Job Scheduling in a Partitionable Mesh Using a Two-Dimensional Buddy System Partitioning Scheme

Keqin Li, *Member, IEEE,* and Kam-Hoi Cheng

***Abstract*—In this paper, we address the job scheduling problem in a partitionable mesh-connected system when jobs require square meshes and the system is a square mesh of size a power of two. We present a heuristic algorithm of time complexity $O(n(\log n + \log p))$where n is the number of jobs to be scheduled and p is the size of the system. The algorithm adopts the *largest job first* scheduling policy and uses a *two-dimensional buddy system* as the system partitioning scheme. It is shown that in the worst case, the algorithm produces a schedule four times longer than an optimal schedule. On the average, schedules generated by the algorithm are about twice longer than optimal schedules.**

***Index Terms*— Approximation algorithm, asymptotic performance bound, job scheduling, NP-hard, partitionable mesh-connected system, system partitioning scheme, two-dimensional buddy system.**

I. Introduction

THE job scheduling problem in a *partitionable mesh-connected system* (PMCS) is first introduced in [7]. An interesting feature of a PMCS is that it can be partitioned into many rectangular submeshes and each submesh can be assigned to a job which may execute any parallel algorithm either in SIMD/MIMD manner or in systolic fashion. A job J is specified as $J = (a, b, t)$, which means that job J requires a rectangular mesh of size $a \times b$ and its processing time is t. The *job scheduling* problem is to determine for a given job list $L = (J_1, J_2, \cdots, J_n)$ where $J_i = (a_i, b_i, t_i), 1 \leq i \leq n$, and a PMCS of size $p \times q$, in what order these jobs should be processed and how to arrange these jobs in the PMCS such that the finish time is minimized. Note that only nonpreemptive job scheduling is considered, i.e., whenever a submesh is allocated to a job, it remains occupied until the job finishes. For each job J, a schedule specifies the location and the size of the submesh assigned to J as well as the time interval during which job J is to be executed. Readers are referred to [6]–[8] for more details about the concept of a PMCS and definitions for the job scheduling and other related problems.

In this paper, we are only concerned with the case when each job J_i requires a square submesh (i.e., $a_i = b_i$) and the PMCS is also a square mesh with size a power of 2 (i.e., $p = q = 2^R$). The specification of a job is simplified to $J = (s, t)$ where $s \times s$ denotes the size of the submesh required by job J. An instance of the job scheduling problem is denoted as $I = \langle L, p \rangle$. The fundamental difficulties of the job scheduling problem in PMCS are twofold [8]. The first difficulty is the inherent NP-hardness of job scheduling even in very simple cases. For example, when $p^2 = m$, and $a_i = b_i = 1, 1 \leq i \leq n$, our special case job scheduling problem in PMCS reduces to the multiprocessor scheduling problem which is NP-hard even when $m = 2$ [4]. The second difficulty is that any job scheduling algorithm in PMCS needs to use a *system partitioning scheme,* i.e., to schedule a set of jobs to run simultaneously, we also need to know how to arrange them in the PMCS. Thus, in addition to the scheduling policy, the system partitioning scheme also plays an important role in any job scheduling algorithm in PMCS [8].

One feasible approach to solve NP-hard optimization problems is to design *approximation algorithms* which produce near-optimal solutions [3]. Let $\mathrm{OPT}(I)$ and $A(I)$ be the finish time of an optimal schedule of I and the schedule generated by an approximation algorithm A, respectively. If we can show that there exist constants α and β such that for any instance I,

$$A(I) \leq \alpha \cdot \mathrm{OPT}(I) + \beta \cdot \max_{1 \leq i \leq n} (t_i),$$

then α is called an *asymptotic performance bound* of algorithm A. If one can further demonstrate that for any small $\varepsilon > 0$ and any large $N > 0$, there exists an instance I such that $\mathrm{OPT}(I) > N$ and $A(I) \geq (\alpha - \varepsilon) \cdot \mathrm{OPT}(I)$, then the bound α is tight. An asymptotic performance bound characterizes the behavior of an algorithm as the ratio $\mathrm{OPT}(I)$ to the longest processing time goes to infinity [3].

We propose a polynomial time heuristic algorithm, LJF, which uses the *largest job first* scheduling policy and a *two-dimensional buddy system* (2DBS) partitioning scheme. The dynamic system partitioning scheme 2DBS is developed in [9] for resource allocation and reclamation in a PMCS. It only needs to know job sizes and does not require any knowledge about the processing time of any job. The main advantage of algorithm LJF, over other algorithms such as LL in [8], is the separation of the system partitioning strategy from the job scheduling policy. This nice feature makes it more tractable to analyze the algorithm from an average point of view, which is usually very difficult. Furthermore, a separate

Manuscript received April 28, 1989; revised March 26, 1990. This work was supported in part by the Texas Advanced Research Program under Grant 1028-ARP.

K. Li is with the Department of Mathematics and Computer Science, State University of New York—the College at New Paltz, New Paltz, NY 12561.

K.-H. Cheng is with the Department of Computer Science, University of Houston, Houston, TX 77204.

IEEE Log Number 9102439.

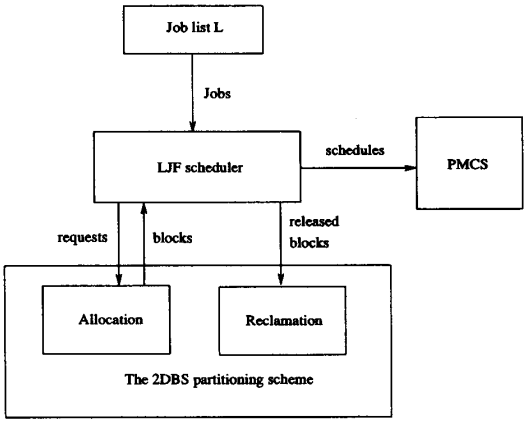


Fig. 1. The solution scheme.

system partitioning strategy makes it easy to schedule jobs on-line or dynamically. Fig. 1 shows the configuration of our solution scheme.

The rest of the paper is organized as follows. A brief review of 2DBS is given in Section II. Our proposed algorithm is then presented in Section III. The algorithm is analyzed in Section IV for worst case performance and in Section V for average case performance. Adaptation of the algorithm for on-line and dynamic job scheduling is discussed in Section VI. Finally the conclusion is given in Section VII.

II. 2DBS: A Dynamic System Partitioning Scheme

The two-dimensional buddy system proposed in [9] is a generalization of the traditional one-dimensional binary buddy system in memory management [11]. This system partitioning scheme is applied to the situation when all jobs need square submeshes and the PMCS itself is a square mesh of size $p = q = 2^R$. In this scheme, the size of a request is rounded up to the nearest power of 2, i.e., if a job requires a submesh of size s, we always assign a submesh of size $\omega(s) = 2^{\lceil \log s \rceil}$. If the required block size is not available, the algorithm searches for a larger block and splits it into smaller blocks. A submesh of the required size is then assigned to the request. When no sufficiently large block can be found, *resource overflow* occurs and the job is delayed (i.e., put into a waiting queue). When a job releases its occupied block, the block is reclaimed and may be combined with other free blocks to form a larger block.

