$, which is enough to pebble $F^{(d)}$ by levels without performing I/O operations above level $k+1$; this means that I/O operations at these levels are performed only on inputs, giving the bound $T_l^{(L)}(\boldsymbol{p}, F^{(d)}, \mathcal{P}) = O(n)$, $n = 2^d$, for $k+1 \le r \le L$. When $r \le k$, we pebble $F^{(d)}$ by decomposing it into $\lceil d/d_k \rceil$ stages such that each stage, except possibly the first, contains 2^{d-d_k} copies of the FFT subgraph $F^{(d_k)}$. The first stage has 2^{d-d^*} copies of $F^{(d^*)}$ of depth $d^* = d - (\lceil d/d_k \rceil - 1)d_k$, which we treat as subgraphs of the subgraph $F^{(d_k)}$ and pebble to completion with a number of operations at each level that is at most the number to pebble $F^{(d_k)}$. Each instance of $F^{(d_k)}$ is pebbled with s_{k-1} pebbles at level $k-1$ or lower and a pebble at level k or higher is left on its output. Since $s_{k+1} \ge n+1$, there are enough pebbles to do this.

Thus $T_l^{(L)}(\boldsymbol{p}, F^{(d)}, \mathcal{P})$ satisfies the following bound for $1 \le l \le L$:

$$T_l^{(L)}(\boldsymbol{p}, F^{(d)}, \mathcal{P}) \le \lceil d/d_k \rceil 2^{d-d_k} T_l^{(L)}(\boldsymbol{p}, F^{(d_k)}, \mathcal{P})$$

Combining this with the earlier result, we have the following upper bound on the number of I/O operations for $1 \le l \le k$:

$$T_l^{(L)}(\boldsymbol{p}, F^{(d)}, \mathcal{P}) \le \lceil d/d_k \rceil (d_k/d_l) 2^{d+1}$$

Since, as noted earlier, $d_r \ge \log(\min(s_r - 1, n))/4$, we obtain the desired upper bound on $T_l^{(L)}(\boldsymbol{p}, F^{(d)}, \mathcal{P})$ by combining this result with the bound on $n_l^{(k)}$ given above. ■

The above results are derived for standard MHG and the family of FFT graphs. We now strengthen these results in two ways when the I/O-limited MHG is used. First, the I/O limitation requires more time for a given amount of storage and, second, the lower bound we derive applies to every graph for the discrete Fourier transform, not just those for the FFT.

It is important to note that the efficient pebbling strategy used in the standard MHG makes extensive use of level-L pebbles on intermediate vertices of the FFT graph. When this is not allowed, the lower bound on the I/O time is much larger. Since the lower bounds for the standard and I/O-limited MHG on matrix multiplication are about the same, this illustrates that the DFT and matrix multiplication make dramatically different use secondary memory. (In the following theorem a **linear straight-line program** is a straight-line program in which the operations are additions and multiplications by constants.)

THEOREM 11.5.6 *Let $FFT(n)$ be* **any DAG** *associated with the DFT on n inputs when realized by a linear straight-line program. Let $FFT(n)$ be pebbled with strategy $\mathcal{P}$ in the* **I/O-limited MHG** *with resource vector $\mathbf{p}$ and let $s_l = \sum_{j=1}^{l} p_j$. If $S = s_{L-1} \le n$, then for each pebbling strategy $\mathcal{P}$, the computation and I/O time at level l must satisfy the following bounds:*

$$T_l^{(L)}(\boldsymbol{p}, FFT(n), \mathcal{P}) = \Omega\left(\frac{n^2}{S}\right) \quad \textit{for } 1 \le l \le L$$

Also, when $n = 2^d$, there is a pebbling $\mathcal{P}$ of the FFT graph $F^{(d)}$ such that the following relations hold simultaneously when $S \ge 2\log n$:

$$T_l^{(L)}(\boldsymbol{p}, F^{(d)}, \mathcal{P}) = \begin{cases} O\left(\frac{n^2}{S} + n\log S\right) & l = 1 \\ O\left(\frac{n^2}{S} + n\frac{\log S}{\log s_{l-1}}\right) & 2 \le l \le L \end{cases}$$

Proof The lower bound follows from Theorem 11.3.1 and Theorem 10.5.5. We show that the upper bounds can be achieved on $F^{(d)}$ under the I/O limitation simultaneously for $1 \leq l \leq L$.

The pebbling strategy meeting the lower bounds is based on that used in the proof of Theorem 10.5.5 to pebble $F^{(d)}$ using $S \leq 2^d + 1$ pebbles in the red pebble game. The number of level-1 pebble placements used in that pebbling is given in the statement of Theorem 10.5.5. A level-2 I/O operation occurs once on each of the 2^d outputs and 2^{d-e} times on each of the 2^d inputs of the bottom FFT subgraphs, for a total of $2^d(2^{d-e} + 1)$ times.

The pebbling for the L-level MHG is patterned after the aforementioned pebbling for the red pebble game, which is based on the decomposition of Lemma 6.7.4. (See Fig. 11.9.) Let e be the largest integer such that $S \geq 2^e + d - e$. Pebble the binary subtrees on 2^{d-e} inputs in the 2^e bottom subgraphs $F_{b,m}^{(d-e)}$ as follows: On an input vertex level-L pebbles are replaced by pebbles at all levels down to and including the first level. Then level-1 pebbles are advanced on the subtrees in the order that minimizes the number of level-1 pebbles in the red pebble game. It may be necessary to use pebbles at all levels to make these advances; however, each vertex in a subtree (of which there are $2^{d-e+1} - 1$) experiences at most two transitions at each level in the hierarchy. In addition, each vertex in a bottom tree is pebbled once with a level-1 pebble in a computation step. Therefore, the number of level-l transitions on vertices in the subtrees is at most $2^{d+1}(2^{d-e+1} - 1)$ for $2 \leq l \leq L$, since this pebbling of 2^e subtrees is repeated 2^{d-e} times.

Once the inputs to a given subgraph $F_{t,p}^{(e)}$ have been pebbled, the subgraph itself is pebbled in the manner indicated in Theorem 11.5.5, using $O(e2^e/\log s_{l-1})$ pebbles at each level l for $2 \leq l \leq L$. Since this is done for each of the 2^{d-e} subgraphs $F_{t,p}^{(e)}$, it follows that on the top FFT subgraphs a total of $O(e2^d/\log s_{l-1})$ level-l transitions occur, $2 \leq l \leq L$. In addition, each vertex in a graph $F_{t,p}^{(e)}$ is pebbled once with a level-1 pebble in a computation step.

It follows that at most

$$T_l^{(L)}(\boldsymbol{p}, F^{(d)}, \mathcal{P}) = O\left(2^d(2^{d-e+1} - 1) + \frac{e2^d}{\log s_{l-1}}\right)$$

level-l I/O operations occur for $2 \leq l \leq L$, as well as

$$T_1^{(L)}(\boldsymbol{p}, F^{(d)}, \mathcal{P}) = O(2^d(2^{d-e+1} - 1) + e2^d)$$

computation steps. It is left to the reader to verify that $2^e < 2^e + d - e \leq S < 2^{e+1} + d - e - 1 \leq 42^e$ when $e + 1 \geq \log d$ (this is implied by $S \geq 2d$), from which the result follows. ■

11.5.4 Convolution

The convolution function $f_{\text{conv}}^{(n,m)} : R^{n+m} \mapsto R^{n+m-1}$ over a commutative ring $\mathcal{R}$ (see Section 6.7.4) maps an n-tuple $\boldsymbol{a}$ and an m-tuple $\boldsymbol{b}$ onto an $(n + m - 1)$-tuple $\boldsymbol{c}$ and is denoted $\boldsymbol{c} = \boldsymbol{a} \otimes \boldsymbol{b}$. An efficient straight-line program for the convolution is described in Section 6.7.4 that uses the convolution theorem (Theorem 6.7.2) and the FFT algorithm. The convolution theorem in terms of the $2n$-point DFT and its inverse is

$$\boldsymbol{a} \otimes \boldsymbol{b} = F_{2n}^{-1}(F_{2n}(\boldsymbol{a}) \times F_{2n}(\boldsymbol{b}))$$

Obviously, when $n = 2^d$ the $2n$-point DFT can be realized by the $2n$-point FFT. The DAG associated with this algorithm, shown in Fig. 11.11 for $d = 4$, contains three copies of the FFT graph $F^{(2d)}$.

We derive bounds on the computation and I/O time in the standard and I/O-limited memory-hierarchy game needed for the convolution function using this straight-line program. For the standard MHG, we invoke the lower bounds and an efficient algorithm for the FFT. For the I/O-limited MHG, we derive new lower bounds based on those for two back-to-back FFT graphs as well as upper bounds based on the I/O-limited pebbling algorithm given in Theorem 11.5.4 for FFT graphs.

