[16] Ravi Jain, John Werth, and J. C. Browne. A general model for scheduling of parallel computations and its application to parallel I/O operations. In *Proc. Intl. Conf. Par. Proc.*, 1991.

[17] A. Kandappan. Data allocation and scheduling for parallel I/O systems. Master's thesis, Dept. of Elect. and Comp. Eng., Univ. of Texas at Austin, 1990.

[18] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, 1976.

[19] P. L'Eculyer. *Comm. ACM*, 31:742–774, June 1988.

[20] T. Lovett and S. Thakkar. The Symmetry multiprocessor system. In *Proc. Intl. Conf. Par. Proc.*, pages 303–310, 1988.

[21] T. N. Mudge, J. P. Hayes, and D. C. Winsor. Multiple bus architectures. *Computer*, 20(6):42–48, June 1987.

[22] G. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM research parallel processor (RP3): Introduction and architecture. In *Proc. Intl. Conf. Par. Proc.*, pages 764–771, 1985.

[23] Kiran Somalwar. Data transfer scheduling. Technical Report TR-88-31, Univ. Texas at Austin, Dept. of Comp. Sci., 1988.

[24] A. W. Wilson, Jr. Hierarchical cache/bus architecture for shared memory multiprocessors. In *14th Intl. Symp. Comp. Arch.*, pages 244–252, 1987.

[25] W. Zhao, K. Ramamritham, and J. A. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Trans. Comp.*, page 949, Aug. 1987.

[2] Claude Berge. *Graphs*. North-Holland, 1985.

[3] G. Bongiovanni, D. Coppersmith, and C. K. Wong. An optimum time slot assignment algorithm for an SS/TDMA system with variable number of transponders. *IEEE Trans. Comm.*, 29(5):721–726, May 1981.

[4] H. Boral and D. J. DeWitt. Database machines: An idea whose time has passed? A critique of the future of database machines. In *Third Intl. Workshop on Database Machines*, pages 166–187, 1983.

[5] J. C. Browne, G. E. Onstott, P. L. Soffa, Ron Goering, S. Sivaramakrishnan, Harish Balan, and Kiran Somalwar. Design and evaluation of external memory architectures for multiprocessor computer systems: Second quarter report to IBM Yorktown Heights Research Lab. Technical report, Univ. Texas at Austin, Dept. of Comp. Sci., 1987. Available from J. C. Browne.

[6] E. F. Codd. Multiprogram scheduling. *Comm. ACM*, 3, June 1960.

[7] R. Cole and J. Hopcroft. On edge coloring bipartite graphs. *SIAM J. Computing*, 11(3):540–546, 1982.

[8] P. J. Denning. Effects of scheduling on file memory operations. In *Proc. AFIPS Spring Joint Comp. Conf.*, pages 9–21, 1967.

[9] H. Gabow. Using euler partitions to edge color bipartite multigraphs. *Intl. J. Computer and Inf. Sci.*, 5:345–355, 1976.

[10] H. Gabow and O. Kariv. Algorithms for edge coloring bipartite graphs and multigraphs. *SIAM J. Computing*, 11(1):117–129, 1982.

[11] T. Gonzalez and S. Sahni. Open shop scheduling to minimize finish time. *J. Ass. Comp. Mach.*, 23:665–679, 1976.

[12] B. Hancock. Multiprocessors are NOT always better. *Digital Review*, page 59, Dec. 2, 1991.

[13] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, 1990.

[14] J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2:225–231, 1973.

[15] Ravi Jain. Scheduling I/O in parallel computing environments. Unpublished manuscript, Dec. 1990.

Algorithm **A1** is faster than **KT**. When $K = 1$, **A2** is faster than **KT**, **A1**, **A3** and the algorithm in Somalwar [23]. Algorithm **A2** is faster than both **A1** and **KT** when the maximum edge weight $K$ is small, relative to $n^{1.5}$, roughly. Algorithm **A3** is faster than **KT** and **A1** for a much larger range of values for $K$, i.e., when $\log K$ is small relative to $n^{1.5}$.

# 8 Conclusions

We have developed and characterized algorithms for simultaneous resource scheduling. We have shown how thiese algorithms apply to scheduling parallel I/O operations for a significant class of problems and a common class of parallel architectures. The applications include 3D visualization and decision support systems using large databases. The multiprocessor architectures include the Sequent [20], Encore Multimax [24] and the IBM RP3 [22]. We have carried out an experimental evaluation which shows, for the situations studied, the resulting improvement in algorithm execution time over the previously available algorithm.

Further experimental evaluation of the performance of this algorithm is currently in progress. This experimental work extends the previous related work of Somalwar [23], Kandappan [17] and Balan [1]. The evaluation is based on performance measurements as well as simulations. We are also investigating the extension of the algorithm to problems where there are precedence constraints among the I/O requests and to problems involving a larger class of I/O architectures. Finally, multiple resource scheduling has many other possible applications in parallel system architectures, including those arising from the presence of mutual exclusion and precedence constraints between tasks [16], which are under investigation.

# Acknowledgements

The first author wishes to thank Prof. Galen Sasaki for many useful and enlightening discussions which improved this paper. The authors thank the referees for their comments.

# References

[1] H. Balan. Master's thesis, Dept. of Elect. and Comp. Eng., Univ. of Texas at Austin, 1990.

Figure 3: Running time versus number of parallel transfers possible for $m = 1000$

Since **A3** is framed in terms of optimal edge-coloring of bipartite multigraphs, we review this literature briefly. First consider the optimal edge-coloring of bipartite graphs with unit-weight edges. An algorithm taking time Gabow and Kariv [10] present algorithms taking time $O(m \log^2 n)$ and $O(n^2 \log n)$. Finally, Cole and Hopcroft [7] improve the time to $O(m \log n)$. For optimal edge-coloring of bipartite graphs with edge weights that are not all unity, Gonzalez and Sahni [11] present an algorithm that takes time $O(m^2)$. Gabow and Kariv [10] show an algorithm that takes time $O(nm \log K)$, where $K$ is the largest edge weight; this is faster than the algorithm of Gonzalez and Sahni [11] for a large class of graphs, i.e., whenever $n \log K = o(m)$.