Since the system size is $p = 2^R$ and sizes of all blocks are powers of 2, the block sizes provided by the 2DBS are $1, 2, 4, \cdots, 2^i, \cdots, 2^R$. If we use a two-dimensional coordinate system for a PMCS, then each processor will have a unique location (x, y), where $0 \le x, y \le 2^R - 1$. A block (i.e., a square submesh) is specified as $B(x, y, k)$, where (x, y) is the location of the processor at the lowest left-hand corner of the block and 2^k is its size. We are only interested in those blocks generated in the splitting and the combining processes. As a result, not all $B(x, y, k), 0 \le x, y \le 2^R - 1, 0 \le k \le R$, are meaningful blocks in 2DBS. For example, $B\left(2^{R-2}, 2^{R-2}, R-2\right)$ is valid but $B\left(2^{R-1}+1, 2^{R-1}+1, R-1\right)$ is invalid. Formally a *block* is defined as follows:

1) $B(0, 0, R)$ is a block.
2) If $B(x, y, k)$ is a block and $k > 0$, then $B(x, y, k-1)$, $B\left(x + 2^{k-1}, y, k-1\right), B\left(x, y + 2^{k-1}, k-1\right)$ and $B\left(x + 2^{k-1}, y + 2^{k-1}, k-1\right)$ are also blocks.

Blocks $B(x, y, k-1), B(x + 2^{k-1}, y, k-1), B(x, y + 2^{k-1}, k-1)$, and $B\left(x + 2^{k-1}, y + 2^{k-1}, k-1\right)$ are *buddies* of one another, i.e., except the block $B(0, 0, R)$ which represents the original whole system, all blocks in a 2DBS have three buddies. We call them lower-left (LL), lower-right (LR), upper-left (UL), and upper-right (UR) buddies, respectively.

To allocate and reclaim blocks dynamically, the determination of the existence of buddies of a block is very important. First, we give two lemmas which are not difficult to prove by using the definition of a block.

Lemma 1: For all $0 \le k \le R$, there are 4^{R-k} blocks of size 2^k. ■

Lemma 2: $B(x, y, k)$ is a block if and only if $x \bmod 2^k = 0$ and $y \bmod 2^k = 0$, i.e., if binary representations of coordinates x and y are $x = (\alpha_{R-1}\alpha_{R-2}\cdots\alpha_0)_2$ and $y = (\beta_{R-1}\beta_{R-2}\cdots B_0)_2$, respectively, we have $\alpha_i = 0, \beta_i = 0$ for $k > i \ge 0$. ■

Based on these observations, we define an 1-1 function called ORDER which maps all blocks of size 2^k onto $\{0, 1, 2, \cdots, 4^{R-k} - 1\}$, i.e., we have an order among all blocks of the same size. The ORDER value of a block $B(x, y, k)$ is recursively defined as follows:

1) $\text{ORDER}(B(0, 0, R)) = 0$.
2) If $\text{ORDER}(B(x, y, k)) = N$ and $k > 0$, then
   $\text{ORDER}(B(x, y, k-1)) = 4N$,
   $\text{ORDER}(B(x + 2^{k-1}, y, k-1)) = 4N + 1$,
   $\text{ORDER}(B(x, y + 2^{k-1}, k-1)) = 4N + 2$,
   $\text{ORDER}(B(x + 2^{k-1}, y + 2^{k-1}, k-1)) = 4N + 3$.

By the definition of the ORDER function, buddies have consecutive ORDER values. For the block $B(0, 0, R)$, we simply define its ORDER value to be 0. For the block $B(x, y, k)$ where $k < R$, Lemma 3 [9] gives an algorithm to compute its ORDER value.

Lemma 3: If $x = (\alpha_{R-1}\alpha_{R-2}\cdots\alpha_0)_2$, $y = (\beta_{R-1}\beta_{R-2}\cdots\beta_0)_2$ and $0 \le k < R$, then $\text{ORDER}(B(x, y, k)) = (\beta_{R-1}\alpha_{R-1}\beta_{R-2}\alpha_{R-2}\cdots\beta_k\alpha_k)_2$. ■

The main data structure for implementing the 2DBS consists of $R + 1$ *free block lists* (FBL's). Let FBL_i be a doubly linked list of all available blocks of size 2^i, $0 \le i \le R$. Blocks in FBL_i are arranged in increasing order of their ORDER values. A block B_1 is on the left of a block B_2 if and only if $\text{ORDER}(B_1) < \text{ORDER}(B_2)$. This ordered list makes locating buddies quite simple. To determine the existence of all three buddies of the block $B(x, y, k)$, we first find the place in FBL_k where the block $B(x, y, k)$ is supposed to be according to its ORDER value. The decision procedure is then based on $\text{ORDER}(B(x, y, k)) \bmod 4$. For example, let the block $B(x, y, k)$ be an LR buddy. If the LL buddy is in FBL_k, then it must be on the left of the block $B(x, y, k)$; and if the UL buddy is also in FBL_k, it must be on the right of the block $B(x, y, k)$, and so on. Since the list FBL_k is implemented as a doubly linked list, we have *left* and *right*

pointers in each node. To determine the presence of all its three buddies, we check the following condition:

$$ptr.left \neq nil \ \&\ \text{ORDER}(ptr.left) = ord - 1 \ \&$$
$$ptr.right \neq nil \ \&\ \text{ORDER}(ptr.right) = ord + 1 \ \&$$
$$ptr.right.right \neq nil \ \&\ \text{ORDER}(ptr.right.right) = ord + 2$$

where ptr points to the block $B(x, y, k)$ in FBL_k and $ord = \text{ORDER}(B(x, y, k))$. It is not too difficult for the reader to come up with conditions for other cases.

The allocation procedure for a request of size s and the reclamation procedure for releasing a block $B(x, y, i)$ are formally described in Algorithms 1 and 2, respectively.

The allocation procedure

1) Round the size s to the smallest power of 2, i.e., let $i \leftarrow \lceil \log s \rceil$.
2) Search free block lists starting from FBL_i for the smallest j such that $j \geq i$ and FBL_j is not empty. If for all $j, i \leq j \leq R$, FBL_j is empty, then return "not found"; otherwise *remove* $B_0(x, y, j)$, the first block in the list FBL_j, from FBL_j.
3) While $j > i$, split the block B_0 into four buddies: $B_0(x, y, j-1), B_1(x + 2^{j-1}, y, j-1), B_2(x, y + 2^{j-1}, j-1)$, and $B_3(x + 2^{j-1}, y + 2^{j-1}, j-1)$. *Append* B_1, B_2, and B_3 into FBL_{j-1}. Set $j \leftarrow j - 1$.
4) Return the block $B_0(x, y, j)$.

Algorithm 1

The reclamation procedure

1) *Insert* $B(x, y, i)$ into FBL_i.
2) If at least one buddy of $B(x, y, i)$: $B(x_1, y_1, i)$,$B(x_2, y_2, i)$, and $B(x_3, y_3, i)$ is not in the list FBL_i, then return; otherwise,
   2.1) *Remove* all four buddies from FBL_i.
   2.2) Combine them into a new block $B(x, y, i+1)$ where $x \leftarrow \min(x, x_1, x_2, x_3)$, $y \leftarrow \min(y, y_1, y_2, y_3)$.
   2.3) Go back to step 1.

Algorithm 2

Let us examine the time complexity of the allocation procedure. Steps 1 and 4 each take $O(1)$ time. Step 2 requires at most $O(R)$ time because it is repeated at most $R + 1$ times (note that there are only $R + 1$ lists) and *remove* is an $O(1)$ operation. Step 3 takes the same amount of time as Step 2 because it goes back through all lists checked in Step 2, and *append* is an $O(1)$ operation. Thus, the overall time complexity of the allocation procedure is $O(R)$.