THEOREM 11.5.7 *Let $G^{(n)}_{\text{convolve}}$ be the graph of a straight-line program for the convolution of two n-tuples using the convolution theorem, $n = 2^d$. Let $G^{(n)}_{\text{convolve}}$ be pebbled in the standard MHG with the resource vector $\boldsymbol{p}$. Let $s_l = \sum_{j=1}^{l} p_j$ and let k be the largest integer such that $s_k \le n$. When $p_1 \ge 3$ there is a pebbling of $G^{(n)}_{\text{convolve}}$ for which the following bounds hold simultaneously:*

$$T_l^{(L)}(\boldsymbol{p}, F^{(d)}) = \begin{cases} \Theta(n \log n) & l = 1 \\ \Theta\left(\frac{n \log n}{\log s_{l-1}}\right) & 2 \le l \le k+1 \\ \Theta(n) & k+2 \le l \le L \end{cases}$$

Proof The lower bound follows from Lemma 11.3.2 and Theorem 11.5.5. From the former, it is sufficient to derive lower bounds for a subgraph of a graph. Since $F^{(d)}$ is contained in $G^{(n)}_{\text{convolve}}$, the lower bound follows.

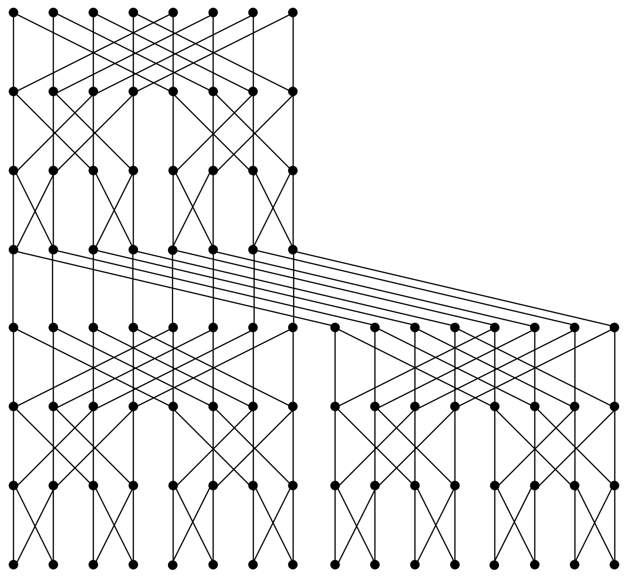


Figure 11.11 A DAG for the graph of the convolution theorem on $n = 8$ inputs.

The upper bound follows from Theorem 11.5.5. We advance level-L pebbles to the outputs of each of the two bottom FFT graphs $F^{(2d)}$ in Fig. 11.11 and then pebble the top FFT graph. The number of I/O and computation steps used is triple that used to pebble one such FFT graph. In addition, we perform $O(n)$ I/O and computation steps to combine inputs to the top FFT graph. ■

The bounds for the I/O-limited version of the MHG for the convolution problem are considerably larger than those for the standard MHG. They have a much stronger dependence on S and n than do those for the FFT graph.

THEOREM 11.5.8 *Let $H^{(n)}_{\text{convolve}}$ be the graph of any DAG for the convolution of two n-tuples using the convolution theorem, $n = 2^d$. Let $H^{(n)}_{\text{convolve}}$ be pebbled in the I/O-limited MHG with the resource vector $\boldsymbol{p}$ and let $s_l = \sum_{j=1}^{l} p_j$. If $S = s_{L-1} \leq n$, then the time to pebble $H^{(n)}_{\text{convolve}}$ at the lth level, $T_l^{(L)}(\boldsymbol{p}, H^{(n)}_{\text{convolve}})$, satisfies the following lower bounds simultaneously for $1 \leq l \leq L$:*

$$T_l^{(L)}(\boldsymbol{p}, H^{(n)}_{\text{convolve}}) = \Omega\left(\frac{n^3}{S^2}\right)$$

when $S \leq n/\log n$.

Proof A lower bound is derived for this problem by considering a generalization of the graph shown in Fig. 11.11 in which the three copies of the FFT graph $F^{(2d)}$ are replaced by an arbitrary DAG for the DFT. This could in principle yield in a smaller lower bound on the time to pebble the graph. We then invoke Lemma 11.3.2 to show that a lower bound can be derived from a reduction of this new graph, namely, that consisting of two back-to-back DFT graphs obtained by deleting one of the bottom FFT graphs. We then derive a lower bound on the time to pebble this graph with the red pebble game and use it together with Theorem 11.3.1 to derive the lower bounds mentioned above.

Consider pebbling two back-to-back DAGs for the DFT on n inputs, n even, in the red pebble game. From Lemma 10.5.4, the n-point DFT function is $(2, n, n, n/2)$-independent. From the definition of the independence property (see Definition 10.4.2), we know that during a time interval in which $2(S + 1)$ of the n outputs of the second DFT DAG on n-inputs are pebbled, at least $n/2 - 2(S + 1)$ of its inputs are pebbled. In a back-to-back DFT graph these inputs are also outputs of the first DFT graph. It follows that for each group of $2(S + 1)$ of these $n/2 - 2(S + 1)$ outputs of the first DFT DAG, at least $n/2 - 2(S + 1)$ of its inputs are pebbled. Thus, to pebble a group of $2(S + 1)$ outputs of the second FFT DAG (of which there are at least $\lfloor n/(2(S + 1))\rfloor$ groups), at least $\lfloor (n/2 - 2(S + 1))/2(S + 1)\rfloor(n/2 - 2(S + 1))$ inputs of the first DFT must be pebbled. Thus, $T_l^{(L)}(\boldsymbol{p}, H^{(n)}_{\text{convolve}}) \geq n^3/(64(S + 1)^2)$, since it holds both when $S \leq n/4\sqrt{2}$ and when $S > n/4\sqrt{2}$.

Let's now consider a pebbling strategy that achieves this lower bound up to a multiplicative constant. The pebbling strategy of Theorem 11.5.5 can be used for this problem. It represents the FFT graph $F^{(d)}$ as a set of FFT graphs $F^{(e)}$ on top and a set of FFT graphs $F^{(d-e)}$ on the bottom. Outputs of one copy of $F^{(e)}$ are pebbled from left to right. This requires pebbling inputs of $F^{(d)}$ from left to right once. To pebble all outputs of $F^{(d)}$, 2^{d-e} copies of $F^{(e)}$ are pebbled and the 2^d inputs to $F^{(d)}$ are pebbled 2^{d-e} times.

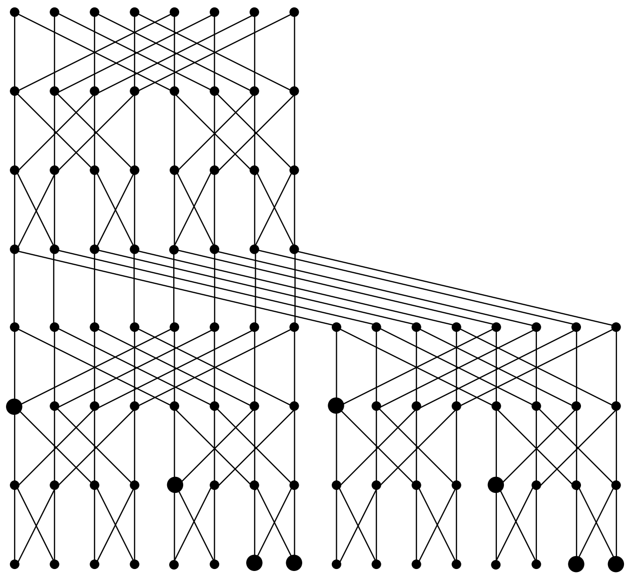


Figure 11.12 An I/O-limited pebbling of a DAG for the convolution theorem showing the placement of eight pebbles.

Consider the graph $G^{(n/2)}_{\rm convolve}$ consisting of three copies of $F^{(d)}$, two on the bottom and one on top, as shown in Fig. 11.12. Using the above strategy, we pebble the outputs of the two bottom copies of $F^{(d)}$ from left to right in parallel a total of 2^{d-e} times. The outputs of these two graphs are pebbled in synchrony with the pebbling of the top copy of $F^{(d)}$. It follows that the number of I/O and computation steps used on the bottom copies of $F^{(d)}$ in $G^{(n/2)}_{\rm convolve}$ is $2(2^{d-e})$ times the number on one copy, with twice as many pebbles at each level plus the number of such steps on the top copy of $F^{(d)}$. It follows that $G^{(n/2)}_{\rm convolve}$ can be pebbled with three times the number of pebbles at each level as can $F^{(d)}$, with $O(2^{d-e})$ times as many steps at each level. The conclusion of the theorem follows from manipulation of terms. ■

The bounds given above also apply to some permutation and merging networks. Since, as shown in Section 6.8, the graph of Batcher's bitonic merging network is an FFT graph, the bounds on I/O and computation time given earlier for the FFT also apply to it. Also, as shown in Section 7.8.2, since a permutation network can be constructed of two FFT graphs connected back-to-back, the lower bounds for convolution apply to this graph. (See the proofs of Theorems 11.5.7 and 11.5.8.) The same order-of-magnitude upper bounds follow from constructions that differ only in details from those given in these theorems.

