

The Dance Party Problem and its Application to Collective Communication In Computer Networks

Xin Wang^{†*} Edward K. Blum^{‡*} D. Stott Parker[†] Daniel Massey[†]

[†] Computer Science Department, UCLA, Los Angeles, CA 90024

[‡] Mathematics Department, USC, Los Angeles, CA 90089

* Compbionics Inc., Los Angeles, CA 90049

Abstract

Motivated by implementing collective communication operations on a network of processors connected via ethernet and similar bus lines, a problem of scheduling a dance party is formulated. The problem is solved by two algorithms based on searching and divide-and-conquer that generate suboptimal schedules and an algorithm based on graph factorization that generates optimal schedules. It is shown how to use dance schedules to implement collective communication operations such as `all-gather`.

*email address: xwang@cs.ucla.edu, fax: (213) 749-8489.

1 The Problem

There are a number of people at a dance party. Each one wants to dance with everyone else once and only once. (Gender is ignored.) How can one schedule dance partners at each round so that everyone either knows he or she should be idle, or is able to find a right partner? What schedules are optimal in the sense that the party can be over as early as possible?

Let n be the number of people at the party and label the people 1 through n . Examples of optimal schedules for $n = 2, 3, 4$, respectively, are

	1
1	2
2	1

	1	2	3
1	2	3	1
2	1	2	3
3	3	1	2

	1	2	3
1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

Here, an entry at row i and column r in a schedule denotes the partner of person i at round r . If the partner is p him- or herself (e.g., entries at (1, 3), (2, 2) and (3, 1) of the schedule for $n = 3$), then it means that i is idle at that round. These schedules are optimal because no other schedules can use fewer rounds.

Besides its natural applications to scheduling dance parties and similar activities such as round-robin tournaments, the dance party problem has an interesting application in using a network of n processors connected via ethernet as a “virtual” MIMD machine. In a distributed parallel programming environment (such as PVM [?], and p4 [?]), efficient implementation of global and collective communication operations according to commonly used communication patterns is important. A typical example of a global and collective communication operation is `all-gather` [?] or equivalently `all-to-all personalized communication` [?], which sends data from all

processors to all processors in the network. This operation is fundamental for implementing other collective operations such as calculating global maximum, minimum, summation and product. Semantically, the **all-gather** operation is equivalent to a double loop in which each processor takes its turn to send individual messages to all other processors. If the order of sending and receiving messages is not properly scheduled, **all-gather** may result in deadlocks in which two or more processors either all try to send messages concurrently to the others or all expect to receive messages concurrently from the others. On the other hand, if the operation is implemented as the double loop mentioned above, little parallelism is achieved in the communication and the time needed to perform such an operation is the time of $n(n - 1)$ primitive **send/receive** operations.

If the n processors are identified as the people at the dance party, then whom and when a processor should send messages to and receive messages from correspond to which partner and at which round a person should dance with. The reason for requiring two people to dance exactly once is that, when a communication connection is established between two processors, the processors can exchange messages by first letting one send and the other receive and then switching their send and receive roles; who sends first can be decided by comparing unique processor IDs. A schedule of the dance party problem provides a dead-lock free communication algorithm and an optimal schedule leads to an efficient algorithm for implementing the **all-gather** operation and related collective communication operations.

Optimal algorithms for implementing the **all-gather** operation have been developed for particular network architectures such as ring, mesh and hypercube. The time complexities of some known efficient algorithms for n processors [?] are $O(n^2)$ for ring, $O(n\sqrt{n})$ for 2-D mesh and $O(n \log_2 n)$ for hypercube. But, unlike these particular network architectures, ethernet makes it possible to send messages between any two processors on the network without going through other processors. Even though the ethernet

allows only one message on a network bus segment at a time, exploiting parallelism at the algorithm level and letting ethernet hardware perform the bus arbitration to resolve bus contention very likely achieves more overall parallelism than enforcing sequential communication at the algorithm level. The same is true of FDDI token rings. This is particularly important in heterogeneous networks where processors may run at different speeds and execute different code segments. By casting it into the dance party problem, the problem of implementing the collective operation **all-gather** by the point-to-point communication primitives **send** and **receive** has solutions of time complexity $O(n)$.