In the reclamation procedure, Step 1 takes $O(p^2)$ time in the worst case. For example, consider the case when FBL_i are empty for $1 \leq i \leq R$ and FBL_0 contains blocks $B(x, y, 0)$ such that $\text{ORDER}(B(x, y, 0)) \bmod 4 \neq 3$. Clearly the length of FBL_0 is $\frac{3}{4}p^2$. Now, when the block $B(p-1, p-1, 0)$ is inserted into FBL_0, Step 1 takes $\frac{3}{4}p^2$ units of time because $B(p-1, p-1, 0)$ should be put at the end of the list. Since the overall time complexity of the reclamation procedure is bounded by $O(p^2)$, its time complexity is $O(p^2)$.

III. Algorithm LJF and its Time Complexity

Our proposed job scheduling algorithm, denoted as LJF, uses the Largest Job First scheduling policy. Given a job list $L = (J_1, J_2, \cdots, J_n)$, where $J_i = (s_i, t_i)$, $1 \leq i \leq n$, algorithm LJF first sorts the list L in nonincreasing order of job size. The ordering for jobs with the same size but different processing times is arbitrary. Then jobs are considered one by one in this order. For each job J_i, algorithm LJF produces a schedule $(B(x_i, y_i, k_i), S_i)$ which gives the starting time S_i of J_i as well as the location (x_i, y_i) and size 2^{k_i} of the submesh assigned to J_i. The block $B(x_i, y_i, k_i)$ is found by using the allocation procedure of 2DBS. A variable T is used to record the current time and the list L_a is used to keep *active* jobs. For each job J_i which has been scheduled, a record $r_i = (B(x_i, y_i, k_i), f_i)$ is inserted into the list L_a where f_i is the time at which J_i will finish. The list L_a is maintained in nondecreasing order of f. Thus, the scheduler knows that blocks in L_a are released in this order. Whenever the allocation procedure reports that no block is large enough for the current job, the scheduler calls the reclamation procedure of 2DBS to release blocks at the beginning of L_a until a large enough block is available. The variable T is then updated accordingly. A formal description of algorithm LJF is given in Algorithm 3.

Algorithm LJF

1) Initialize all data structures.
2) Sort list L in nonincreasing order of job size.
3) While L is not empty,
   3.1) Remove the first job $J = (s, t)$ from L.
   3.2) Call the *allocation procedure* of 2DBS to find a block of size $2^{\lceil \log s \rceil}$.
   3.3) If such a block $B(x, y, k)$ is not found, call the *reclamation procedure* of 2DBS to release all nodes at the beginning of L_a which have the same f, set $T \leftarrow f$, go back to Step 3.2.
   3.4) Generate the schedule of job J which is $(B(x, y, k), T)$ and insert the record $(B(x, y, k), T + t)$ into L_a.

Algorithm 3

Let us discuss the time complexity of algorithm LJF. It is possible that the situation shown in the analysis of the reclamation procedure may occur in Step 3.3 when the reclamation procedure is called. Thus, the time complexity of LJF is at least $O(p^2)$. Note that p is not a measurement of problem size, but the magnitude of an element in the problem instance. Such an algorithm, whose time complexity is a polynomial function of the magnitude of an input, is called a *pseudo polynomial time* algorithm [4]. This means that the time complexity of LJF is still an exponential function of the length of the input.

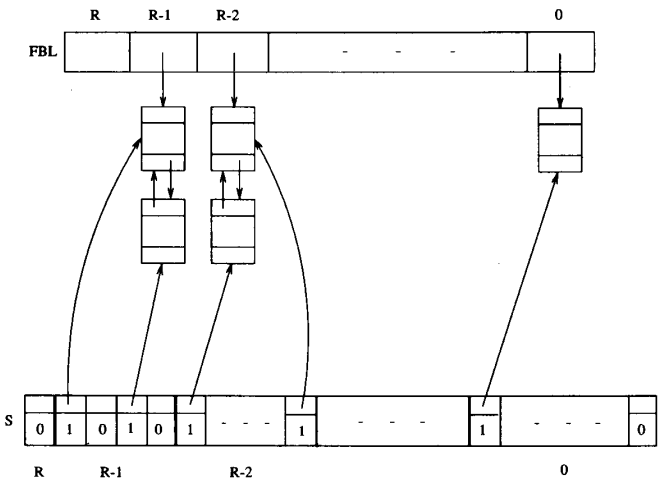


Fig. 2. Data structures of the 2DBS.

To solve this problem, we require that both allocation and reclamation procedures take at most $O(R) = O(\log p)$ time. For this purpose, we propose an alternative data structure for implementing the 2DBS (Fig. 2). The new structure also has $R + 1$ doubly linked FBL's; however, each list is not ordered by ORDER values. In addition, a *status* array S of length

$$l = \sum_{i=0}^{R} 4^i = \frac{1}{3}(4^{R+1} - 1) = \frac{1}{3}(4p^2 - 1)$$

is used to record the status of each possible block. The status of the block $B(x, y, k)$ is recorded in $S[index]$ where

$$index = \sum_{i=0}^{k-1} 4^i + \text{ORDER}(B(x, y, k)).$$

There are two fields in each element of S: a bit b and a pointer ptr. The bit b indicates whether the block $B(x, y, k)$ is available (1) or not (0). If the block is free, ptr is the pointer to the node representing $B(x, y, k)$ in FBL_k; otherwise, it is *nil* or meaningless.

With this new data structure, the only modification in the algorithms is to change the *insert* operation in Step 1 of the reclamation procedure to the *append* operation (cf. Algorithm 2). Note that all *remove* and *append* operations should also modify S accordingly. The determination for the existence of buddies of a released block can easily be accomplished by the ORDER values and the information kept in S. It is not hard to see that the determination for the existence of buddies, *remove* and *append* all take constant time, i.e., both allocation and reclamation take $O(R) = O(\log p)$ time.

The time complexity of algorithm LJF, based on this new scheme is analyzed as follows. The sorting process in Step 2 takes $O(n \log n)$ time. It is clear that to schedule n jobs, Step 3.3 is executed at most $n - 1$ times because the last job will not be removed from list L_a. Thus, Step 3.3 needs no more than $O(nR)$ time. For each job, Steps 3.1, 3.2, and 3.4 take constant, $O(R)$, and $O(n)$ time, respectively. So the total amount of time spent for each job (excluding Step 3.3) is bounded by $O(n + R)$. Since there are n jobs to be scheduled, the overall time complexity of algorithm LJF is bounded above by

$$\begin{aligned} O(n \log n + nR + n(R+n)) &= O(n(\log n \\ &\quad + 2\log p + n)) \\ &= O(n(n + \log p)) \end{aligned}$$

which is polynomial in the input length. Note that L_a may be maintained as a priority queue. If it is implemented as a heap [5], then the time complexity of Step 3.4 is reduced to $O(\log n)$; hence, the time complexity of algorithm LJF is $O(n(\log n + \log p))$.

IV. Worst Case Performance Analysis

Let us now discuss the performance of algorithm LJF. First we present a lemma which states an important property of the 2DBS.

Lemma 4: Let $s_1 \geq s_2 \geq \cdots \geq s_n$ be a sequence of requests where $s_i = 2^{e_i}, 1 \leq i \leq n$. Then upon resource overflow FBL_j is empty for all $0 \leq j \leq R$.

Proof: Consider a request of size $s = 2^e$. If it is satisfied by the allocation procedure, then either a block $B(x, y, e)$ is removed from FBL_e or a block $B(x, y, k)$ with $k > e$ is divided into several smaller blocks, one of them is allocated to the request and the rest are added to FBL's with the property that none of them have size less than s. Now suppose resource overflow occurs when the allocation procedure is called for the request with size $s_i = 2^{e_i}$. Obviously FBL_j is empty for $e_i \leq j \leq R$, otherwise the request will be satisfied. In addition, no block of size less than s_{i-1} is generated in satisfying requests $s_1, s_2, \cdots, s_{i-1}$, i.e., FBL_j is also empty for $0 \leq j \leq e_i - 1$. ■

We call a request sequence $s_1, s_2, \cdots, s_n$ a *power of 2 nonincreasing* (P2NI) sequence if $s_1 \geq s_2 \geq \cdots \geq s_n$ and

for all $1 \le i \le n, s_i = 2^{e_i}$. Lemma 4 shows that there is no external fragmentation for any P2NI-sequence of requests. Clearly there is no internal fragmentation as well.