11.6 Block I/O in the MHG

Many memory units move data in large blocks, not in individual words, as generally assumed in the above sections. (Note, however, that one pebble can carry a block of data.) Data is moved in blocks because the time to fetch one word and a block of words is typically about the

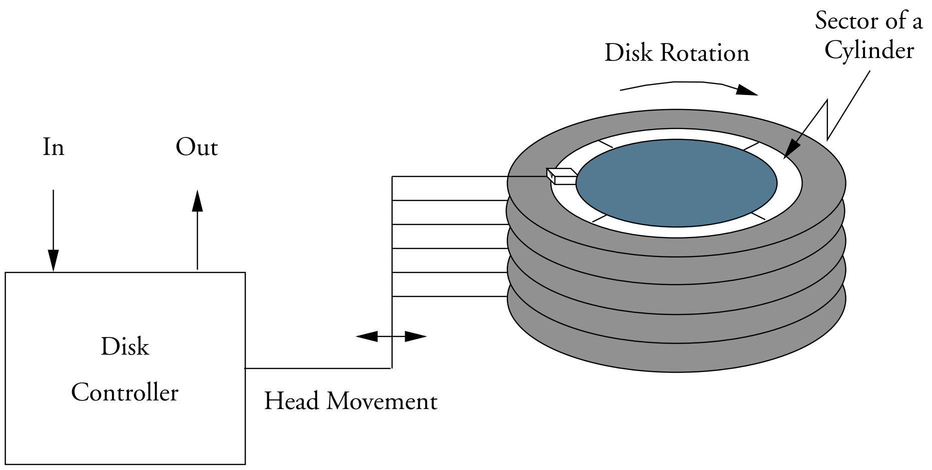


Figure 11.13 A disk unit with three platters and two heads per disk. Each track is divided into four sectors and heads move in and out on a common arm. The memory of the disk controller holds the contents of one track on one disk.

same. Figure 11.13 suggests why this is so. A disk spinning at 3,600 rpm that has 40 sectors per track and 512 bits per sector (its block size) requires about 10 msec to find data in the track under the head. However, the time to read one sector of 64 bytes (512 bits) is just .42 msec.

To model this phenomenon, we assume that the time to access k disk sectors with consecutive addresses is $\alpha + k\beta$, where α is a large constant and β is a small one. (This topic is also discussed in Section 7.3.) Given the ratio of α to β, it makes sense to move data to and from a disk in blocks of size about equal to the number of bytes on a track. Some operating systems move data in track-sized blocks, whereas others move them in smaller units, relying upon the fact that a disk controller typically keeps the contents of its current track in a fast random-access memory so that successive sector accesses can be done quickly.

The gross characteristics of disks described by the above assumption hold for other storage devices as well, although the relative values of the constants differ. For example, in the case of a tape unit, advancing the tape head to the first word in a consecutive sequence of words usually takes a long time, but successive words can be read relatively quickly.

The situation with **interleaved random-access memory** is similar, although the physical arrangement of memory is radically different. As depicted in Fig. 11.14, an interleaved random-access memory is a collection of 2^r memory modules, $r \geq 1$, each containing 2^k b-bit words. Such a memory can simulate a single 2^{r+k}-word b-bit random-access memory. Words with addresses $0, 2^r, 2\,2^r, 3\,2^r, \ldots, 2^{k-1}2^r$ are stored in the first module, words with addresses $1, 2^r+1, 2\,2^r+1, 3\,2^r+1, \ldots, 2^{k-1}2^r+1$ in the second module, and words with addresses $2^r-1, 2\,2^r-1, 3\,2^r-1, 4\,2^r-1, \ldots, 2^{r+k}-1$ in the last module.

To access a word in this memory, the high order k bits are provided to each module. If a set of words is to be read, the words with these common high-order bits are copied to the registers. If a set of words is to be written, new values are copied from the registers to them.

When an interleaved memory is used to simulate a much faster random-access memory, a CPU writes to or reads from the 2^r registers serially, whereas data is transferred in parallel between the registers and the modules. The use of two sets of registers (**double buffering**)

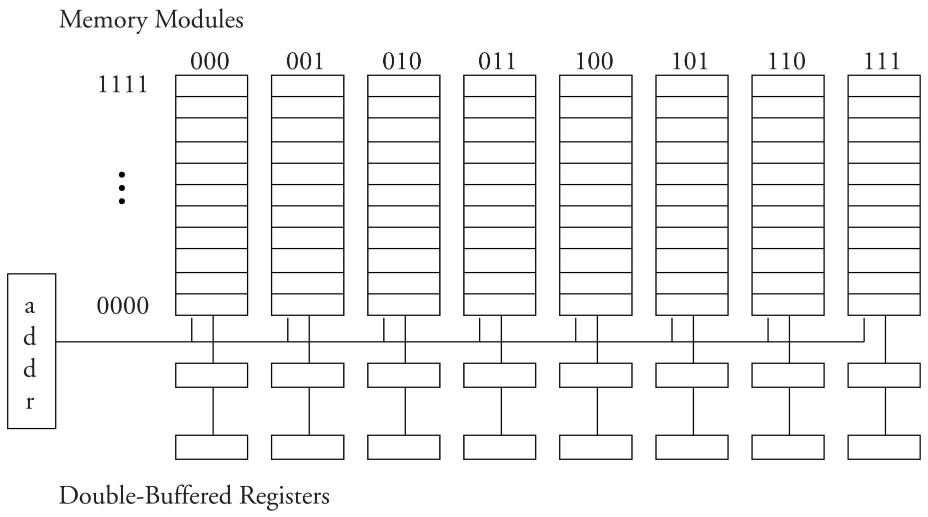


Figure 11.14 Eight interleaved memory modules with double buffering. Addresses are supplied in parallel while data is pipelined into and out of the memory.

allows the register sets to be alternated so that data can be moved continuously between the CPU and the modules. This allows the interleaved memory to be about 2^r times slower than the CPU and yet, with a small set of fast registers, appear to be as fast as the CPU. This works only if the program accessing memory does not branch to a new set of words. If it does, the startup time to access a new word is about 2^r times the CPU speed. Thus, an interleaved random-access memory also requires time of the form $\alpha + k\beta$ to access k words. For example, for a moderately fast random-access chip technology α might be 80 nanoseconds whereas β might be 10 nanoseconds, a ratio of 8 to 1.

This discussion justifies assuming that the time to move k words with consecutive addresses to and from the lth unit in the memory hierarchy is $\alpha_l + k\beta_l$ for positive constants α_l and β_l, where α_l is typically much larger than β_l. If $k = b_l = \lceil \alpha_l/\beta_l \rceil$, then $\alpha_l + k\beta_l \approx 2\alpha_l$ and the time to retrieve one item and b_l items is about the same. Thus, efficiency dictates that items should be fetched in blocks, especially if all or most of the items in a block can be used if one of them is used. This justifies the block-I/O model described below. Here we let t_l be the time to move a block at level l. We add the requirement that data stored together be retrieved together to reflect physical constraints existing in practice.

DEFINITION 11.6.1 (Block-I/O Model) *At the lth level in a memory hierarchy, I/O operations are performed on blocks. The block size and the time in seconds to access a block at the lth level are b_l and t_l, respectively. For each l, b_l/b_{l-1} is an integer. In addition, any data written as part of a block at level l must be read into level $l - 1$ by reading the entire block in which it was stored.*

The lower bounds on the number of I/O steps given in Section 11.5 can be generalized to the block-I/O case by dividing the number of I/O operations by the size b_l of blocks moving between levels $l - 1$ and l. This lower bound can be achieved for matrix-vector and matrix-matrix multiplication because data is always written to and read from the higher-level memory in the same way for these problems. (See Problems 11.13 and 11.14.)

For the FFT graph in the standard MHG, instead of pebbling FFT subgraphs on 2^{d_r} inputs, we pebble b_l FFT subgraphs on $2^{d_r}/b_l$ inputs (assuming that b_l is a power of 2). Doing so allows all the data moving back and forth in blocks between memories to be used and accommodates the transposition mentioned at the beginning of Section 11.5.3. This provides an upper bound of $O(n \log n/(b_{l-1} \log(s_{l-1}/b_{l-1})))$ on the I/O time at level l. Clearly, when b_{l-1} is much smaller than s_{l-1}, say $b_{l-1} = O(\sqrt{s_{l-1}})$, the upper and lower bounds match to within a multiplicative factor. (This follows because we divide n by b_{l-1} and $\log b_{l-1} = O(\log s_{l-1})$.) These observations apply to the FFT-based problems as well.

11.7 Simulating a Fast Memory in the MHG

In this section we revisit the discussion of Section 11.1.2, taking into account that a memory hierarchy may have many levels and that data is moved in blocks.

We ask the question, "How do we assess the effectiveness of a memory hierarchy on a particular problem?" For several problems we have upper and lower bounds on their number of computation and I/O steps in memory hierarchies parameterized by block sizes and numbers of storage locations. If we add to this mix the time to move a block between levels, we can derive bounds on the time for all computation and I/O steps. We then ask under what conditions this time is the best possible. Since data must typically be stored and retrieved from archival memory, we cannot expect the performance to exceed that of a two-level hierarchy (modeled by the red-blue pebble game) in which all the available storage locations, except for those in the archival memory, are in first-level storage. For this reason we use the two-level memory as our reference model. We now define these terms and state a condition for optimality of a pebbling strategy.

For $1 \leq l \leq L-1$ we let t_l be the time to move one block of b_l words between levels $l-1$ and l of a memory hierarchy, measured as a multiple of the time to perform one computation step. Thus, the time for one computation step is $t_1 = 1$.