To our knowledge, the only results on optimal $k$-coloring of bipartite graphs are in Bongiovanni et al [3] and Somalwar [23]. Algorithm **KT** of [3] operates on graphs with non-unit edge weights, and, as shown in Theorem 3, takes time $O(n^5)$. The algorithm of Somalwar [23] is applicable to unit edge weight graphs, and takes time $O(n^{1.5} p \log p)$, where $p$ is the bound of the graph, i.e., takes time $O(n^{3.5} \log n)$.

Algorithms **A1 - A3** generalize previous edge-coloring algorithms by allowing non-unit edge weights as well as a restriction on the number of edges that may be colored with a single color. Algorithms **A1 - A3** are also applicable to more general problem classes than the algorithm in Somalwar [23].

Figure 2: Running time versus number of transfers for $k = 4$

In Fig. 3 we plot the CPU time taken by **KT** and **A2** as the number of parallel transfers possible, $k$, increases and the number of data transfers, $m$, is kept fixed. As expected, both algorithms run faster as $k$ increases. Informally, this is because more transfers take place at each time slot and hence the schedule length decreases, which in turn means that fewer matchings need to be calculated. Further, as expected, we observe that as $k$ increases the benefits of **A2** over **KT** decrease.

We are currently engaged in evaluating the performance of I/O scheduling algorithms for non-unit-length transfers. In this case we are comparing the performance of **KT** with the algorithm for large transfer lengths, **A3**. We are also continuing our study of the behavior of **A2** for unit-length transfers. Some of this work has been done by estimating the I/O demands of specific applications, such as split-step Fourier migration and volume visualization, and preliminary results have been reported elswhere [23].

# 7 Previous work

Algorithm **A3** provides a method for scheduling parallel I/O in multiple-bus parallel computers, and is an improvement in time complexity over algorithm **KT** for a large class of problems.

information, e.g. computer-aided tomography (CAT) scans, or oil prospecting information, e.g. seismic data from acoustical depth soundings, etc. The database is processed and stored at a parallel computer site, and users view the images by requesting image files to be displayed on their graphics workstations. The parallel computer is a shared-bus system, in which processors and disks are connected to a set of common buses (or a single high-speed system bus that is shared in a time-multiplexed fashion), which allow multiple I/O transfers to proceed in parallel. In order to provide a reasonable response time for the users, the workstations are also connected to the common buses. A user request for an image file is processed by the CPUs at the parallel computer, and results in image data being transferred from the system disks to the user's workstation across the common buses. The workstations off-load some low-level image-processing tasks from the parallel computer, such as rendering, shading, etc.

In this scenario, the parallel computer's operating system batches the image file requests and schedules the resulting I/O transfers. We focus on the case of equal-length transfers. For this scenario, we assume that the number of disks allowing parallel I/O is 16, and the number of workstations is 64, these quantities being reasonable considering the current scalability problems of shared-bus architectures. We also assume that between 4 and 12 data transfers may proceed in parallel. The number of data transfers requested varies between 100 and 1000.

For unit-length transfers, our primary contribution has been the development of the **A2** algorithm. This algorithm relies on a divide-and-conquer approach to obtain a substantial theoretical improvement in worst-case time complexity over **KT**. We have compared the running time of **A2** and **KT** experimentally. Casting the scenario sketched above in terms of the algorithms, we consider as inputs bipartite graphs with 64 vertices in one partition and 16 in the other, and $k$ fixed at either 4, 8, or 12. Edges are generated by a pseudo-random number generator [19], with the number of edges, $m$, being fixed at 100, 200, 400, 800 or 1000. Note that for this situation, the bound of the graph, i.e. optimal length of the schedule, is given by $\lceil m/k \rceil$.

The results of running these experiments on a Sun Sparc 2 workstation are shown in the plots below. In Fig. 2 we plot the CPU time taken by **KT** and **A2** as the number of transfers, $m$, increases and the number of transfers that may be performed simultaneously, $k$, is kept fixed at 4. As expected, both algorithms take longer to run as the number of transfers increases. Informally, this is because the schedule length becomes longer and hence more matchings need to be calculated. Further, as $m$ increases **A2** out-performs **KT** by greater amounts. Informally, this is because the benefits of the divide-and-conquer approach are reaped more often when calculating a longer schedule.

critical $k$-coloring and $b$ is its duration.

0. s := $\langle\ \rangle$;
1. Add dummy traffic to make $G$ $k$-complete with respect to $q$.
2. G' := $k$-**fill**(G);
3. C := **weighted-edge-color**(G'); /* C is an edge coloring of G'*/
4. **for** each color $c \in C$
5. M', b := edges colored by c, duration c is used; /* M' is max. matching on G' */
6. M := M' $\cap$ E;
7. s := s || $\langle$ (M, b) $\rangle$;
8. **end for**
9. Delete dummy traffic from $s$.
10. **end A3**.

**Theorem 7.** Algorithm **A3** takes time $O(n^3(\log n + \log K))$.

*proof.* As noted earlier, $k$-**fill** takes time $O(n)$. The weighted edge coloring algorithm [10] takes time $O(|V||E|\log J)$ for a weighted bipartite graph with largest edge weight $J$. In **A3** we apply this algorithm to $G'$, where $|V| = 2n - k$, $|E| \leq (2n-k)^2 = O(n^2)$, and $J = K' \leq q \leq mK$. Thus the total time is $O(n^3(\log n + \log K))$ to obtain an edge-coloring of $G'$. Since the coloring algorithm [10] uses $O(|E|\log J)$ colors, the number of iterations of the **for** loop is $O(n^2(\log n + \log K))$. Since $|M'| = 2n - k$, lines 5 - 7 can be implemented in time $O(n)$, and thus the **for** loop takes time $O(n^3(\log n + \log K))$. □

Algorithm **A3** is faster than **KT** and **A1** for a large class of graphs, i.e., when $\log K$ is small relative to $n^{1.5}$. Note that **A2** is still faster than **A3** for $K = 1$ or when $K$ is bounded by a small constant.

# 6 Experimental evaluation

In this section we present the results of a preliminary experimental evaluation of the algorithms described in this paper. This work has confirmed that our approach provides results that are superior to the **KT** algorithm for the situations studied. It has also led to the investigation of even faster algorithms for I/O scheduling, which are heuristic in nature [15], although these are not discussed here as they are outside the scope of this paper.