Corollary: For any instance $I = \langle L, p\rangle$ where $L = (J_1, J_2, \cdots, J_n), J_i = (2^{e_i}, t_i), 1 \le i \le n$, and $\sum_{i=1}^{n} (2^{e_i})^2 \le p^2$, we have $\mathrm{LJF}(I) = \mathrm{OPT}(I)$.

Proof: After the sorting step, L is arranged as a P2NI-sequence according to job sizes. By Lemma 4, all jobs in L can fit in the system. Thus, algorithm LJF generates a schedule in which all jobs in L start at the same time. This is obviously an optimal schedule. ■

In [7], it is proved that under conditions of the above corollary, all jobs in L *can* be done simultaneously; but that result does not tell us *how* to arrange them in the PMCS. By using a 2DBS, such an optimal arrangement can be realized.

Let the *volume* of a job $J = (s, t)$ be $V(J) = s^2 \cdot t$, and the volume of a job list L be $V(L) = \sum_{J \in L} V(J)$. The volume $V(J)$ indicates the amount of *resource* × *time* needed by job J. Using $V(L)$, an obvious lower bound of $\mathrm{OPT}(I)$ is $\mathrm{OPT}(I) \ge V(L)/p^2$.

Theorem 1: For any instance $I = \langle L, p\rangle$ where $L = (J_1, J_2, \cdots, J_n), J_i = (2^{e_i}, t_i)$, $1 \le i \le n$, we have $\mathrm{LJF}(I) < \mathrm{OPT}(I) + \max_{1 \le i \le n}(t_i)$.

Proof: Suppose Step 3.3 is executed m times during the computation of algorithm LJF on input L. Since variable T is incremented each time Step 3.3 is done, let $T_1 < T_2 < \cdots < T_m$ be these values and $T_0 = 0$. Since the sequence of job sizes is sorted, so the next job can always fit into the system each time T is updated, i.e., T is never updated twice before the next job is allocated. At every $T_i, 0 \le i \le m-1$, jobs keep being allocated until the next resource overflow occurs. By Lemma 4, at each moment T_i $(0 \le i \le m-1)$, the PMCS is fully utilized. In addition, in the time interval $[T_i, T_{i+1})$, no job finishes and so this 100% utilization lasts until T_{i+1}. In other words, the system is fully utilized in $[0, T_m)$. At time T_m, there may not be enough jobs to fill the system and after time T_m, the utilization drops whenever a job finishes. Let $r = (B(x, y, k), f)$ be the last record in L_a at time T_m. Then the situation is illustrated in Fig. 3. Clearly $V(L) > T_m \cdot p^2$. Therefore,

$$\mathrm{OPT}(I) \ge \frac{V(L)}{p^2} > T_m.$$

Since the job with finish time f starts no later than T_m, i.e., $f - T_m \le \max_{1 \le i \le n}(t_i)$, we have

$$\mathrm{LJF}(I) = f < \mathrm{OPT}(I) + \max_{1 \le i \le n}(t_i).$$

The theorem is thus proven. ■

Theorem 1 states that algorithm LJF is asymptotically optimal ($\alpha = 1$) for jobs which require square submeshes with sizes being powers of 2. In general, job sizes are not powers of 2, which implies that there is internal fragmentation in most blocks. The performance of algorithm LJF in the general case is evaluated in the next theorem.

Theorem 2: For any instance $I = \langle L, p\rangle$ where $L = (J_1, J_2, \cdots, J_n), J_i = (s_i, t_i), 1 \le i \le n$, we have $\mathrm{LJF}(I) < 4 \cdot \mathrm{OPT}(I) + \max_{1 \le i \le n}(t_i)$.

Proof: Note that after sorting in LJF and rounding in the allocation procedure, the sequence of job sizes is transformed into a P2NI-sequence. The scenario is similar to that of Theorem 1 except that due to internal fragmentation, system utilization is not always 100% during the time interval $[0, T_m)$ (Fig. 4). In the worst case, internal fragmentation is almost 75% when $s_i = 2^{e_i} + 1$ for all $1 \le i \le n$. Since we can only guarantee that

$$\omega(s_i) = 2^{\lceil \log s_i \rceil} < 2^{1+\log s_i} = 2s_i, \qquad 1 \le i \le n$$

and

$$\sum_{i=1}^{n} \omega^2(s_i) \cdot t_i > T_m p^2,$$

we have

$$V(L) = \sum_{i=1}^{n} s_i^2 \cdot t_i > \frac{1}{4} \sum_{i=1}^{n} \omega^2(s_i) \cdot t_i > \frac{1}{4} T_m p^2.$$

Hence,

$$\mathrm{OPT}(I) \ge \frac{V(L)}{p^2} > \frac{1}{4} T_m.$$

The rest of the proof is similar to that of Theorem 1. ■

While the performance bound of algorithm LL (cf. [8]) can be very close to 1 when jobs are very small compared to the system size, it is not the case for algorithm LJF. The main disadvantage of algorithm LJF is that the performance bound 4 is still tight even for very small jobs. The reason is that no matter how small jobs are, internal fragmentation introduced by the allocation procedure remains 75% in the worst case.

Theorem 3: For any small $\varepsilon > 0$ and any large integers m and N, there exists an instance $I = \langle L, p\rangle$ such that $\mathrm{LJF}(I) > (4 - \varepsilon) \cdot \mathrm{OPT}(I)$, $\mathrm{OPT}(I) > N$, and for all $J = (s, t) \in L$, $s \le p/m$.

Proof: See the Appendix. ■

V. Average Case Performance Evaluation

Worst (or best) case performance analysis reveals the behavior of an algorithm under certain extreme cases. For example, Theorem 2 demonstrates that there exist lists of jobs such that the finish times of their LJF schedules are almost 4 times as long as the finish times of their optimal schedules. On the other hand, Theorem 1 shows that if all job sizes are exactly powers of 2, then algorithm LJF can find almost optimal schedules. However, these situations seldom occur. Hence, it is necessary to analyze the performance of algorithm LJF from an *average case* point of view. It is believed that results on average case analysis are more informative than those of worst case analysis, and hence provide a better understanding on the proposed heuristic algorithm.

There are many approaches for average case performance analysis of approximation algorithms. In addition to Monte Carlo simulations, mathematical results are obtained by assuming certain probability distributions of problem instances. For example, one goal of probabilistic analysis is to prove that certain algorithms will generate optimal or near-optimal solutions with high probability. Another way is to study

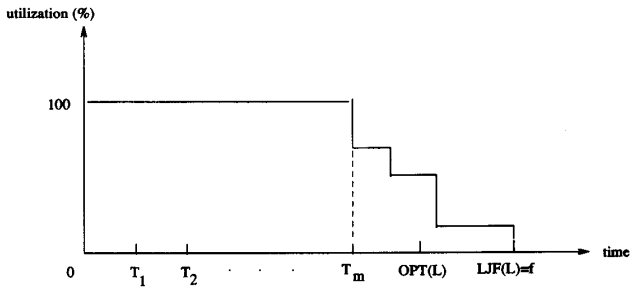


Fig. 3. Scenario in the proof of Theorem 1.