Let $\mathcal{P}$ be a pebbling strategy for a graph G in the L-level MHG that uses the resource vector $\boldsymbol{p} = (p_1, p_2, \dots, p_{L-1})$ (p_l pebbles are used at the lth level) and moves data in blocks of size specified by $\boldsymbol{b} = (b_2, b_3, \dots, b_L)$ (b_l words are moved between levels $(l-1)$ and l). Let $T_l^{(L)}(\boldsymbol{p}, \boldsymbol{b}, G)$ denote the number of level-l I/O operations with $\mathcal{P}$ on G. We define the **time for the pebbling strategy $\mathcal{P}$**, $T(\mathcal{P}, G)$ on the graph G as

$$T(\mathcal{P}, G) = \sum_{l=1}^{L} t_l \cdot T_l^{(L)}(\boldsymbol{p}, \boldsymbol{b}, G)$$

Thus, $T(\mathcal{P}, G)$ measures the absolute time expended to pebble a graph relative to the time to perform one computation step under the assumption that I/O operations cannot be overlapped.

From the above discussion, a pebbling is efficient if $T(\mathcal{P}, G)$ is at most some small multiple of $T_1^{(2)}(s_{L-1}, G)$, the normalized time to pebble G in the red-blue pebble game when all the pebbles at level $L-1$ or less in the MHG (there are s_{L-1} such pebbles) are used as if they were red pebbles.

A two-level computation exhibits **locality of reference** if it is likely in the near future to refer to words currently in its primary memory. Such computations perform fewer I/O operations than those that don't meet this condition. This idea extends to multiple levels: a multi-level memory hierarchy exhibits locality of reference if it uses its higher-level memory units much less often that its lower-level units. Formally, we say that a pebbling strategy $\mathcal{P}$ is **c-local** if $T(\mathcal{P}, G)$ satisfies the following inequality:

$$\sum_{l=1}^{L} t_l \cdot T_l^{(L)}(\boldsymbol{p}, \boldsymbol{b}, G, \mathcal{P}) \leq c\, T_1^{(2)}(s_{L-1}, G)$$

The definition of a c-local pebbling strategy is illustrated by the results for matrix multiplication in the standard MHG when block I/O is not used. Let k be the largest integer such that $s_k \leq 3n^2$. From Theorem 11.5.3 for matrix-matrix multiplication, we see that there exists an optimal pebbling if

$$\sum_{l=2}^{k} \frac{t_l}{b_l \sqrt{s_{l-1}}} + \sum_{l=k+1}^{L} \frac{t_l}{n b_l} \leq c^* \tag{11.1}$$

for some $c^* > 0$ since $T_1^{(2)}(S, G) = \Theta(n^3)$.

We noted in Section 11.1.2 that the imbalance between the computation and I/O times for matrix multiplication is becoming ever more serious with the advance of technology. We re-examine this issue in light of the above condition. Consider the case in which $k + 1 = L$; that is, the highest-level memory is used to store the arguments and results of a computation. In this case the second term on the left-hand side of (11.1) is a relative measure of the time to bring data into lower-level memories from the highest-level memory. It is negligible when nb_L is large. For example, if t_L = 2,000,000 and b_L = 10,000, say, then n must be at least 200, a modest-sized matrix. The first term on the left-hand side reflects the number of times data moves between the levels of the hierarchy holding the data. It is small when $b_l\sqrt{s_{l-1}}$ is large by comparison with t_l for $2 \leq l \leq k$, a condition that is not hard to meet. For example, if $s_{l-1} = 32 \times 10^6$ (about 4 Mbytes) and b_l = 1,000, then t_l must be less than about 45, a condition that certainly applies to low level memories such as today's random-access memories. Problems 11.15 and 11.16 provide opportunities to explore this issue with the FFT and convolution.

11.8 RAM-Based I/O Models

The MHG assumes that computations are done by pebbling the vertices of a directed acyclic graph. That is, it assumes that computations are straight-line. While the best known algorithms for the problems studied earlier in this chapter are straight-line, some problems are not efficiently done in a straight-line fashion. For example, binary search in a tree that holds a set of keys in sorted order (see Section 11.9.1) is much better suited to data-dependent computation of the kind allowed by an unrestricted RAM. Similarly, the merging of two sorted lists can be done more efficiently on a RAM than with a straight-line program. For this reason we consider RAM-based I/O models, specifically the block-transfer model and the hierarchical memory model.

11.8.1 The Block-Transfer Model

The block-transfer model is a two-level I/O model that generalizes the red-blue pebble game to RAM-based computations by allowing programs that are not straight-line.

DEFINITION 11.8.1 *The* **block-transfer model** *(BTM) is a serial computer in which a CPU is attached to an M-word primary memory and to a secondary memory of unlimited size that stores words in blocks of size B. Words are moved in blocks between the memories and words that leave primary memory in one block must return in that block. An* **I/O operation** *is the movement of a block to or from secondary memory. The* **I/O time** *with the BTM is the number of I/O operations.*

The secondary memory in the BTM can be a main memory if the primary memory is a cache, or can be a disk if the primary memory is a random-access memory. In fact, it can model I/O operations between any two devices. Since a block can be viewed as the contents of one track of a disk, the time to retrieve any word on the track is comparable to the time to retrieve the entire track. (See Section 11.6.) Since data is moved in blocks in the BTM, it makes sense to define simple I/O operations.

DEFINITION 11.8.2 *An I/O operation in the BTM is* **simple** *if, after a block or word is copied from one memory to the other, the copy in the first memory is deleted.*

Simple I/O operations for the pebble game are defined in Problem 11.10. In this problem the reader is asked to show that replacing all I/O operations with simple I/O operations has the effect of at most doubling the number of I/O operations. The proof of this fact applies equally well to the BTM.

We illustrate the use of the block-transfer model by examining the sorting problem. We derive a lower bound on the I/O time for all sorting algorithms and exhibit a sorting algorithm that meets the lower bound, up to a constant multiplicative factor. To derive the lower bound, we limit the range of sorting algorithms to those based on the comparison of keys, as stated below. (Sorting algorithms that are not comparison-based, such as the various forms of **radix sort**, assume that keys consist of individual digits and that digits are used to classify keys.)

ASSUMPTION 11.8.1 *All words to be sorted are located initially in the secondary memory. The* **compare-exchange operation** *is the only operation available to implement sorting algorithms on the BTM. In addition, an arbitrary permutation of the contents of the primary memory of the BTM can be done during the time required for one I/O operation.*

The assumption that the CPU can perform an arbitrary permutation on the contents of the primary memory during one I/O operation acknowledges that I/O operations take a very long time relative to CPU instructions.

Algorithms consistent with these assumptions are described by the multiway decision trees discussed below. They are a generalization of the **binary decision tree**, a binary tree in which each vertex has associated with it a comparison between two variables. For example, if keys x_1 and x_2 are compared at the root vertex, the comparison has two outcomes, namely $x_1 < x_2$ or $x_1 \geq x_2$, which are associated with the subtrees to the left and right of the root, respectively. Similar comparisons and outcomes are possible at each vertex of these two subtrees. A sequence of comparisons terminates on a leaf node.

Since a binary decision tree captures each of the data-dependent comparisons between keys in comparison-based sorting algorithm, each leaf is associated with the permutation of the original sequence of variables that puts the sequence into sorted order. Thus, a binary decision tree for sorting must have at least $n!$ distinct leaves, one for every permutation of n items. The length of a path through a binary decision tree is the number of comparisons performed on the particular input, and the length of the longest path is a measure of the worst-case number of comparisons. A binary tree with N leaves has a longest path of length at least $\log_2 N$ because if it were smaller, it would have fewer than $2^{\log_2 N} < N$ leaves. Since the length of the longest path is an integer, it must be at least $\lceil \log_2 N \rceil$. We summarize this result as a lemma that uses the lower bound on $n!$ given in Problem 2.23.

LEMMA 11.8.1 *The length of the longest path in a binary decision tree that sorts n inputs is at least $\lceil \log_2 n! \rceil = \Theta(n \log n)$.*

The **multiway decision tree** in Fig. 11.15 extends the above concept by permitting multiple comparisons at each vertex. 2^k outcomes are possible if k comparisons of variable pairs are associated with each vertex.

THEOREM 11.8.1 *Let B divide M and M divide n. Under Assumption 11.8.1 on the BTM, in the worst case the number of block I/O steps to sort a set of n records using M words of primary memory and block size B, $T_{\mathrm{BTMsort}}(n)$, satisfies the following bounds for $B \leq M/2$ and M large:*

$$T_{\mathrm{BTMsort}}(n) = \Theta\left(\max\left[\frac{n}{B}, \frac{(n/B)\log(n/B)}{\log(M/B)}\right]\right)$$

Proof Let's now apply the multiway decision tree to the BTM. Since each path in such a tree corresponds to a sequence of comparisons by the CPU, the tree must have at least $n!$ leaves. To complete the lower-bound derivation we need to determine the number of descendants of vertices in the multiway tree.