Consider a scenario where users, who may be physicians, health care workers, scientists, etc., need to share and access a large image database. The images may consist of medical

$+ (\log m)cn^{.5}m$, for all $m > m'$, i.e., $T(m,m) = O(n^{.5}m \log n)$. Therefore, the total time $T(q,m) = O(Kmn^{.5} \log n)$. □

Thus **A2** is superior to **KT** for problem classes where $K$ is small relative to $n^{4.5}/(m \log n)$, i.e., say smaller than $n^2$. **A2** is superior to **A1** when $K$ is small relative to $n^4/(m \log n)$, say smaller than $n^{1.5}$.

# 5 An algorithm for large transfer lengths

While **A2** is satisfactory for problems where $K$ increases at a modest rate relative to $n$, it does not handle problems with large transfer lengths well. In this section we describe how an algorithm for edge coloring of weighted bipartite graphs with $k = n$ [10], can be applied for $IOS$ when $k \leq n$. The key observation is that deleting a maximal matching from a $k$-filled graph (see sec. 3) leaves a $k$-filled graph.

Consider a weighted bipartite graph $G = (A, B, E)$ with weight $W$, maximum vertex weight $w$, maximum edge weight $K$, and bound $q$. By Lemma 4, it can be converted to a $k$-complete graph with respect to $q$ by adding dummy traffic, and then to a $k$-filled graph $G'$ with respect to $q$ using the $k$-**fill** algorithm (see sec. 3). This conversion is performed in **KT** and **A1**, and takes time $O(m + n)$.

Let $G' = (A', B', E')$ with $|A| = |B| = 2n - k$, weight $W' = (2n - k)q$, maximum edge weight $K'$, and all vertex weights equal to $q$. In both **KT** and **A1**, a maximal matching $M'$ of size $2n - k$ on $G'$ is obtained, and the $k$ edges of $M = M' \cap E$ are deleted from $G$. This results in a new graph $H$ which is $k$-complete with respect to $q - 1$ but not $k$-filled with respect to $q - 1$. It is necessary to invoke $k$-**fill** on $H$ before obtaining the next critical $k$-matching. We observe that if all $2n - k$ edges of $M'$ are deleted from $G'$, the result is a graph $G''$ that is $k$-filled with respect to $q - 1$, obviating the need for $k$-filling again before obtaining the next matching.

The argument above leads to the following algorithm, where an edge-coloring on $G'$ is used to obtain a series of maximal matchings on $G'$, which are then pruned to obtain a series of critical $k$-matchings on $G$. The weighted edge-coloring algorithm of Gabow and Kariv [10] is used to edge-color $G'$.

**Algorithm A3.**
**Input:** Bipartite graph $G = (A, B, E)$ with bound $q$ and maximum edge weight $K$.
**Output:** A minimum-length $k$-coloring of $G$, as a sequence $s$ of $(M, b)$ pairs, where $M$ is a

4. Add dummy traffic to make $G$ $k$-complete with respect to $q$.
5. **A2-color**(G); /* Updates $s$ */
6. Delete dummy traffic from $s$.
7. **end A2.**

**Procedure A2-color**(G)
1. **if** q is odd **then**
2. M := **CKM**(G); /* **CKM** will find a critical $k$-matching */
3. s, E, q := s || ⟨ (M, 1) ⟩, E - M, q - 1;
4. **end if**
5. $H$, $H'$ = **ED**(G);
6. **A2-color**(H);
7. **A2-color**(H');
8. **end A2-color**

**Theorem 5.** Algorithm **A2** finds a minimum-length schedule for an instance of $IOS$.

*proof.* We need to show that **A2** generates a minimum-length $k$-coloring of $G$. Lines 1 - 4 and 6 - 7 of **A2** are similar to **KT**. Procedure **A2-color** is called with $G$, a bipartite graph $k$-complete with respect to its bound $q$, as its argument. Thus, from Theorem 1, $G$ contains a critical $k$-matching, which can be found by the procedure **CKM** outlined in the proof of Theorem 1. If $q$ is odd, a critical $k$-matching is found in line 2 of **A2-color**. Now the graph $G$ with edge set $E - M$ has an even bound $q - 1$, and from a corollary of Theorem 2, is also $k$-complete with respect to its bound. Thus from Lemma 6, **ED** generates an Euler division of $G$ into $H$, $H'$, which are $k$-complete graphs with respect to their bounds $\lfloor q/2 \rfloor$. **A2-color** is applied to them recursively, and by the induction hypothesis generates two $k$-colorings of length $\lfloor q/2 \rfloor$. Together with the $k$-matching generated if $q$ is odd, we obtain a $k$-coloring in $s$ of length $q$. □

**Theorem 6.** Algorithm **A2** runs in time $O(Kmn^{.5} \log n)$, where $K$ is the maximum edge weight of $G$.

*proof.* Procedure **CKM** takes time $O(n^{.5}m)$ using the **MaxMatch** algorithm of Hopcroft and Karp [14]. By Lemma 7, each invocation of **ED** takes time $O(n+m)$. As long as $K > 1$, **ED** results in two subgraphs with bound $\lfloor q/2 \rfloor$, each having upto $n$ vertices and $m$ edges. When $K = 1$, **ED** results in two subgraphs with bound $\lfloor q/2 \rfloor$, each having roughly $m/2$ edges. Therefore, for $K > 1$, the time $T(q, m)$ for an invocation of **A2-color** with a graph of bound $q$ and $m$ edges is $T(q, m) = O(n^{.5}m) + 2T(\lfloor q/2 \rfloor, m) = KT(q/K, m) + (K-1)O(n^{.5}m)$ $\leq KT(m, m) + KO(n^{.5}m)$. Now $T(m, m) = 2T(m/2, m/2) + O(n^{.5}m)$, i.e., for some $c, m' \in N$, $T(m, m) \leq 2T(m/2, m/2) + cn^{.5}m$ for all $m > m'$. Hence $T(m, m) \leq mT(1, 1)$

in the Euler partition are assigned to $H$ and $H'$ in such a manner that the weights of $H$ and $H'$ differ by at most 1 at the end of the algorithm.

We now show that $H, H'$ are an Euler division of $G$. Clearly $E = F \cup F'$ by construction. Let $X$ and $X'$ be, respectively, the weights of $H$ and $H'$, $x$, $x'$ the weights of their critical vertices, and $r$, $r'$ their bounds. Since $q$ is even, and $W = kq$, $W$ is even. Also, $W = X + X'$ and $|X - X'| \leq 1$ by construction. Hence $X = X' = kq/2$. Consequently, if we show that $r = r' = q/2$, then $H, H'$ will be $k$-complete with respect to their bounds and thus be an Euler division of $G$.