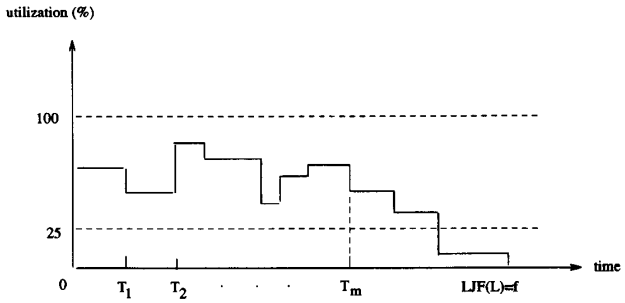


Fig. 4. Scenario in the proof of Theorem 2.

the relationship between $\boldsymbol{E}(A(I))$ and $\boldsymbol{E}(\mathrm{OPT}(I))$ (such as $\boldsymbol{E}(A(I))/\boldsymbol{E}(\mathrm{OPT}(I))$ and $\boldsymbol{E}(A(I)) - \boldsymbol{E}(\mathrm{OPT}(I))$ *et al.*) as the problem size n goes to infinity [1], where $\boldsymbol{E}(A(I))$ is the expected value of solutions produced by algorithm A, while $\boldsymbol{E}(\mathrm{OPT}(I))$ is the expected value of optimal solutions. In the rest of this section, we first define an average case performance measure to evaluate algorithm LJF; then results are presented for several different assumptions on the distribution of job size and processing time.

Let $\boldsymbol{J}_1, \boldsymbol{J}_2, \cdots, \boldsymbol{J}_k, \cdots$ be a sequence of mutually independent discrete random variables with a common distribution over the space $\Omega = [S_1, S_2] \times [T_1, T_2]$ where S_1, S_2, T_1, T_2 are positive integers. We assume that p is fixed in the following discussion. Thus, an instance of the job scheduling problem only consists of a list of jobs L_n where $L_n = (\boldsymbol{J}_1, \boldsymbol{J}_2, \cdots, \boldsymbol{J}_n)$. We are interested in the expectation of $\mathrm{LJF}(L_n)/\mathrm{OPT}(L_n)$,

$$\boldsymbol{E}\left(\frac{\mathrm{LJF}(L_n)}{\mathrm{OPT}(L_n)}\right), \tag{1}$$

especially when $n \to \infty$. However, direct computation of this quantity seems rather difficult. Instead, we derive an upper bound on $\boldsymbol{E}(\mathrm{LJF}(L_n)/\mathrm{OPT}(L_n))$.

Recall that $V(J) = s^2 t$ where $J = (s, t)$ and $V(L_n) = \sum_{i=1}^{n} V(J_i)$. Define $V_\omega(J) = \omega^2(s) \cdot t$ and $V_\omega(L_n) = \sum_{i=1}^{n} V_\omega(J_i)$. Let $\mu_1 = \boldsymbol{E}(V_\omega(\boldsymbol{J}_k))$ and $\mu_2 = \boldsymbol{E}(V(\boldsymbol{J}_k))$.

Theorem 4: For any distribution of $\boldsymbol{J}$, we have

$$\lim_{n\to\infty} \boldsymbol{E}\left(\frac{\mathrm{LJF}(L_n)}{\mathrm{OPT}(L_n)}\right) \le \frac{\mu_1}{\mu_2}.$$

Proof: It is easy to see from the last section that if no job in L_n has processing time longer than T_2, then

$$\mathrm{LJF}(L_n) < \frac{V_\omega(L_n)}{p^2} + T_2.$$

Since $\mathrm{OPT}(L_n) \ge V(L_n)/p^2$, we have

$$\frac{\mathrm{LJF}(L_n)}{\mathrm{OPT}(L_n)} < \frac{V_\omega(L_n)}{V(L_n)} + \frac{p^2 T_2}{V(L_n)}.$$

Thus,

$$\boldsymbol{E}\left(\frac{\mathrm{LJF}(L_n)}{\mathrm{OPT}(L_n)}\right) < \boldsymbol{E}\left(\frac{V_\omega(L_n)}{V(L_n)}\right) + p^2 T_2 \cdot \boldsymbol{E}\left(\frac{1}{V(L_n)}\right).$$

Clearly we always have $V(\boldsymbol{J}_k) \ge S_1^2 T_1$, i.e., $V(L_n) \ge n S_1^2 T_1$. Hence,

$$\boldsymbol{E}\left(\frac{1}{V(L_n)}\right) \le \frac{1}{n S_1^2 T_1} \to 0 \qquad \text{as } n \to \infty.$$

The implication of the above inequality is that

$$\lim_{n\to\infty} \boldsymbol{E}\left(\frac{\mathrm{LJF}(L_n)}{\mathrm{OPT}(L_n)}\right) \le \lim_{n\to\infty} \boldsymbol{E}\left(\frac{V_\omega(L_n)}{V(L_n)}\right). \tag{2}$$

Define a random variable $\boldsymbol{R}_n$ to be

$$\boldsymbol{R}_n = \frac{V_\omega(\boldsymbol{J}_1) + V_\omega(\boldsymbol{J}_2) + \cdots + V_\omega(\boldsymbol{J}_n)}{V(\boldsymbol{J}_1) + V(\boldsymbol{J}_2) + \cdots + V(\boldsymbol{J}_n)}.$$

Let $\Pr(A)$ denote the probability of event A. Define

$$p_1(n,\varepsilon) = \Pr\left\{\left|\frac{1}{n}\sum_{i=1}^{n} V_\omega(\boldsymbol{J}_i) - \mu_1\right| < \varepsilon\right\},$$
$$p_2(n,\varepsilon) = \Pr\left\{\left|\frac{1}{n}\sum_{i=1}^{n} V(\boldsymbol{J}_i) - \mu_2\right| < \varepsilon\right\}.$$

Note that

$$\mu_1 - \varepsilon < \frac{1}{n}\sum_{i=1}^{n} V_\omega(\boldsymbol{J}_i) < \mu_1 + \varepsilon$$

and

$$\mu_2 - \varepsilon < \frac{1}{n}\sum_{i=1}^{n} V(\boldsymbol{J}_i) < \mu_2 + \varepsilon$$

implies that

$$\frac{\mu_1 - \varepsilon}{\mu_2 + \varepsilon} < \boldsymbol{R}_n < \frac{\mu_1 + \varepsilon}{\mu_2 - \varepsilon}.$$

Hence,

$$\Pr\left\{\frac{\mu_1 - \varepsilon}{\mu_2 + \varepsilon} < \boldsymbol{R}_n < \frac{\mu_1 + \varepsilon}{\mu_2 - \varepsilon}\right\} \geq \Pr\left\{\left|\frac{1}{n}\sum_{i=1}^{n} V_\omega(\boldsymbol{J}_i) - \mu_1\right| < \varepsilon \quad \text{and} \quad \left|\frac{1}{n}\sum_{i=1}^{n} V(\boldsymbol{J}_i) - \mu_2\right| < \varepsilon\right\}$$
$$\geq p_1(n,\varepsilon) + p_2(n,\varepsilon) - 1.$$

The last inequality is based on the following fact. Let A and B be two events. Since $1 \geq \Pr(A \cup B) = \Pr(A) + \Pr(B) - \Pr(AB)$, we have $\Pr(AB) \geq \Pr(A) + \Pr(B) - 1$. By Khintchine's law of large numbers (cf. [2, p. 243]), for any $\varepsilon > 0$,

$$\lim_{n\to\infty} p_1(n,\varepsilon) = 1 \quad \text{and} \quad \lim_{n\to\infty} p_2(n,\varepsilon) = 1.$$