Initially the n unsorted words are stored in n/B blocks in the secondary memory. The first time one of these blocks is moved to the primary memory, up to $B!$ permutations can be performed on the words in it. No more permutations are possible between these words no matter how many times they are simultaneously in primary memory, even if they return to the memory as members of different blocks. When a block of B words arrives in the M-word memory, the number of possible permutations between them (given that the order among the $M - B$ words originally in the memory has previously been taken into

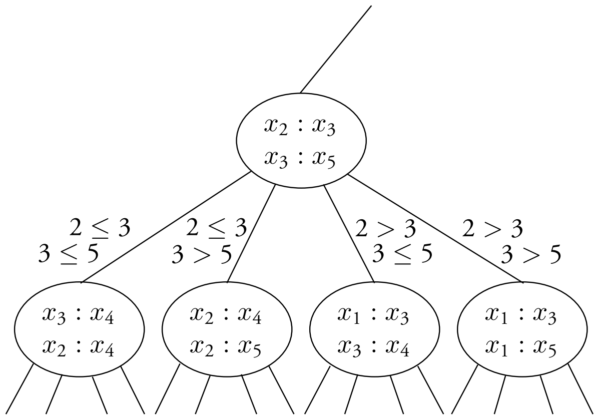


Figure 11.15 A multiway decision tree in which multiple comparisons of keys are made at each vertex.

account, as has the order among the B words in a block) is at most $\rho = \binom{M}{B}$, the binomial coefficient. (To see this, observe that places for the B new (and indistinguishable) words in the primary memory can be any B of the M indistinguishable places.) It follows that the multi-comparison decision tree for every BTM comparison-based sorting algorithm on the BTM has at most n/B vertices with at most $\rho B!$ possible outcomes (vertices corresponding to the first arrival of one of the blocks in primary memory) and that each of the other vertices has at most ρ outcomes.

It follows that if a sorting algorithm executes $T_{\text{BTMsort}}(n)$ block I/O steps, the function $T_{\text{BTMsort}}(n)$ must satisfy the following inequality:

$$(B!)^{n/B}\binom{M}{B}^{T_{\text{BTMsort}}(n)} \geq n!$$

Using the approximation to $n!$ given in Lemma 11.8.1, the upper bound of $(M/B)^B e^B$ on $\binom{M}{B}$ derived in Lemma 10.12.1, and the fact that $T \geq n/B$, we have the desired conclusion.

An upper bound is obtained by extending the standard merging algorithm to blocks of keys. The merging algorithm is divided into phases, an initialization phase and merging phases, each of which takes $(2n/B)$ I/O operations. In the initialization phase, a set of n/M sorted sublists of M keys or M/B blocks is formed by bringing groups of M keys into primary memory, sorting, and then writing them out to secondary memory. In a merging phase, M/B sorted sublists of L blocks ($L = M/B$ in the first merging phase) are merged into one sorted sublist of ML/B blocks, as suggested in Fig. 11.16. The first block of keys (those with the smallest values) in each sublist is brought into memory and the B smallest keys in this set is written out to the new sorted sublist that is being constructed. If any block from an input sublist is depleted, the next block from that list is brought in. There is always sufficient space in primary memory to do this. Thus, after k phases the sorted sublists contain $(M/B)^k$ blocks. When $(M/B)^k \geq n/B$, the merging is done. Thus, $(2n/B)\lceil \log_2(n/B)/\log_2(M/B)\rceil$ I/O operations are performed by this algorithm. ■

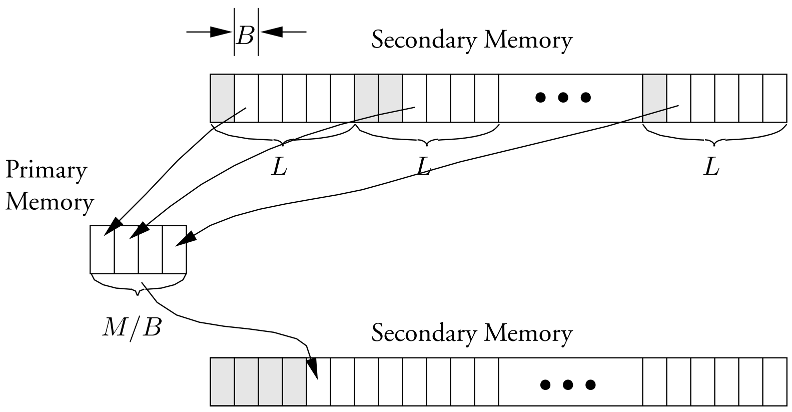


Figure 11.16 The state of the block merging algorithm after merging four blocks. The algorithm merges M/B sublists, each containing L blocks of B keys.

Similar results can be obtained for the permutation networks defined in Section 7.8.2 (see Problem 11.18), the FFT defined in Section 6.7.3 (see Problem 11.19), and matrix transposition defined in Section 6.5.4 (see [9]).

11.9 The Hierarchical Memory Model

In this section we define the hierarchical memory model and derive bounds on the time to do matrix multiplication, the FFT and binary search in this model. These results provide another opportunity to evaluate the performance of memory hierarchies, this time with a single cost function applied to memory accesses at all levels of a hierarchy. We make use of lower bounds derived earlier in this chapter.

DEFINITION 11.9.1 *The* **hierarchical memory model** *(HMM) is a serial computer in which a CPU without registers is attached to a random-access memory of unlimited size for which the time to access location a for reading or writing is the value of a monotone nondecreasing* **cost function** $\nu(a) : \mathbb{N} \mapsto \mathbb{N}$ *from the integers* $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ *to* $\mathbb{N}$. *The* **cost of computing** $f^{(n)} : \mathcal{A}^n \mapsto \mathcal{A}^m$ **with the HMM** *using the cost function $\nu(a)$, $\mathcal{K}_\nu(f)$, is defined as*

$$\mathcal{K}_\nu(f) = \max_{\boldsymbol{x}} \sum_{j=1}^{T(\boldsymbol{x})} \nu(a_j) \tag{11.2}$$

where a_j, $1 \leq j \leq T(\boldsymbol{x})$, is the address accessed by the CPU on the jth computational step and $T(\boldsymbol{x})$ is the number of steps when the input is $\boldsymbol{x}$.

The HMM with cost function $\nu(a) = 1$ is the standard random-access machine described in Section 3.4. While in principle the HMM can model many of the details of the MHG, it is more difficult to make explicit the dependence of $\nu(a)$ on the amount of memory at each level in the hierarchy as well as the time for a memory access in seconds at that level. Even though the HMM can model programs with branching and looping, following [7] we assume straight-line programs when studying the FFT and matrix-matrix multiplication problems with this model.

Let $n(f, \boldsymbol{x}, a)$ be the number of times that address a is accessed in the HMM for f on input $\boldsymbol{x}$. It follows that the cost $\mathcal{K}_\nu(f)$ can be expressed as follows:

$$\mathcal{K}_\nu(f) = \max_{\boldsymbol{x}} \sum_{1 \leq a} n(f, \boldsymbol{x}, a)\nu(a) \tag{11.3}$$

Many cost functions have been studied in the HMM, including $\nu(a) = \lceil \log_2 a \rceil$, $\nu(a) = a^\alpha$, and $\nu(a) = U_m(a)$, where $U_m(a)$ is the following threshold function with threshold m:

$$U_m(a) = \begin{cases} 1 & a \geq m \\ 0 & \text{otherwise} \end{cases}$$

It follows that

$$\mathcal{K}_{U_m}(f) = \max_{\boldsymbol{x}} \sum_{m \leq a} n(f, \boldsymbol{x}, a)$$

For the matrix-matrix multiplication and FFT problems, the cost $\mathcal{K}_{U_m}(f)$ of computing f is directly related to the number of I/O operations with the red-blue pebble game played with $S = m$ red pebbles discussed in Sections 11.5.2 and 11.5.3. For this reason we call this cost **I/O complexity**. The principal difference is that in the HMM no cost is assessed for data stored in the first m memory locations.

Let the differential cost function $\Delta\nu(a)$ be defined as

$$\Delta\nu(a) = \nu(a) - \nu(a-1)$$

As a consequence, we can write $\nu(a)$ as follows if we set $\nu(-1) = 0$:

$$\nu(a) = \sum_{0 \le b \le a} \Delta\nu(b) \tag{11.4}$$

Since $\nu(a)$ is a monotone nondecreasing function, $\Delta\nu(m)$ is nonnegative.

Rewriting (11.3) using (11.4), we have

$$\begin{aligned}
\mathcal{K}_\nu(f) &= \max_{\boldsymbol{x}} \sum_{1 \le a} n(f, \boldsymbol{x}, a) \sum_{0 \le b \le a} \Delta\nu(b) \\
&= \left[\max_{\boldsymbol{x}} \sum_{c=0}^{\infty} \Delta\nu(c) \sum_{d=c}^{\infty} n(f, \boldsymbol{x}, d) \right] \\
&= \sum_{c=0}^{\infty} \Delta\nu(c) \left[\max_{\boldsymbol{x}} \sum_{d=c}^{\infty} n(f, \boldsymbol{x}, d) \right] \qquad (11.5) \\
&= \sum_{c=0}^{\infty} \Delta\nu(c) \mathcal{K}_{U_c}(f)
\end{aligned}$$

11.9.1 Lower Bounds for the HMM

Before deriving bounds on the cost to do a variety of tasks in the HMM, we introduce the binary search problem.