By definition of bound, showing $r = r' = q/2$ requires that we show that $x$, $x' \leq q/2$. Let $z = wt(G, c)$ be the weight of an arbitrary vertex $c$ in $G$ at the start of the algorithm. Note that at line 1, if $z$ is even, there are an even number of odd-weight edges incident upon $c$ in $G$, and an odd number otherwise. Consequently, after line 6, $wt(G', c) = d$ is even if $z$ is even, and odd otherwise. In either case, after line 6, $wt(H, c) = wt(H', c) = (z - d)/2$ is even. Next, recall that by definition an Euler partition results in an odd-degree vertex being at the end of exactly one open walk; hence observe that **EP** followed by lines 8 - 18 results in a vertex of weight $d$ in $G'$ having weight at most $\lceil d/2 \rceil$ in $H$, $H'$. Therefore, lines 7 - 18 result in $wt(H, c), wt(H', c) \leq (z - d)/2 + \lceil d/2 \rceil = \lceil z/2 \rceil$. Since $G$ is $k$-complete with respect to $q$, $z \leq q$, and since $q$ is even, $\lceil z/2 \rceil \leq \lceil (q-1)/2 \rceil$. Hence $wt(H, c), wt(H', c) \leq q/2$. □

**Lemma 7. ED** takes time $O(n + m)$.

*proof.* The first **for** loop takes time $O(m)$. From Gabow [9], **EP** takes time $O(n + m)$. The **for** loop from lines 11 - 19 also takes time $O(m)$. □

We can now state the sequential algorithm **A2** which is based on a divide-and-conquer strategy. In the following, angle brackets delimit a sequence and parallel bars denote sequence concatenation.

**Algorithm A2.**
**Input:** Bipartite graph $G = (A, B, E)$ with weight function $wt : E \cup A \cup B \rightarrow N$, and integer $k$.
**Output:** A minimum-length $k$-coloring of $G$, as a sequence $s$ of $(M, b)$ pairs, where $M$ is a critical $k$-coloring and $b$ is its duration.

0. s := ⟨ ⟩;
1. w := max(wt(v): v ∈ A ∪ B;
2. W := sum(wt(e): e ∈ E);
3. q := max(w, ⌈ W/k ⌉);

and $H, H'$ are $k$-complete with respect to their bounds.

In the following we will show that if $q$ is even, an Euler division always exists. (This result is required for the divide-and-conquer algorithm **A2** we develop later in this section). The proof is constructive, and is based on the Euler partition algorithm of gabow [9] and the *perfect Euler split* algorithm of Somalwar [23], both of which are designed for bipartite graphs with unit-weight edges. We use the notation $wt(Y, y)$ to refer to the weight of a vertex or an edge $y$ in a weighted bipartite graph $Y$.

**Algorithm ED** /* Euler Division of bipartite graphs with even bounds */
**Input:** Bipartite graph $G = (A, B, E)$ that is $k$-complete with respect to its bound $q$, which is even.
**Output:** An Euler division of $G$ into bipartite graphs $H = (A, B, F)$, $H' = (A, B, F')$.

```
1. F, F' := {}, {};
2. for each e ∈ E such that wt(G, e) > 1 do
3.      wt(H, e), wt(H', e), wt(G, e) := ⌊ wt(G, e) / 2 ⌋, ⌊ wt(G, e) / 2 ⌋,
            wt(G, e) - 2 * ⌊ wt(G, e) / 2 ⌋;
4.      F, F' := F ∪ {e}, F' ∪ {e};
5. end for
6. G' := G;          /* G' introduced for convenience only */
7. P := EP(G');          /* EP generates an Euler partition [9]*/
8. balance := true;
9. for each walk p ∈ P do
10.     for each edge e ∈ p do
11.         if balance = true then
12.             wt(H, e), F := wt(H, e) + 1, F ∪ {e};
13.         else
14.             wt(H', e), F' := wt(H', e) + 1, F' ∪ {e};
15.         end if
16.         balance := ¬balance;
17.     end for
18. end for
```

**Lemma 6.** Algorithm **ED** generates an Euler division of $G$.

*proof.* The first **for** loop divides all even weight edges from $G$ and converts all odd weight edges to unit weight. Thus $G'$ is a unit-weight bipartite graph, and at line 6 $H$ and $H'$ have equal weights and equal degrees at every vertex. Algorithm **EP** generates an Euler partition for a unit-weight bipartite graph [9]. It is clear that lines 9 - 18 ensure that edges from walks

# 4 A divide-and-conquer scheduling algorithm

In this section we obtain an optimal algorithm for *IOS* which we call **A2**. Algorithm **A2** is applicable both to situations where all transfers are of unit length and where they are of unequal lengths; for the case of unit-length transfers, **A2** represents a substantial improvement in worst-case running time over both algorithms **KT** and **A1**. The key observation is that the bipartite graph $G$ can be partitioned into two graphs of roughly equal weight which represent two independent sub-problems of roughly half the complexity of the original problem. Then algorithm **A1** can be applied recursively to the subproblems to obtain an optimal schedule.

**Def.** A *walk* is any sequence of distinct edges. The first and last vertex of the sequence are called the *ends* of the walk. An *open* walk is one in which the ends are distinct; otherwise the walk is *closed*.

A walk differs from a path in that any vertex may be included more than once (not just the first vertex). This definition is used in the following two definitions, which are based on Cole and Hopcroft [7].

**Def.** An *Euler partition* of a graph is a partition of the edges into open and closed walks, so that each vertex of odd degree is at the end of exactly one open walk, and each vertex of even degree is at the end of no open walk.

**Def.** An *Euler split* of a bipartite graph $G = (A, B, E)$ is a pair of bipartite graphs $H = (A, B, F)$ and $H' = (A, B, F')$ where $E = F \cup F'$ and a vertex of degree $d$ in $G$ has degree $\lceil d/2 \rceil$ in one of $H$, $H'$ and $\lfloor d/2 \rfloor$ in the other.