Thus, for any $\varepsilon > 0$,

$$\lim_{n\to\infty} \Pr\left\{\frac{\mu_1 - \varepsilon}{\mu_2 + \varepsilon} < \boldsymbol{R}_n < \frac{\mu_1 + \varepsilon}{\mu_2 - \varepsilon}\right\} \geq 1 + 1 - 1 = 1$$

which implies that

$$\lim_{n\to\infty} \boldsymbol{E}(\boldsymbol{R}_n) = \frac{\mu_1}{\mu_2}. \tag{3}$$

Equations (2) and (3) imply that the ratio μ_1/μ_2 gives an upper bound on the objective value in (1) to be evaluated as $n \to \infty$. ■

The remaining task is to compute μ_1/μ_2. Let $\rho_{s,t} = \Pr\{\boldsymbol{J}_k = (s,t)\}$ where $(s,t) \in \Omega$. Thus,

$$\mu_1 = \sum_{s=S_1}^{S_2}\sum_{t=T_1}^{T_2} \rho_{s,t}\omega^2(s)\cdot t$$

and

$$\mu_2 = \sum_{s=S_1}^{S_2}\sum_{t=T_1}^{T_2} \rho_{s,t}s^2 t.$$

To evaluate $\mu_i, i = 1,2$, we need information about ρ. For example, we may have the following set of assumptions.

A1. Job sizes are uniformly distributed in the range $[S_1, S_2]$ where $1 \leq S_1 \ll S_2$. Let ρ_s be the probability that the size of $\boldsymbol{J}_k$ is s where $s \in [S_1, S_2]$.

A2. Job processing times are distributed in the range $[T_1, T_2]$. Let ρ_t be the probability that the processing time of $\boldsymbol{J}_k$ is t where $t \in [T_1, T_2]$.

A3. Distributions of job size and processing time are independent, i.e., for all $(s,t) \in \Omega, \rho_{s,t} = \rho_s \cdot \rho_t$.

It is obvious that under assumptions A1, A2, and A3,

$$\frac{\mu_1}{\mu_2} = \frac{\sigma_\omega(S_1,S_2)}{\sigma(S_1,S_2)} \tag{4}$$

where

$$\sigma_\omega(S_1,S_2) = \sum_{s=S_1}^{S_2} \omega^2(s) \quad \text{and} \quad \sigma(S_1,S_2) = \sum_{s=S_1}^{S_2} s^2.$$

We observe that

$$\sigma(S_1,S_2) = \sum_{s=1}^{S_2} s^2 - \sum_{s=1}^{S_1-1} s^2$$
$$= \frac{S_2^3}{3} + \frac{S_2^2}{2} + \frac{S_2}{6} - \frac{(S_1-1)^3}{3} - \frac{(S_1-1)^2}{2} - \frac{S_1-1}{6}. \tag{5}$$

Since

$$\omega(2^i+1) = \omega(2^i+2) = \cdots = \omega(2^{i+1}) = 2^{i+1}, \qquad i \geq 0,$$

let G_i [see Fig. 5(a)] be defined as

$$G_i = \{2^i+1, 2^i+2, \cdots, 2^{i+1}\}, \qquad i \geq 0.$$

In addition, let

$$g_i = \sum_{a\in G_i} \omega^2(a) = 2^i \cdot (2^{i+1})^2 = 4\cdot 8^i,$$

and for $k_2 \geq k_1$,

$$f(k_1,k_2) = \sum_{i=k_1}^{k_2} g_i$$
$$= 4\cdot\left(8^{k_1} + \cdots + 8^{k_2}\right) = \frac{4}{7}\left(8^{k_2+1} - 8^{k_1}\right).$$

Let $S_1 = 2^{k_1}(1+\delta_1)$ and $S_2 = 2^{k_2}(1+\delta_2)$ where $0 \leq \delta_i < 1, i = 1,2$. The value δ_i represents the relative distance of S_i from the nearest power of 2 which is less than or equal to S_i. We distinguish two cases in evaluating $\sigma_\omega(S_1,S_2)$.

Case 1. When $\delta_1 = 0$, as illustrated in Fig. 5(b),

$$\sigma_\omega(S_1,S_2) = (2^{k_1})^2 + f(k_1,k_2-1) + \delta_2\cdot g_{k_2}$$
$$= 4^{k_1} + \frac{4}{7}\left(8^{k_2} - 8^{k_1}\right) + 4\delta_2\cdot 8^{k_2}. \tag{6}$$

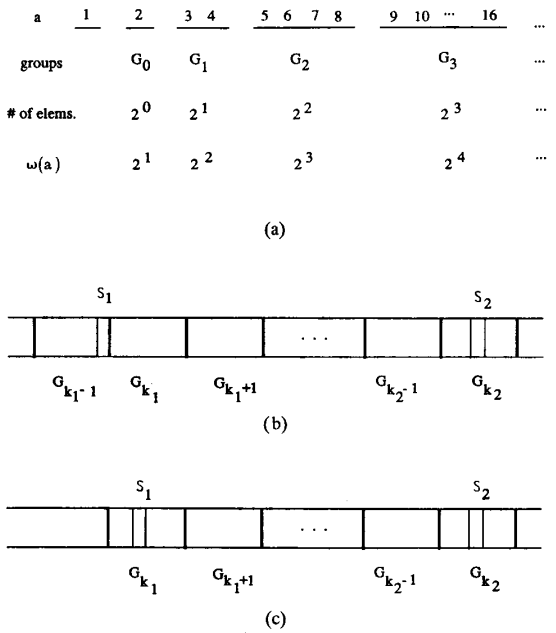


Fig. 5. Computation of $\sigma_\omega(S_1, S_2)$. (a) Groups of ω values. (b) Case 1. (c) Case 2.

Case 2. When $\delta_1 \neq 0$, as illustrated in Fig. 5(c),

$$\begin{aligned}\sigma_\omega(S_1, S_2) &= (2^{k_1+1})^2 + (1-\delta_1)g_{k_1} \\ &\quad + f(k_1+1, k_2-1) + \delta_2 \cdot g_{k_2} \\ &= 4^{k_1+1} + \frac{4}{7}\left(8^{k_2} - 8^{k_1}\right) \\ &\quad + 4(\delta_2 \cdot 8^{k_2} - \delta_1 \cdot 8^{k_1}). \end{aligned} \tag{7}$$

We observe that the first term in both (6) and (7) are very small when compared to other terms and hence negligible. Then (6) and (7) can be unified as

$$\sigma_\omega(S_1, S_2) \approx 4 \cdot \left[\left(\frac{1}{7} + \delta_2\right) \cdot 8^{k_2} - \left(\frac{1}{7} + \delta_1\right) \cdot 8^{k_1}\right].$$

Since

$$8^{k_i} = (2^{k_i})^3 = \frac{S_i^3}{(1+\delta_i)^3}, \qquad i = 1, 2,$$

we have

$$\begin{aligned}&\sigma_\omega(S_1, S_2) \\ &\approx 4 \cdot \left[\left(\frac{1}{7} + \delta_2\right) \cdot \frac{S_2^3}{(1+\delta_2)^3} - \left(\frac{1}{7} + \delta_1\right) \cdot \frac{S_1^3}{(1+\delta_1)^3}\right].\end{aligned} \tag{8}$$

Furthermore, if $S_1 \ll S_2$, then only terms of S_2^3 in (5) and (8) are dominant. Therefore,

$$\sigma(S_1, S_2) \approx \frac{1}{3} S_2^3 \tag{9}$$

and

$$\sigma_\omega(S_1, S_2) \approx \frac{4}{7} \cdot \frac{1+7\delta_2}{(1+\delta_2)^3} \cdot S_2^3. \tag{10}$$

Substituting results in (9) and (10) into (4), we have

$$\frac{\mu_1}{\mu_2} \approx R_1(\delta_2) = \frac{12}{7} \cdot \frac{1+7\delta_2}{(1+\delta_2)^3}.$$

That is, only δ_2 affects the ratio μ_1/μ_2. Note that

$$\frac{d}{d\delta_2} R_1(\delta_2) = \frac{24}{7} \cdot \frac{2-7\delta_2}{(1+\delta_2)^4}.$$

Hence, $R_1(\delta_2)$ has its maximum value when $\delta_2 = 2/7$ and minimum value when $\delta_2 = 0$, i.e.,

$$1.71 \approx \frac{12}{7} \leq R_1(\delta_2) \leq \frac{196}{81} \approx 2.42. \tag{11}$$

Similar results can also be obtained based on other assumptions such as

A4. Job sizes obey a truncated exponential distribution in the range $[1, m]$, i.e.,

$$\rho_s = \frac{e^\lambda - 1}{1 - e^{-m\lambda}} \cdot e^{-\lambda s}, \qquad 1 \leq s \leq m,$$

where $1 \ll 1/\lambda \ll m$.