A **binary tree** is a tree in which each vertex has either one or two descendants except leaf vertices, which have none. (See Fig. 11.17.) Also, every vertex except the root vertex has one

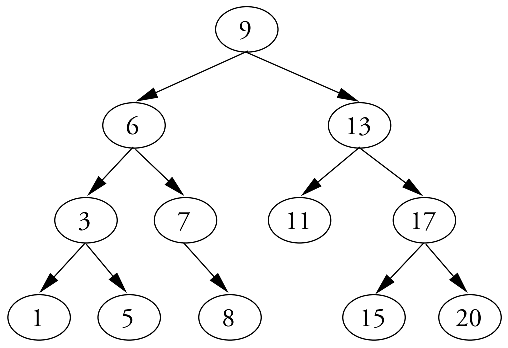


Figure 11.17 A binary search tree.

parent vertex. The **length of a path** in a tree is the number of edges on that path. The **left (right) subtree** of a vertex is the subtree that is detached by removing the left (right) descending edge. A **binary search tree** is a binary tree that has one key at each vertex. (This definition assumes that all the keys in the tree are distinct.) The value of this one key is larger than that of all keys in the left subtree, if any, and smaller than all keys in the right subtree, if any. A **balanced binary search tree** is a binary search tree in which all paths have length k or $k+1$ for some integer k.

LEMMA 11.9.1 *The length of the longest path in a binary tree with n vertices is at least $\lceil \log_2(n+1)/2 \rceil$.*

Proof A longest path in a binary tree with n vertices is smallest when all levels in the tree are full except possibly for the bottom level. If such a tree has a longest path of length l, it has between 2^l and $2^{l+1}-1$ vertices. It follows that the longest path in a binary search tree containing n keys is at least $\lceil \log_2(n+1)/2 \rceil$. ■

The **binary search procedure** searches a binary search tree for a key value v. It compares v against the root value, stopping if they are equal. If they are not equal and v is less than the key at the root, the search resumes at the root vertex of the left subtree. Otherwise, it resumes at the root of the right subtree. The procedure also stops when a leaf vertex is reached.

We can now state bounds on the cost on the HMM for the logarithmic cost function $\nu(a) = \lceil \log_2 a \rceil$. This function applies when the memory hierarchy is organized as a binary tree in which the low-indexed memory locations are located closest to the roots and the time to retrieve an item is proportional to the number of edges between it and the root. We use it to illustrate the techniques developed in the previous section.

Theorem 11.9.1 states lower performance bounds for straight-line algorithms. Thus, the computation time is independent of the particular argument of the function f provided as input. Matching upper bounds are derived in the following section. (The logarithmic cost function is polynomially bounded.)

THEOREM 11.9.1 *The cost function $\nu(a) = \lceil \log_2 a \rceil$ on the HMM for the $n \times n$ matrix multiplication function $f^{(n)}_{A\times B}$ realized by the classical algorithm, the n-point FFT associated with the graph $F^{(d)}$, $n = 2^d$, comparison-based sorting on n keys $f^{(n)}_{\text{sort}}$, and binary search on n keys, $f^{(n)}_{\text{BS}}$, satisfies the following lower bounds:*

Matrix multiplication:	$\mathcal{K}_\nu(f^{(n)}_{A\times B}) = \Omega(n^3)$
Fast Fourier transform:	$\mathcal{K}_\nu(F^{(d)}) = \Omega(n \log n \log\log n)$
Comparison-based sorting:	$\mathcal{K}_\nu(f^{(n)}_{\text{sort}}) = \Omega(n \log n \log\log n)$
Binary search:	$\mathcal{K}_\nu(f^{(n)}_{\text{BS}}) = \Omega(\log^2 n)$

Proof The lower bounds for the logarithmic cost function $\nu(a) = \lceil \log_2 a \rceil$ use the fact that $\Delta\nu(a) = 1$ when $a = 2^k$ for some integer k but is otherwise 0. It follows from (11.5) that

$$\mathcal{K}_\nu(f) = \sum_{k=1}^{t} \mathcal{K}_{U_{2^k}}(f) \tag{11.6}$$

for the task characterized by f, where t satisfies $2^t \leq N$ and N is the space used by task. $N = 2n^2$ for $n \times n$ matrix multiplication, $N = n$ for the FFT graph $F^{(d)}$, and $N = n$ for binary search.

In Theorem 11.5.3 it was shown that the number of I/O operations to perform $n \times n$ matrix multiplication with the classical algorithm is $\Omega(n^3/\sqrt{m})$. The model of this theorem assumes that none of the inputs are in the primary memory, the equivalent of the first m memory locations in the HMM.

Since no charge is assessed by the $U_m(a)$ cost function for data in the first m memory locations, a lower bound on cost with this measure can be obtained from the lower bound obtained with the red-blue pebble game by subtracting m to take into account the first m I/O operations that need not be performed.

Thus for matrix multiplication, $\mathcal{K}_{U_m}(f^{(n)}_{A\times B}) = \Omega\left((n^3/\sqrt{m}) - m\right)$. Since

$$\left(n^3/\sqrt{m}\right) - m \geq (\sqrt{8} - 1)n^3/\sqrt{8m}$$

when $m \leq n^2/2$, it follows from (11.6) that $\mathcal{K}_\nu(f^{(n)}_{A\times B}) = \Omega(n^3)$ because $\sum_{k=0}^{t} n^3/2^k = \Omega(n^3)$.

For the same reason, $\mathcal{K}_{U_m}(F^{(d)}) = \Omega\left((n \log n)/\log m - m\right)$ (see Theorem 11.5.5) and $(n \log n/\log m) - m \geq n \log n/(2 \log m)$ for $m \leq n/2$. It follows that $\mathcal{K}_\nu(F^{(d)})$ satisfies

$$\begin{aligned}\mathcal{K}_\nu(F^{(d)}) &= \Omega\left(\sum_k \frac{n \log n}{\log(2^k)}\right) \\ &= \Omega\left(\sum_{k=1}^{\log n} \frac{n \log n}{k}\right) = \Omega\left(n \log n \log\log n\right)\end{aligned}$$

The last equation follows from the observation that $\sum_{k=1}^{p} 1/k$ is closely approximated by $\int_1^p \frac{1}{x}\, dx$, which is $\ln p$. (See Problem 11.2.)

The lower bound for comparison-based sorting uses the $\Omega(n \log n/\log m)$ sorting lower bound for the BTM with a block size $B = 1$. Since the BTM assumes that no data are resident in the primary memory before a computation begins, the lower bound for the HMM cost under the U_m cost function is $\Omega\left((n \log n/\log m) - m\right)$. Thus, the FFT lower bound applies in this case as well.

Finally, we show that the lower bound for binary search is $\mathcal{K}_{U_m}(f^{(n)}_{\mathrm{BS}}) = \Omega(\log n - \log m)$. Each path in the balanced binary search tree has length $d = \lceil \log(n+1)/2 \rceil$ or $d - 1$. Choose a query path that visits the minimum number of variables located in the first m memory locations. To make this minimum number as large as possible, place the items in the first m memory locations as close to the root as possible. They will form a balanced binary subtree of path length $l = \lceil \log_2(m+1)/2 \rceil$ or $l - 1$. Thus no full path will have more than l edges and $l - 1$ variables from the first m memory locations. It follows that there is a path containing at least $d - 1 - (l - 1) = d - l = \lceil \log(n+1) \rceil - \lceil \log(m+1) \rceil$

variables that are not in the first m memory locations. At least one I/O operation is needed per variable to operate on them. It thus follows that

$$\begin{aligned}\mathcal{K}_\nu(f_{\rm BS}^{(n)}) &= \sum_{d=0}^{\log n} \Omega(\log n - \log(2^d)) \\ &= \sum_{d=0}^{\log n} \Omega(\log n - d) \\ &= \Omega(\log^2 n)\end{aligned}$$

The last inequality is a consequence of the fact that $\log n - d$ is greater than $(\log n)/2$ for $d \leq (\log n)/2$. ■

Lower bounds on the I/O complexity for these problems can be derived for a large variety of cost functions. The reader is asked in Problem 11.20 to derive such bounds for the cost function $\nu(a) = a^\alpha$.

11.9.2 Upper Bounds for the HMM

A natural question in this context is whether these lower bounds can be achieved. We already know from Theorems 11.5.3 and 11.5.5 that for each allocation of memory to each memory-hierarchy level, it is possible to match upper and lower bounds on the number of I/O operations and computation time. As a consequence, for each of these problems near-optimal solutions exist for any cost function on memory accesses for these problems.

11.10 Competitive Memory Management

The results stated above for the hierarchical memory model assume that the user has explicit control over the location of data, an assumption that does not apply if storage is allocated by an operating system. In this section we examine **memory management** by an operating system for the HMM model, that is, algorithms that respond to memory requests from programs to move stored items (instructions and data) up and down the memory hierarchy. We examine offline and online memory management algorithms. An **offline algorithm** is one that has complete knowledge of the future. **Online algorithms** cannot predict the future and must act only on the data received up to the present time.

We use **competitive analysis**, a type of analysis not appearing elsewhere in this book, to show that the two widely used online page-replacement algorithms, least recently used (LRU) and first-in, first-out (FIFO), use about twice as many I/O operations as does MIN, the optimal offline page-replacement algorithm, when these two algorithms are allowed to use about twice as much memory as MIN. Competitive analysis bounds the performance of an online algorithm in terms of that of the optimum offline algorithm for the problem without knowing the performance of the optimum algorithm.