Every graph has an Euler partition, but only bipartite graphs need have Euler splits [2, 7]. An Euler split can be formed from an Euler partition of $G$ by placing alternate edges of walks into $F$ and $F'$; both can be found in time $O(n+m)$ for bipartite graphs and multigraphs [9]. An Euler split of the multigraph $G'$ corresponding to $G$, using the algorithm of Gabow [9], suffices to divide $G$ into two subgraphs each having roughly half the weight of the original. However, this approach would take time proportional to the maximum edge weight in $G$. In the following we develop an algorithm which is faster because it avoids converting $G$ into a multigraph.

**Def.** An *Euler division* of a weighted bipartite graph $G = (A, B, E)$ which is $k$-complete with respect to its bound $q$ is a pair of bipartite graphs $H = (A, B, F)$ and $H' = (A, B, F')$ with $E = F \cup F'$ where the bounds of $H$ and $H'$ are $r = \lceil q/2 \rceil$ and $r' = \lfloor q/2 \rfloor$ respectively,

It should be pointed out that the objective of Bongiovanni et al [3] was to obtain a minimum-length schedule while not paying too high a price in terms of $L$, the number of times that critical $k$-matchings have to be calculated. The concern in this paper is to obtain a minimum-length schedule while reducing the total running time of the algorithm. In the following we evaluate the running time of **KT** and suggest an improvement.

**Theorem 3.** The running time of algorithm **KT** to find a minimum-length schedule for an instance of *IOS* is $O(n^5)$.

*proof.* We show in [23] that augmenting a bipartite graph to make it $k$-complete is $O(n)$, as is deleting the dummy traffic. Bongiovanni et al [3] show that $L = O(n^2)$, but the running time of **KT** is not calculated. However, we thus know that the number of times that the max-min bipartite matching algorithm **MaxMinMatch** [18] is called is $O(n^2)$. Since the max-min matching algorithm has a running time of $O(n^3)$, the result follows. □

## 3.2 An algorithm based on maximal matching

In this section we show how **KT** can be improved. We recall that in order to minimize the running time an algorithm that simply finds a maximum-cardinality matching in $H'$, the regular weight bipartite graph with $2n - k$ vertices in each partition, suffices. Such an algorithm will find the required critical $k$-matching in $H$. We can thus use the maximum cardinality matching algorithm of [14], which we call **MaxMatch**. We call the resulting optimal scheduling algorithm **A1**.

**Theorem 4.** The running time of algorithm **A1** to find a minimum-length schedule for an instance of *IOS* is $O(n^{4.5})$.

*proof.* The maximum cardinality matching algorithm **MaxMatch** of [14] takes time $O(|V|^{.5}|E|)$ $= O(|V|^{2.5})$ for a bipartite graph with $|V|$ vertices and $|E|$ edges. For **A1**, $|V| = 2n - k$. Using the maximum cardinality matching algorithm does not affect the worst-case value of $L$, or the time complexity of adding and deleting dummy traffic. We can use the reasoning of Theorem 3 to obtain the result. □

$k$-complete with respect to $q$, there exists a critical $k$-matching.

*proof.* The proof is by construction. Use $k$-**fill** to generate $H' = (A \cup C, B \cup D, E \cup F)$ from $H$. From Lemma 5, there exists a matching $M$ of size $2n - k$ for $H'$. Since $H'$ contains no edges in $C \times D$, $n - k$ edges of $M$ are required to cover the vertices in $C$, and similarly for $D$. This leaves $2n - k - 2(n - k) = k$ edges of $M$ having vertices only in $H$. Further, these $k$ edges cover every critical weight vertex in $H$, since such vertices do not need to be augmented by adding edges in $F$. Thus $M \cap E$ gives the required critical $k$-matching on $H$. □

We call the construction outlined in the proof of Theorem 1 the critical $k$-matching algorithm **CKM**. It is specified precisely in [3, 23]. We now cite the theorem that, together with Theorem 1, ensures that it is always possible to $k$-color $G$.

**Def.** The *duration* of a matching $M$ of a graph with bound $q$ is $r = \min(r', q - r'')$, where $r'$ is the minimum of the edge weights of $M$ and $r''$ is the maximum vertex weight among vertices not covered by $M$.

**Theorem 2.** Let $H = (A, B, E)$ be a $k$-complete bipartite graph with $|A| = |B|$ which is $k$-complete with respect to its bound $q$. Let $M$ be a critical $k$-matching on $H$. Let $M'$ be $M$ with the weight of all edges set to the duration $r$ of $M$. Then the graph $HM = (A, B, E - M')$ is $k$-complete with respect to $q' = q - r$.

*proof.* See Bongiovanni et al [3]. □

Informally, the duration of a matching is the number of time slots a critical $k$-matching can be used repeatedly. At each time slot that it is used, the weight of all the edges in the graph that are also in the matching decreases by one, since the remaining length of the transfer represented by that edge decreases by one. A new critical $k$-matching must be calculated if either the weight of one of the edges of the matching decreases to zero, or a vertex that was previously not critical weight becomes critical. Thus $L$, the number of times that critical $k$-matchings have to be calculated, satisfies $L \leq q$.

Theorems 1 and 2 together lead to an optimal algorithm which called **KT** [3]. In **KT**, dummy traffic is added to augment the bipartite graph and make it $k$-complete with respect to its bound. The well-known max-min bipartite weighted matching algorithm (see [18]), which we call **MaxMinMatch**, is then invoked to find a critical $k$-matching. The result of Theorem 4 is used to calculate its duration. A sequence of critical $k$-matchings is found by calling the max-min bipartite algorithm repeatedly. Finally, the dummy traffic is removed from the $k$-matchings to obtain an optimal schedule.

## 3.1 An algorithm based on max-min matching

We develop the **KT** algorithm in a graph-theoretic framework. The key observation in Bongiovanni et al [3] is that a bipartite graph can always be augmented to have a certain characteristic, called *k-completeness*, such that a critical $k$-matching exists. Further, deleting a critical $k$-matching from the graph leaves it $k$-complete. Thus a sequence of critical $k$-matchings can be found, and hence a schedule. The following two lemmas are used in the proof of Theorem 1, which implies that a critical $k$-matching exists in $G$.

**Def.** A graph of weight $W$ and maximum vertex weight $w$ is *k-complete with respect to a constant* $r$ if $w \leq r$ and $W = kr$.