A5. Job processing times satisfy a linear function of job sizes, i.e., $\rho_t = \rho_s$ if $t = c_1 s + c_2$ where c_1, c_2 are constants.

It should be noted that

$$\frac{\mu_1}{\mu_2} = \frac{1}{1 - F_{\text{internal}}}$$

where F_{internal} is the expected internal fragmentation of the system and is defined as

$$F_{\text{internal}} = 1 - \frac{\sum_{s=S_1}^{S_2} \sum_{t=T_1}^{T_2} \rho_{s,t} s^2 t}{\sum_{s=S_1}^{S_2} \sum_{t=T_1}^{T_2} \rho_{s,t} \omega^2(s) \cdot t}.$$

By directly applying results about F_{internal} obtained in [9], we know that under assumptions A2, A3, and A4,

$$\frac{\mu_1}{\mu_2} \approx \frac{3}{2ln2} \approx 2.164, \tag{12}$$

and under assumptions A1 and A5,

$$\frac{\mu_1}{\mu_2} \approx R_2(\delta_2) = \frac{8}{5} \cdot \frac{5\delta_2^2 + 10\delta_2 + 1}{(1+\delta_2)^4}$$

where

$$1.6 \leq R_2(\delta_2) \leq 2.5. \tag{13}$$

VI. On-Line and Dynamic Job Scheduling

Algorithm LJF is a static job scheduling algorithm in the sense that all information about the job list L are available before it generates the schedules of jobs in L. The separation of the system partitioning scheme from the job scheduling policy in algorithm LJF makes it easy to adapt LJF for on-line and dynamic job scheduling. By *on-line* scheduling we mean that an algorithm produces schedules of jobs in the given order of L, i.e., it generates the schedule $(B(x_i, y_i, k_i), S_i)$ for job J_i by only using information of jobs $J_1, J_2, \cdots, J_i$.

In other words, it should give a schedule of J_i without any knowledge of jobs $J_{i+1}, J_{i+2}, \cdots, J_n$, although these jobs are in the waiting job list and hence may be checked. In real applications, jobs may arrive at unpredictable times, and this is the so called *dynamic* scheduling. Note that if the differences among job arrival times are so small that the job list is nonempty whenever the scheduler checks the list, then dynamic scheduling is similar to on-line scheduling except that jobs may not need to be scheduled in the given order. In fact, an on-line scheduler is basically a first come first serve scheduler in dynamic scheduling. Since an on-line scheduler does not require information about future jobs, it is more realistic than a static scheduling algorithm.

Note that the only step in algorithm LJF which requires global information of list L is Step 2. An On-Line Scheduler (OLS) may be obtained by just removing this global sorting step. In a real system, OLS may be a nonterminating process which has two states, namely, *active* and *sleeping*. Basically, it removes jobs from the front of the job waiting list L and produces their schedules until there is no job in the list or overflow occurs. At this time, process OLS goes to sleep. It wakes up when one (or both) of the following two events occurs.

1) A new job arrives when L is empty.
2) A running job J finishes.

In the first case, process OLS resumes its active state. In the second case, process OLS calls the reclamation procedure and then resumes the active state. Clearly, process OLS is able to handle both on-line and dynamic scheduling situations.

Now let us look at the performance of process OLS for the on-line case. Since job sizes in L cannot be transformed into a P2NI-sequence just by rounding, Lemma 4 is not applicable in this case, which implies that there will be external fragmentation upon overflow. This observation reveals the fact that process OLS has a poor worst case performance as shown in the following example. Let $L = (J_1, J_2, \cdots, J_n)$ where $n = 2p^2, J_i = (p, 1)$ for $i = 1, 3, 5, \cdots, n-1$, and $J_i = (1, t)$ for $i = 2, 4, 6, \cdots, n$. Clearly, in the OLS schedule of L, jobs are executed sequentially. Thus, $\mathrm{OLS}(L) = p^2(1+t)$. However, in an optimal schedule, all even number indexed jobs are executed simultaneously, i.e., $\mathrm{OPT}(L) = p^2 + t$. Thus, $\mathrm{OLS}(L)/\mathrm{OPT}(L) = p^2(1+t)/(p^2+t)$, which can be arbitrarily close to p^2 as $t \to \infty$.

However, these contrived instances seldom occur in real applications. Thus, it is more reasonable to examine the average case performance of process OLS. We suggest using *utilization* as a performance measure for on-line job scheduling algorithms. For a finite job list $L = (J_1, J_2, \cdots, J_n)$, where $J_i = (s_i, t_i), 1 \le i \le n$, the utilization of the system using algorithm A is defined as

$$U_A(L) = \frac{\sum_{i=1}^{n} s_i^2 t_i}{Tp^2} \tag{14}$$

where T is the finish time of the schedule of L generated by algorithm A and p is the system size. If J_i's are independent random variables with a common distribution, then the average utilization over all lists of length n, $E_n = \boldsymbol{E}(U_A(L))$, is determined by n, the algorithm A, and distributions of job sizes and execution times. For a given distribution and sufficiently large n, the system utilization is solely affected by the scheduling algorithm (including the system partitioning scheme). Thus, $\lim_{n\to\infty} E_n$ is an appropriate measure of the performance of an on-line scheduling algorithm. Note that this definition is also applicable to a dynamic scheduling situation if process OLS never goes to sleep because of an empty job waiting list L. If this assumption is not valid, e.g., in a lightly used system, then (14) is not a proper measure since low utilization of the system may be due to not enough jobs to be processed, instead of an inappropriate scheduler.

Evaluation of process OLS is done by extensive simulation where it is reported in [9] that if probability distributions of job sizes and processing times are independent and processing times are uniformly distributed, then the average utilization is roughly in the range 35–55% for both uniform and exponential distributions of job sizes. However, as already been noted in [9], external fragmentation only accounts for a very small fraction of the loss in the computation power. Thus, when job sizes are powers of 2, i.e., there is no internal fragmentation, the average system utilization is as high as in the range 85–90%. Recently, worst case analysis of process OLS using a different performance metric has been reported in [10] for both external and internal fragmentations.

VII. Summary

The job scheduling problem in partitionable mesh-connected systems is considered in this paper for the case when jobs require square meshes and the system is a square mesh of size a power of 2. the heuristic algorithm LJF of time complexity $O(n(\log n + \log p))$ is proposed which adopts the largest job first scheduling policy and a two-dimensional buddy system as the system partitioning scheme. We analyzed the performance of algorithm LJF. In the worst case, it produces a schedule four times longer than an optimal schedule due to large internal fragmentation introduced by the 2DBS. But on the average, as indicated in (11)–(13), schedules generated by algorithm LJF are only about twice longer than optimal schedules. In addition, Theorem 4 shows that for any job distribution, the expected performance of algorithm LJF depends only on the ratio of the average allocated processing power over the average requested processing power. We also stress that it is very easy to modify algorithm LJF for on-line and dynamic job scheduling due to the separation of the system partitioning scheme from the job scheduling policy.