Virtual memory-management systems allow the programmer to program for one large virtual random-access memory, such as that assumed by the HMM, although in reality the memory contains multiple physical memory units one of which is a fast random-access unit accessed by the CPU. In such systems the hardware and operating system cooperate to move data from secondary storage units to the primary storage unit in **pages** (a collection of items). Each reference to a virtual memory location is checked to determine whether or not the referenced item is in primary memory. If so, the virtual address is converted to a physical one and the item fetched by the CPU. If not (if a **page fault** occurs), the page containing the virtual address is moved into primary memory and the tables used to translate virtual addresses are updated. The item at the virtual address is then fetched. To make room for the newly fetched page, one page in the fast memory is moved up the memory hierarchy.

A **page-replacement algorithm** is an algorithm that decides which page to remove from a full primary memory to make space for a new page. We describe and analyze page-replacement algorithms for two-level memory hierarchies both because they are important in their own right and because they are used as building blocks for multi-level page-replacement algorithms. A two-level hierarchy has primary and secondary memories. Let the primary memory contain n pages and let the secondary memory be of unlimited size.

The **FIFO** (first-in, first-out) page-replacement algorithm is widely used because it is simple to implement. Under this replacement policy, the page replaced is the first page to have arrived in primary memory. The **LRU** (least recently used) replacement algorithm requires keeping for each page the time it was last accessed and then choosing for replacement the page with the earliest time, an operation that is more expensive to implement than the FIFO shift register.

Under the **optimal two-level page-replacement algorithm**, called **MIN**, primary memory is initialized with the first n pages to be accessed. **MIN** replaces the page p_i in primary memory whose time t_i of next access is largest. If some other page, p_j, were replaced instead of p_i, p_j would have to return to the primary memory before p_i is next accessed, and one more page replacement would occur than is required by MIN.

Implementing MIN requires knowledge of the future, a completely unreasonable assumption on the part of the operating system designer. Nonetheless, MIN is very useful as a standard against which to compare the performance of other page-replacement algorithms such as FIFO and LRU.

11.10.1 Two-Level Memory-Management Algorithms

To compare the performance of FIFO, LRU, and MIN, we characterize memory use by a **memory-address sequence** $\boldsymbol{s} = \{s_1, s_2, \dots\}$ of HMM addresses accessed by a computation. We assume that no memory entries are created or destroyed. We let $F_{\text{FIFO}}(n, \boldsymbol{s})$, $F_{\text{LRU}}(n, \boldsymbol{s})$, and $F_{\text{MIN}}(n, \boldsymbol{s})$ be the number of page faults with each page-replacement algorithm on the memory address sequence $\boldsymbol{s}$ when the primary memory holds n pages.

We now bound the performance of the FIFO and LRU page-replacement algorithms in terms of that of MIN. We show that if the number of pages available to FIFO and LRU is double the number available to MIN, the number of page faults with FIFO and LRU is at most about double the number with MIN. It follows that FIFO and LRU are very good page-replacement algorithms, a result seen in practice.

THEOREM 11.10.1 *Let n_{FIFO}, n_{LRU}, and n_{MIN} be the number of primary memory pages used by the FIFO, LRU, and MIN algorithms. Let $n_{\text{FIFO}} \ge n_{\text{MIN}}$ and $n_{\text{LRU}} \ge n_{\text{MIN}}$. Then, for any memory-address sequence $\boldsymbol{s}$ the following inequalities hold:*

$$F_{\text{FIFO}}(n_{\text{FIFO}}, \boldsymbol{s}) \le \frac{n_{\text{FIFO}}}{n_{\text{FIFO}} - n_{\text{MIN}} + 1} F_{\text{MIN}}(n_{\text{MIN}}, \boldsymbol{s}) + n_{\text{MIN}}$$

$$F_{\text{LRU}}(n_{\text{LRU}}, \boldsymbol{s}) \leq \frac{n_{\text{LRU}}}{n_{\text{LRU}} - n_{\text{MIN}} + 1} F_{\text{MIN}}(n_{\text{MIN}}, \boldsymbol{s}) + n_{\text{MIN}}$$

Proof We establish the result for FIFO, leaving it to the reader to show it for LRU. (See Problem 11.23.) Consider a contiguous subsequence $\boldsymbol{t}$ of $\boldsymbol{s}$ that immediately follows a page fault under FIFO and during which FIFO makes $\phi^{\text{FIFO}} = f \leq n_{\text{FIFO}}$ page faults. In the next paragraph we show that at least f different pages are accessed by FIFO during $\boldsymbol{t}$. Let MIN make ϕ^{MIN} faults during $\boldsymbol{t}$. Because MIN has n_{MIN} pages, $\phi^{\text{MIN}} \geq f - n_{\text{MIN}} + 1 \geq 0$. Thus, the ratio of page faults by FIFO and MIN is $f/\phi^{\text{MIN}} \leq f/(f - n_{\text{MIN}} + 1)$.

Let p_i be the page on which the fault occurs just before the start of $\boldsymbol{t}$. To show that at least f different pages are accessed by FIFO during $\boldsymbol{t}$, consider the following cases: a) FIFO faults on p_i in $\boldsymbol{t}$; b) FIFO faults on some other page at least twice in $\boldsymbol{t}$; and c) neither case applies. In the first case, FIFO accesses at least n_{FIFO} different pages because if it accessed fewer, then p_i would still be in its primary memory the second time it is accessed. In the second case, the same statement applies to the page accessed multiple times. In the third case, FIFO can have only f faults if it accesses at least f different pages during $\boldsymbol{t}$.

Now subdivide the memory access sequence $\boldsymbol{s}$ into subsequences $\boldsymbol{t}_0, \boldsymbol{t}_1, \dots, \boldsymbol{t}_k$ such that $\boldsymbol{t}_i$, $i \geq 1$, starts immediately after a page fault under FIFO and contains n_{FIFO} faults and $\boldsymbol{t}_0$ contains at most n_{FIFO} page faults. This set of subsequences can be found by scanning $\boldsymbol{s}$ backwards. Since MIN makes $\phi_j^{\text{MIN}} \geq n_{\text{FIFO}} - n_{\text{MIN}} + 1$ faults on the jth interval, $j \geq 1$, and $\phi_0^{\text{MIN}} \geq \phi_0^{\text{FIFO}} - n_{\text{MIN}}$ faults on the zeroth interval (that is, $\phi_0^{\text{FIFO}} \leq \phi_0^{\text{MIN}} + n_{\text{MIN}}$), the number of faults by FIFO, $F_{\text{FIFO}}(n_{\text{FIFO}}, \boldsymbol{s}) = \phi_0^{\text{FIFO}} + \phi_1^{\text{FIFO}} + \cdots + \phi_k^{\text{FIFO}}$ satisfies the condition of the theorem because $\phi_j^{\text{FIFO}} \leq n_{\text{FIFO}}\phi_j^{\text{MIN}}/(n_{\text{FIFO}} - n_{\text{MIN}} + 1)$ for $j \geq 1$. ■

The upper bounds are almost best possible because, as stated in Problem 11.24, for any online algorithm A there is a memory-access sequence such that the number of page faults $F_A(\boldsymbol{s})$ satisfies the following lower bound:

$$F_A(n_A, \boldsymbol{s}) \geq \frac{n_A}{n_A - n_{\text{MIN}} + 1} F_{\text{MIN}}(n_{\text{MIN}}, \boldsymbol{s})$$

The difference between this lower bound and the upper bounds given for FIFO and LRU is n_{MIN}, which takes into account for the possibility that the initial entries in the primary memory of MIN and FIFO can be completely different.

It follows that the FIFO and LRU page-replacement strategies are very effective strategies for two-level memory hierarchies.

Problems

MATHEMATICAL PRELIMINARIES

11.1 Let a and b be integers satisfying $1 \leq a \leq b$. Show that $b/2 \leq a\lfloor b/a \rfloor \leq b$.

Hint: Consider values of b in the range $ka \leq b \leq (k+1)a$ for k an integer.

11.2 Derive a good lower bound on $\sum_{k=1}^{m}(1/k)$ of the form $\Omega(\log m)$ using an approach similar to that of Problem 2.2.

PEBBLING MODELS

11.3 Show that the graph of Fig. 11.2 can be completely pebbled in the three-level MHG with resource vector $\boldsymbol{p} = (2, 4)$ using only four third-level pebbles.

11.4 Consider pebbling a graph with the red-blue game. Suppose that each I/O operation uses twice as much time as a computation step. Show by example that a red-blue pebbling minimizing the total time to pebble a graph does not always minimize the number of I/O operations.

I/O TIME RELATIONSHIPS

11.5 Let S_{min} be the minimum number of pebbles needed to pebble the graph $G = (V, E)$ in the red pebble game. Show that if in the MHG a pebbling strategy $\mathcal{P}$ uses s_k pebbles at level k or less and $s_k \geq S_{\text{min}} + k - 1$, then no I/O operations at level $k + 1$ or higher are necessary except on input and output vertices of G.

11.6 The rules of the red-blue pebble game suggest that inputs should be prefetched from high-level memory units early enough that they arrive when needed. Devise a schedule for delivering inputs so that the number of I/O operations for matrix multiplication is minimized in the red-blue pebble game.