**Lemma 4.** Any bipartite graph $G$ with bound $q$ can be augmented by adding appropriate weighted edges so as to obtain a bipartite graph $H$ that is $k$-complete with respect to $q$.

*proof.* The proof is constructive, via an algorithm that takes time $O(n)$. See Somalwar [23] for details. The added weighted edges are called *dummy traffic* and are deleted at the end of the scheduling algorithm. □

**Def.** A bipartite graph is *regular* if all its vertices have the same degree; it is *regular weight* if all vertices have the same weight.

**Lemma 5.** For any regular weight bipartite graph $G = (A, B, E)$ with $|A| = |B|$, there exists a matching of size $|A|$, which is therefore maximal.

*proof.* See Berge [2]. □

**Def.** A bipartite graph of weight $W$ and maximum vertex weight $w$ is *k-filled with respect to a constant* $r$ if it is $k$-complete with respect to $r$, and all vertices have weight $r$.

A bipartite graph $H = (A, B, E)$ with $|A| = |B|$ that is $k$-complete with respect to $r$ can be transformed to one that is $k$-filled with respect to $r$ by a construction sketched as follows. Let $|A| = |B| = n$. Augment $H$ so that $H' = (A \cup C, B \cup D, E \cup F)$ with vertex sets $|C| = |D| = n - k$ and edge set $F = AD \cup BC$ where $AD \subseteq A \times D$ and $BC \subseteq B \times C$. The edges in $F$ are added and their weights assigned such that all vertices in $H'$ have weight $r$ and yet $H'$ is also $k$-complete with respect to $r$. We call this algorithm **$k$-fill**. See Bongiovanni et al [3] for a detailed description of this algorithm and its proof of correctness. We observe that **$k$-fill** takes time $O(n)$.

**Theorem 1.** For any bipartite graph $H = (A, B, E)$ with $|A| = |B|$ and bound $q$ that is

*proof.* Clearly, at least $d$ colors are necessary, since a critical vertex requires that each incident edge have a different color. The proof of sufficiency is by induction, sketched as follows [2]. Find a critical matching $M$, which, from Lemma 1, must exist. Color the edges in $M$ a single color and delete them from the graph. The remaining graph has degree $d-1$, and by the induction hypothesis can be colored using $d-1$ colors. Hence the graph can be colored with $d$ colors. □

**Def.** A *k-coloring* of a graph is an edge-coloring in which each color may be used to color at most $k$ edges.

**Lemma 3.** At least $p = \max(d, \lceil m/k \rceil)$ colors are necessary to $k$-color a bipartite graph with $m$ edges, degree $d$, and unit-weight edges.

*proof.* If $d < \lceil m/k \rceil$, at least $\lceil m/k \rceil$ colors are required to color the graph. Otherwise the argument of Lemma 2 applies. □

**Notation.** Let $w$ denote the maximum weight of any vertex in the bipartite graph $G$, and $W$ denote the weight of $G$.

**Def.** The *bound* of the bipartite graph $G$ for an instance of $IOS$ is defined as $q = \max(w, \lceil W/k \rceil)$.

Any scheduling algorithm which produces a schedule of length equal to the *bound* for every instance of $IOS$, is optimal. Notice that a scheduling algorithm has two measures of performance: optimality, i.e., how close the length of the schedule it produces is to the minimum length, and how long the algorithm takes to run.

**Def.** For a bipartite graph $G$ representing the transfers in $IOS$, a *critical weight vertex* is one of weight equal to the bound $q$. A *critical weight matching* is one which includes all critical weight vertices. A *critical k-matching* is a critical weight matching of cardinality $k$.

# 3 An improved scheduling algorithm

In this section we present the outline of the algorithm **KT** [3], followed by our first direct improvement to its time complexity. The algorithm is presented in suficient detail so as to justify the improvement and to provide a basis for the divide-and-conquer algorithm to be presented in sec. 4.

of vertices and $E \subseteq A \times B$ is the set of edges. Let $|A| + |B| = n$ and $|E| = m$. Each edge $e \in E$ has a *weight* $wt(e)$ associated with it, and the weight of a vertex is the sum of the weights of the edges incident upon it. Thus $wt : E \cup A \cup B \rightarrow N$. We can also represent the weighted bipartite graph $G$ as a multigraph $G' = (A, B, E')$ where each edge $e \in E$ is replaced by $wt(e)$ parallel edges of unit weight in $E'$. Then $G$ is referred to as the *underlying graph* of $G'$, and $G'$ as the multigraph corresponding to $G$.

Consider an instance of *IOS* where the architecture allows at most $k$ simultaneous transfers, and I/O transfers may be of arbitrary positive integer length. Then the problem can be represented as a weighted bipartite graph $G$, where edge weights represent transfer lengths. Since preemption is allowed, a schedule can be obtained as an edge-coloring of the multigraph corresponding to $G$, with the restriction that no color may be used more than $k$ times.

We now state the definitions and lemmas used to derive results on edge-coloring. The following two lemmas are well-known results from graph theory. The rest are graph-theoretic versions of those in Bongiovanni et al [3].

**Def.** The *degree* of a vertex is the number of edges incident upon it. The degree of a graph is the maximum of the degrees of its vertices. A *critical vertex* is one of maximum degree.

**Def.** The *weight* of a graph is the sum of the weights of its edges. The *weight* of a vertex is the sum of the weights of the edges incident upon it.

Notice that for a graph with unit-weight edges the degree of a vertex equals its weight.

**Def.** A *matching* $M \subseteq E$ is a set of edges such that no two edges have a common vertex. A *maximal matching* is one such that no other matching has a larger cardinality. An edge in a matching is said to *cover* the vertices that are its endpoints.

**Def.** A *critical matching* is one which covers all critical vertices.

**Lemma 1**. Every bipartite graph has a critical matching.

*proof.* See Berge [2]. □

Although the following Lemma is well known, we sketch the proof here as it provides some intuition for results later in the paper.

**Lemma 2.** Exactly $d$ colors are necessary and sufficient to color a bipartite graph of degree $d$.

Figure 1: Parallel I/O scheduling example

**Def.** An *edge coloring* of a graph $G = (V, E)$ is a function $c : E \rightarrow N$ which associates a color with each edge such that no two edges of the same color have a common vertex.