Appendix
Proof of Theorem 3

We construct an instance $I = \langle L, p \rangle$ as follows. Let $p = 2^{r+k+1}$ and L be a list of $n = 4^k v$ identical jobs with size $2^r + 1$ and processing time t. Integers r, k, v, and t are to be defined below.

Consider the schedule produced by algorithm LJF. Since all jobs are of the same size and they are all rounded to 2^{r+1}, a total of $(p/2^{r+1})^2 = (2^k)^2 = 4^k$ jobs are executed simultaneously. Also because all jobs have the same

processing time t, these 4^k jobs finish at the same time. Thus, the n jobs are divided into $n/4^k = v$ groups and the finish time of the LJF schedule is $\mathrm{LJF}(I) = vt$. The scenario is similar in the optimal schedule. Each group contains $\lfloor p/(2^r+1) \rfloor^2$ jobs. Note that

$$\begin{aligned}\left\lfloor \frac{p}{2^r+1} \right\rfloor^2 &= \left\lfloor \frac{2^{r+k+1}}{2^r+1} \right\rfloor^2 \\ &> \left(\frac{2^{k+1}}{1+2^{-r}} - 1 \right)^2 \\ &= \frac{4^{k+1}}{(1+2^{-r})^2} - \frac{2^{k+2}}{1+2^{-r}} + 1 \\ &> 4^k \cdot \left[\frac{4}{(1+2^{-r})^2} - \frac{1}{2^{k-2}} \right].\end{aligned}$$

For any $\varepsilon > 0$, if we choose r and k such that

$$\frac{4}{(1+2^{-r})^2} > 4 - \frac{\varepsilon}{2} \quad \text{and} \quad \frac{1}{2^{k-2}} < \frac{\varepsilon}{2},$$

that is,

$$r > -\log\left[\sqrt{\frac{8}{8-\varepsilon}} - 1 \right] \quad \text{and} \quad k > 2 + \log\frac{2}{\varepsilon},$$

then we have for some $\delta > 0$, $\lfloor p/(2^r+1) \rfloor^2 = (4-\delta)\cdot 4^k > (4-\varepsilon)\cdot 4^k$. Thus, the finish time of the optimal schedule is

$$\begin{aligned}\mathrm{OPT}(I) &= \left\lceil \frac{n}{\lfloor p/(2^r+1) \rfloor^2} \right\rceil \cdot t \\ &< \left(\frac{4^k v}{\lfloor p/(2^r+1) \rfloor^2} + 1 \right) \cdot t = \left(\frac{v}{4-\delta} + 1 \right) \cdot t.\end{aligned}$$

Therefore, to guarantee that

$$\frac{\mathrm{LJF}(I)}{\mathrm{OPT}(I)} > \frac{(4-\delta)v}{v+(4-\delta)} > 4 - \varepsilon,$$

we just require $v > (4-\delta)(4-\varepsilon)/(\varepsilon-\delta)$.

Furthermore, we note that $V(L) = n(2^r+1)^2 t > 4^r nt$. Hence,

$$\mathrm{OPT}(I) \geq \frac{V(L)}{p^2} > \frac{4^r nt}{4^{r+k+1}} = \frac{vt}{4}.$$

Thus, we can choose t and v sufficiently large such that $\mathrm{OPT}(I) > N$.

Finally, since k can be arbitrarily large, we can select k such that $2^k > m$ for any large integer $m > 1$. Then

$$\frac{p}{m} = \frac{2^{r+1+k}}{m} > 2^{r+1} > 2^r + 1 = s_i, \qquad 1 \leq i \leq n.$$

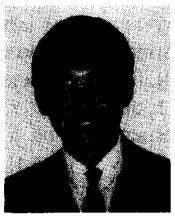
In other words, job sizes in L can be arbitrarily small compared to p.

ACKNOWLEDGMENT

The authors would like to thank the anonymous referees for reading the manuscript carefully. Their comments lead to improvements on the correctness and the organization of the manuscript.

REFERENCES

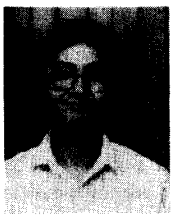
[1] E. G. Coffman, Jr., G. S. Lueker, and A. H. G. Rinnooy Kan, "Asymptotic methods in the probabilistic analysis of sequencing and packing heuristics," *Management Sci.*, pp. 266–290, Mar. 1988.
[2] W. Feller, *An Introduction to Probability Theory and Its Applications*, vol. 1, 3rd ed. New York: Wiley, 1968.
[3] M. R. Garey and D. S. Johnson, "Approximation algorithms for combinatorial problems: An annotated bibliography," in *Algorithms and Complexity: New Directions and Recent Results*, J. F. Traub, ed. New York: Academic, 1976, pp. 41–52.
[4] ——, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA: Freeman, 1979.
[5] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*. Rockville, MD: Computer Science Press, 1978.
[6] W. H. Lee and M. Malek, "MOPAC: A partitionable and reconfigurable multicomputer array," in *Proc. 12th Int. Conf. Parallel Processing*, 1983, pp. 506–510.
[7] K. Li and K. H. Cheng, "Complexity of resource allocation and job scheduling problems in partitionable mesh connected systems," in *Proc. 1st IEEE Symp. Parallel Distributed Processing*, May 1989, pp. 358–365.
[8] ——, "Static job scheduling in partitionable mesh connected systems," *J. Parallel Distributed Comput.*, vol. 10, no. 2, pp. 152–159, Oct. 1990.
[9] ——, "A two dimensional buddy system for dynamic resource allocation in a partitionable mesh connected system," in *Proc. 18th ACM Comput. Sci. Conf.*, Feb. 1990, pp. 22–28; *J. Parallel Distributed Comput.*, vol. 12, no. 1, pp. 79–83, May 1991.
[10] ——, "Worst case performance analysis of the two dimensional binary buddy system," *Int. J. Comput. Math.*, vol. 38, pp. 123–132, 1991.
[11] J. L. Peterson and T. A. Norman, "Buddy systems," *Commun. ACM*, vol. 20, no. 6, pp. 421–431, 1977.



Keqin Li (S'90–M'91) was born in Shanghai, China, on May 26, 1963. He received the Bachelor of Engineering degree in computer science from Tsinghua University, Beijing, China, in 1985, and the Ph.D. degree in computer science from the University of Houston, Houston, TX in 1990.

In August 1990 he joined the faculty of the Department of Mathematics and Computer Science, State University of New York, New Paltz, NY, where he is currently an Assistant Professor. He was a Teaching Assistant during 1987–1988 and a Research Assistant during 1988–1990 at the University of Houston. His research interests include design and analysis of algorithms, parallel and distributed computing, and theory of computation.

Dr. Li is a member of the Association for Computing Machinery, the IEEE Computer Society, and the Society for Industrial and Applied Mathematics.



Kam-Hoi Cheng received the B.Sc. degree in computer science from the University of Manitoba, Winnipeg, Man., Canada, in 1980, and the Ph.D. degree in computer science from the University of Minnesota, Minneapolis, MN in 1985.

He is currently an Associate Professor of Computer Science at the University of Houston, Houston, TX. His research interests include VLSI systems, parallel computing, parallel computer architecture, and parallel algorithms.