THE HONG-KUNG LOWER-BOUND METHOD

11.7 Derive an expression for the S-span $\rho(S, G)$ of the binary tree G shown in Fig. 11.4.

11.8 Consider the pyramid graph G on n inputs shown in Fig. 11.18. Determine its S-span $\rho(S, G)$ as a function of S.

11.9 In Problem 2.3 it is shown that every binary tree with k leaves has $k-1$ internal vertices. Show that if t binary trees have a total of p pebbles, at most $p - 1$ pebbling steps are possible on these trees from an arbitrary initial placement without re-pebbling inputs.

Hint: The vertices that can be pebbled from an initial placement of pebbles form a set of binary trees.

11.10 An I/O operation is **simple** if after a pebble is placed on a vertex the pebble currently residing on that vertex is removed. Show that at most twice as many I/O operations are used at each level by the MHG when every I/O operation is simple.

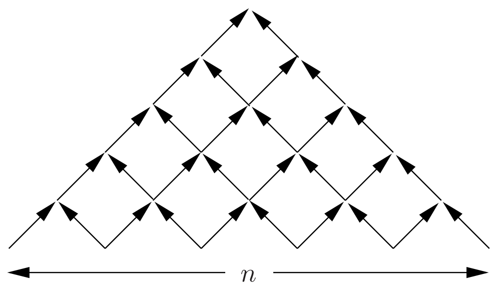


Figure 11.18 The pyramid graph.

Hint: Compare pebble placement with and without the requirement that placements be simple, arguing that if a pebble removed by a simple I/O operation is needed later it can be obtained by one simple I/O operation for each of the original I/O operations.

TRADEOFFS IN THE MEMORY HIERARCHIES

11.11 Using the results of Problem 11.8, derive good upper and lower bounds on the I/O time to pebble the pyramid graph of Fig. 11.18 in terms of n.

11.12 Under the conditions of Problem 11.4, show that any pebbling of a DAG for convolution of n-sequences with the minimal pebbling strategy when $S \geq S_{\min}$ and n is large has much larger total cost than a strategy that treats blue pebbles as red pebbles.

BLOCK I/O IN THE MHG

11.13 Determine how efficiently matrix-vector multiplication can be done in the block-I/O model described in Section 11.6.

11.14 Show that matrix-matrix multiplication can be done efficiently in the block-I/O model described in Section 11.6.

SIMULATING FAST MEMORIES

11.15 Determine conditions on a memory hierarchy under which the FFT can be executed efficiently in the standard MHG. Discuss the extent to which these conditions are likely to be met in practice.

11.16 Repeat the previous problem for convolution realized by the algorithm stated in the convolution theorem.

11.17 The definition of a minimal pebbling stated in Section 11.2 assumes that it is much more expensive to perform a high-level I/O operation than a low-level one. Determine the extent to which the lower bound of Theorem 11.4.1 depends on this assumption. Apply your insight to the problem of matrix multiplication of $n \times n$ matrices in the three-level MHG in which $s_1 < 3n^2$ and $s_2 \geq 3n^2$. (See Theorem 11.5.3.) Determine whether increasing the number of level-3 I/O operations affects the number of level-2 I/O operations.

THE BLOCK-TRANSFER MODEL

11.18 Derive a lower bound on the time to realize a permutation network on n inputs in the block-transfer model.

Hint: Count the number of orderings possible between the n inputs. Base your argument on the number of orderings within blocks and between elements in the primary memory, and the number of ways of choosing which block from the secondary memory to move into the primary memory.

11.19 Derive a lower bound on the time to realize the FFT graph on n inputs in the block-transfer model.

Hint: Use the result of Section 7.8.2 to argue that an n-point FFT graph cannot have many fewer vertices than there are switches in a permutation network.

THE HIERARCHICAL MEMORY MODEL

11.20 Derive the following lower bounds on the cost of computing the following functions when the cost function is $\nu(a) = a^\alpha$:

$$\textbf{Matrix multiplication:} \quad \mathcal{K}_\nu(f^{(n)}_{A\times B}) = \begin{cases} \Omega(n^{2\alpha+2}) & \text{if } \alpha > 1/2 \\ \Omega(n^3 \log n) & \text{if } \alpha = 1/2 \\ \Omega(n^3) & \text{if } \alpha < 1/2 \end{cases}$$

$$\textbf{Fast Fourier transform:} \quad \mathcal{K}^{(n)}_c(F^{(d)}) = \Omega(n^{\alpha+1})$$

$$\textbf{Binary search:} \quad \mathcal{K}_\nu(f^{(n)}_{\text{BS}}) = \Omega(n^\alpha)$$

Hint: Use the following identity to recast expressions for the computation time:

$$\sum_{k=1}^{n} \Delta g(k)h(k) = -\sum_{k=1}^{n-1} \Delta h(k)g(k+1) + g(n+1)h(n) - g(1)h(1)$$

11.21 A cost function $\nu(a)$ is **polynomially bounded** if for some $K > 1$ and all $a \geq 1$. $\nu(2a) \leq K\nu(a)$. Let the cost function $\nu(a)$ be polynomially bounded. Show that there are positive constants c and d such that $\nu(a) \leq ca^d$.

11.22 Derive a good upper bound on the cost to sort in the HMM with the logarithmic cost function $\lceil \log a \rceil$.

COMPETITIVE MEMORY MANAGEMENT

11.23 By analogy with the proof for FIFO in the proof of Theorem 11.10.1, consider any memory-address sequence $\boldsymbol{s}$ and a contiguous subsequence $\boldsymbol{t}$ of $\boldsymbol{s}$ that immediately follows a page fault under LRU and during which LRU makes $\phi^{\text{LRU}} = f \leq n_{\text{LRU}}$ page faults. Show that at least f different pages are accessed by LRU during $\boldsymbol{t}$.

11.24 Let A be any online page-replacement algorithm that uses n_A pages of primary memory. Show that there are arbitrarily long memory-address sequences $\boldsymbol{s}$ such that the number of page faults with A, $F_A(\boldsymbol{s})$, satisfies the following lower bound, where n_{MIN} is the number of pages used by the optimal algorithm MIN:

$$F_A(\boldsymbol{s}) \geq \frac{n_A}{n_A - n_{\text{MIN}} + 1} F_{\text{MIN}}(\boldsymbol{s})$$

Hint: Design a memory-address sequence $\boldsymbol{s}$ of length n_A with the property that the first $n_A - n_{\text{MIN}} + 1$ accesses by A are to pages that are neither in A's or MIN's primary memory. Let $\mathcal{S}$ be the $n_A + 1$ pages that are either in MIN's primary memory initially or those accessed by A during the first $n_A - n_{\text{MIN}} + 1$ accesses. Let the next $n_{\text{MIN}} - 1$ page accesses by A be to pages not in $\mathcal{S}$.

Chapter Notes

Hong and Kung [136] introduced the first formal model for the I/O complexity of problems, the red-blue pebble game, an extension of the pebble game introduced by Paterson and Hewitt [238]. The analysis of Section 11.1.2 is due to Kung [177]. Hong and Kung derived lower bounds on the number of I/O operations needed for specific graphs for matrix multiplication (Theorem 11.5.2), the FFT (Theorem 11.5.4), odd-even transposition sort and a number of other problems. Savage [294] generalized the red-blue pebble game to the memory-hierarchy game, simplified the proof of Theorem 11.4.1, and obtained Theorems 11.5.3 and 11.5.5 and the results of Section 11.3. Lemma 11.5.2 is implicit in the work of Hong and Kung [136]; the simplified proof given here is due to Agrawal and Vitter [9]. The results of Section 11.5.4 are due to Savage [294].

The two-level contiguous block-transfer model of Section 11.8.1 was introduced by Savage and Vitter [295] in the context of parallel space–time tradeoffs. The analysis of sorting of Section 11.8.1 is due to Agrawal and Vitter [9]. In this paper they also derive similar bounds on the I/O time to realize the FFT, permutation networks and matrix transposition.

The hierarchical memory model of Section 11.9 was introduced by Aggarwal, Alpern, Chandra, and Snir [7]. They studied a number of problems including matrix multiplication, the FFT, sorting and circuit simulation, and examined logarithmic, linear, and polynomial cost functions. The two-level bounds of Section 11.10 are due to Sleator and Tarjan [310]. Aggarwal, Alpern, Chandra, and Snir [7] extended this model to multiple levels. The MIN page-replacement algorithm described in Section 11.10 is due to Belady [35].

Two other I/O models of interest are the BT model and the uniform memory hierarchy. Aggarwal, Chandra, and Snir [8] introduced the **BT model**, an extension of the HMM model supporting block transfers in which a block of size b ending at location x is allowed to move in time $f(x) + b$. They establish tight bounds on computation time for problems including matrix transpose, FFT, and sorting using the cost functions $\lceil \log x \rceil$, x, and x^α for $1 \leq \alpha \leq 1$.

Alpern, Carter, and Feig [18] introduced the **uniform memory hierarchy** in which the uth memory has capacity $\alpha\rho^{2u}$, block size ρ^u, and time $\rho^u/\beta(u)$ to move a block between levels; $\beta(u)$ is a bandwidth function. They allow I/O overlap between levels and determine conditions under which matrix transposition, matrix multiplication, and Fourier transforms can and cannot be done efficiently.

Vitter and Shriver [353] have examined three parallel memory systems in which the memories are disks with block transfer, of the HMM type, or of the BT type. They present a randomized version of distribution sort that meets the lower bounds for these models of computation. Nodine and Vitter [231] give an optimal deterministic sorting algorithm for these memory models.