Consider a collection of vertices representing processors and I/O devices, each of which can participate in at most one data transfer at any given time. Then an edge coloring for a graph $G$, where each edge of $G$ represents a data transfer requiring one time unit, corresponds to a schedule for the data transfers, and vice versa. To see this, note that all edges of $G$ colored with the same color are independent in that they have no common vertex. Hence the data transfers they represent can be performed simultaneously. An edge coloring of $G$ represents a schedule, where all edges $e$ with $c(e) = i$, for some $i$, represent data transfers that take place at time $i$, and vice versa. The number of colors required to edge-color $G$ equals the length of the schedule, and vice versa.

**Example.** An instance of the simple I/O scheduling problem is given in Fig. 1. Assume that each processor and each I/O device can participate in at most data transfer at any given time, and each transfer is of unit length. Clearly the minimum number of colors required to edge-color the graph, and hence the minimum length of time for completing the transfers, corresponds to the optimal schedule rather than the schedule obtained by coloring the edges, or executing the tasks, in the order they are numbered.

**Notation.** Let $G = (A, B, E)$ denote a bipartite graph where $A$ and $B$ are two disjoint sets

than one data transfer to be in progress at any given time, but also allow more processors and devices to be interconnected and improve the system's fault tolerance [21].

We have previously shown, using our general scheduling model, the similarity between $IOS$ and the problem of obtaining optimal time-slot assignments in a satellite switching system [16]. We are thus able to utilize the algorithm **KT** for the satellite switching problem, due to Bongiovanni et al [3], as a starting point for developing an optimal algorithm for $IOS$. By a series of improvements we obtain a faster algorithm for solving $IOS$, improving the time complexity from $O(n^5)$ for Bongiovanni et al's **KT** algorithm to $O(n^3(\log n + \log K))$, where $n$ is the number of processor and I/O devices, and $K$ is the length of the longest transfer. Our algorithm is based on optimal $k$-colorings of bipartite graphs.

In section 2 we introduce basic definitions and results from graph theory used throughout the paper. In section 3 we present Bongiovanni et al's algorithm in a graph-theoretic form so that it can be applied readily to the $IOS$ problem. We then show how this algorithm can be improved, in three ways. The first improvement arises from the observation that it is not necessary to obtain a min-max bipartite graph matching, as the authors suggest, but that a maximum cardinality matching suffices. The first improvement is discussed in section 3. The second improvement arises from the observation that a divide-and-conquer strategy can be used to reduce the worst-case time complexity, and is presented in sec. 4. The third improvement arises from observing that the graph of interest can be embedded inside a larger graph, allowing the weighted bipartite matching algorithm of Gabow and Kariv [10] to be applied, hence reducing the time complexity further. This improvement is discussed in sec. 5. In sec. 6 we discuss an experimental study of the efficiency of the scheduling algorithms described in this paper. Finally, in sec. 7 we discuss previous work, and we end with some conclusions.

# 2 Definitions and problem formulation

The **KT** algorithm [3] was stated in the context of a specific application and developed using a matrix formulation. In this section we introduce the definitions and ideas to be used in sec. 3 to give a graph-theoretic presentation of the **KT** algorithm, thus allowing it to be applied directly to $IOS$. The key observation is that computing a schedule corresponds to edge-coloring a bipartite graph where the two vertex partitions represent processors and I/O devices, and edges represent data transfers between them. We first introduce some definitions and notation.

## 1.2 Research approach

This research program is both theoretical and experimental. Earlier work by Somalwar [23] on parallel I/O scheduling developed and evaluated heuristics for scheduling of simultaneous requests for multiple resources, such as I/O requests, while Kandappan [17] and Balan [1] studied the impact of data allocation to disks. This paper defines a specific parallel I/O scheduling (or simultaneous multiple resource) scheduling problem conforming to the requirments of the two families of applications discussed above and presents an efficient algorithm for this problem. Experimental evaluation of the algorithm and comparison with the best known heuristics, both by simulation and execution on a Sequent Symmetry, are currently in progress.

The specific I/O problem to which the algorithms in this paper apply can be stated, as a decision problem, as follows. Since, in principle, the problem may arise between any two levels of the memory hierarchy, it is stated in terms of processors and I/O devices. Given a set of I/O transfers, where

1. each transfer requires a fixed but possibly distinct time,
2. each transfer requires a specified pair of resources, one a processor an the other an I/O device, from two given sets of processors and I/O devices,
3. each processor can communicate via a direct dedicated link with each I/O device, and
4. the transfers may occur in any order,

is there a preemptive schedule for performing the transfers whose total length is at most some given bound?

We call this problem the Simple I/O Scheduling (*SimpleIOS*) problem. (When the problem is stated as an optimization problem the objective is to minimize the schedule length). It is applicable to systems such as the Sequent [20] where I/O devices are connected to processors via a single shared bus. In this paper we also consider an extension to *SimpleIOS* that is useful for modeling some practical parallel I/O architectures. We call this problem *IOS*, and it is identical to *SimpleIOS* except that the system architecture differs:

**Extension E1**. Only a fixed limited number $k$ of I/O operations may take place at any given time.

Extension **E1** arises in multiple-bus systems such as the IBM RP3, where $k$ parallel buses connect processor and I/O devices [22]. In general, such multiple parallel bus architectures are attractive for future high-performance parallel computers as they not only allow more

## 1.1 The structure of I/O requests

It is worthwhile, before proposing algorithms applicable to scheduling of parallel I/O operations on parallel architectures, to consider the likely structure of I/O requests in parallel computing systems. The conditions for scheduling to be effective are that there are choices to be made which affect performance, and that there are resource bottlenecks whose utilization can be improved by scheduling. The nature of the stream of I/O requests and the resource configuration determine whether these conditions hold, and if so, the characteristics of scheduling algorithms that will be effective.

The stream of I/O requests generated by multiprogramming workloads is largely uncorrelated and has traditionally been scheduled dynamically at the level of device controllers or channels. There is usually sufficient randomness among requests to avoid long queues at any given disk so that scheduling parallel I/O operations above the controller level is of little benefit. However for certain applications and parallel architectures there are correlations between the I/O requests leading to substantial resource conflicts. The amount of correlation among the requests varies substantially. Certain applications, such as 3D migration codes in seismic processing where the solution progresses systematically across a coordinate space, yield highly structured and totally predictable patterns of requests for data. In this case scheduling of I/O requests is of little benefit since the order of the requests can be predicted in advance and thus the problem reduces to one of assigning data to storage devices so as to minimize conflicts [5, 17]. On the other hand, certain families of applications, such as 3D visualization and decision support systems, pass through phases with different degrees of parallelism in computation and I/O requests. These applications, whether executing in parallel or sequentially, generate I/O requests in bursts as the locality of the data to be displayed or analyzed changes. It is often the case that the entire set of requests must be satisfied before the computation can proceed. These families of applications may benefit substantially from batch-oriented scheduling of parallel I/O operations if resource bottlenecks exist in the I/O system architecture.

As far as resource bottlenecks go, it is typically the case in large-scale parallel architectures that there are fewer access paths to I/O devices than either I/O devices or processors. This is commonly observed in current bus-oriented shared memory architectures (e.g. [20]).

The combination of bursts of I/O requests and potential resource bottlenecks suggest that there may be utility in an efficient algorithm for generating parallel schedules. An efficient algorithm is needed because it will be executed repeatedly during the execution of the applications. This paper develops and characterizes algorithms which are applicable to a significant class of I/O system architectures.

# 1 Introduction

It has long been recognized that the mechanical delays associated with input/output (I/O) devices represent a significant potential bottleneck in computer system performance. Indeed, the introduction of multiprogramming in computer systems was motivated by the need to overcome the I/O bottleneck [6]. Nonetheless, "Input/Output has been the orphan of computer architecture" [13], and the I/O subsytem has received disproportionately little attention in sequential computer system design.

The I/O subsystem has received even less attention in the design of parallel computer systems. Furthermore, while the scheduling of I/O operations has been studied in the context of sequential computer organizations [8], the potential for improving parallel system performance by exploiting and scheduling parallel I/O operations has been almost completely neglected [16]. (See the work of Ramamritham and Stankovic, e.g. [25], for a discussion of suboptimal heuristics for real-time multiple resource scheduling). However it seems clear that I/O represents as much, if not more, of a potential performance bottleneck for parallel systems as it did for sequential computer systems. The increasing discrepancy between the speeds of computation and I/O, and the ever-increasing quantities of data handled by application programs, indicate that I/O will become an increasingly important bottleneck in the future [4, 5, 12].

Explicit scheduling of parallel I/O operations thus emerges as a potentially significant means of improving the performance of multiprocessor computer systems. Review of previous work reveals that almost all previous research on parallel scheduling deals with tasks which require only a single resource at any given time [16]. This is not relevant for parallel scheduling of I/O operations where each operation requires multiple resources (processor or memory, transfer media and external memory) in order to execute. Serial acquistion of multiple resources does not in general lead to optimal schedules; algorithms which simultaneously assign multiple resources to the members of a request set are required. The research reported here has been carried out in the context of a general model for scheduling problems which covers simultaneous scheduling of multiple resources. This model has been reported elsewhere [16], so this paper focuses on algorithms appropriate for parallel scheduling of batched I/O operations on shared memory multiprocessors.

# Contents

**1 Introduction** **5**

1.1 The structure of I/O requests . . . . . 6

1.2 Research approach . . . . . 7

**2 Definitions and problem formulation** **8**

**3 An improved scheduling algorithm** **11**

3.1 An algorithm based on max-min matching . . . . . 12

3.2 An algorithm based on maximal matching . . . . . 14

**4 A divide-and-conquer scheduling algorithm** **15**

**5 An algorithm for large transfer lengths** **19**

**6 Experimental evaluation** **20**

**7 Previous work** **22**

**8 Conclusions** **24**

**References** **24**

**Typographical List of Symbols Used**

$O$ - capital italic letter "oh"
$o$ - small italic letter "oh"
$n$ - small italic letter "enn"
$N$ - capital italic letter "enn"
log - lower case Roman word "log"
max - lower case Roman word "max"
min - lower case Roman word "min"
$K$ - capital italic letter "kay"
$k$ - small italic letter "kay"
$\rightarrow$ - right arrow
$\subseteq$ - right horseshow with underscore
$|$ - vertical bar
$\times$ - cross, or "times"
$\in$ - differs slightly from Greek epsilon
$\cup$ - large upward cup
$\cap$ - large downward cup
$\lceil\ \rceil$ - "ceiling"
$\lfloor\ \rfloor$ - "floor"
$'$ - italic "apostrophe", or "prime"
' - Roman "apostrophe", or "prime"

Running Head: PARALLEL I/O SCHEDULING

Send proofs to: Prof. J. C. Browne
Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712
Phone: (512)-471-9579. E-mail: browne@cs.utexas.edu

**Abstract**

The I/O subsystem was one of the first areas of computer system design to incorporate parallelism. I/O processing remains a common bottleneck for many applications even on sequential processors. Although multiprocessor architectures increase the rate of I/O requests, enhancement of the parallelism of I/O systems has received little attention in parallel system design. There has been almost no study of the benefit of scheduling parallel I/O operations to increase the multiprocessor system performance.

An I/O transfer typically requires a processor or memory, a channel and an external memory device simultaneously. Parallel I/O thus requires scheduling multiple resources simultaneously, rather than a single resource serially. This paper presents an algorithm for optimal scheduling of batched parallel I/O requests for a common class of shared memory multiprocessors. The algorithm is essentially an optimal $k$-coloring of a bipartite graph with arbitrary edge weights, where the vertices represent processors and memories and the edges represent I/O transfers. The best previously known $k$-coloring algorithm has running time $O(n^5)$. We show a series of improvements to obtain an algorithm with a running time $O(n^3(\log n + \log K))$, where $n$ is the number of vertices and $K$ is the maximum edge weight, i.e., the length of the longest I/O transfer. We conclude with an experimental study of the performance of the algorithms.

# Scheduling Parallel I/O Operations in Multiple Bus Systems*

Ravi Jain, Kiran Somalwar†, John Werth and J. C. Browne‡
Department of Computer Sciences
University of Texas at Austin, Austin, TX 78712

Submitted December 4, 1991
**Revision Submitted May 26, 1992**

*This research was supported by the IBM Corporation through grant 61653 and by the State of Texas through TATP Project 003658-237.

†Current address: Digital Equipment Corp., 550 King St., Littleton, MA 01460.

‡Address correspondence to J. C. Browne: (512)-471-9579 or `browne@cs.utexas.edu`.