In the next section, the problem is formalized and some properties of optimal solutions are discussed. In Sections 3 and 4, two suboptimal schedules are constructed based on searching and divide-and-conquer techniques. In Section 5, an optimal schedule is obtained by relating the problem to a graph-theoretical problem, namely, the one-factorization problem. It is shown in Section 6 how to apply these schedules to implementing the **all-gather** operation.

2 Formalization of the Problem

Let n and t be two positive integers. An $n \times t$ array P with entries in the set $\{1, \dots, n\}$ is called a *schedule* of the dance party problem for n people in t total rounds if it satisfies the following constraints:

- (i) for any $i \neq j$, $1 \leq i, j \leq n$, there is a unique r , $1 \leq r \leq t$, such that $P(i, r) = j$;
- (ii) for any r , $1 \leq r \leq t$, and any j , $1 \leq j \leq n$, there is a unique i , $1 \leq i \leq n$, such that $P(i, r) = j$; and
- (iii) for any i, j , $1 \leq i, j \leq n$, and any r , $1 \leq r \leq t$, $P(i, r) = j$ if and only if $P(j, r) = i$.

In words, if $j = P(i, r)$, then j is scheduled to be the partner of i in the r -th round; if $j = i$, it means that i is idle in that round. The constraint (i) means that any two people dance with each other once and only once; (ii) restricts everybody to dance only with one person at each round; and (iii) states that, in each round, j is the partner of i if and only if i is the partner of j . As everybody dances with all $n - 1$ others once and only once and each dance involves exactly two persons, there are $n(n - 1)/2$ dances altogether which means that in any schedule P the number of entries such that $P(i, r) \neq i$ is equal to $n(n - 1)$.

Let $n(P)$ and $t(P)$ be the number of rows (people) and the number of columns (rounds) of a schedule P , respectively. Formally, the dance party problem is to construct, for any given $n > 1$, a schedule P with $n(P) = n$ such that $t(P)$ is as small as possible.

The simplest is a “sequential” schedule S_n of n people, in which the first person dances first with the other $n - 1$ persons one by one, then the second person dances with the remaining $n - 2$ persons one by one, and so on. For example, S_4 is

	1	2	3	4	5	6
1	2	3	4	1	1	1
2	1	2	2	3	4	2
3	3	1	3	2	3	4
4	4	4	1	4	2	3

Evidently, in every round only a single pair of people dance and all others are idle. Hence, the total number of rounds needed is $t(S_n) = n(n - 1)/2$, which is far from optimal.

A schedule P is called *optimal* if, for any schedule Q with $n(Q) = n(P)$, $t(Q) \geq t(P)$ holds. Clearly, optimal schedules are not unique; for example, relabeling people and/or exchanging rounds (i.e., permuting columns) of an optimal schedule may result in other optimal schedules.

Let $T(n)$ denote the number of total dance rounds of an optimal schedule of the party of n people. Then,

Proposition 1 *For any $n > 0$,*

$$T(n) \geq \begin{cases} n - 1 & \text{if } n \text{ even} \\ n & \text{if } n \text{ odd} \end{cases}$$

and

$$T(n) \leq T(n + 1).$$

Proof. The first inequality holds because, in any schedule (matrix), every one needs to dance with all other $n - 1$ people in different rounds (columns) and at least $n - 1$ rounds (columns) are needed. If n is odd, each round (column) has to have at least one person idle. Therefore, altogether at least n rounds (columns) are needed.

The second inequality holds since a schedule P for n people can be constructed from a schedule P' for $n + 1$ people by considering the $(n + 1)$ -st person as a “dummy”, namely, for any $1 \leq i \leq n$ and $1 \leq r \leq t(P')$, $P(i, r) = i$ if $P'(i, r) = n + 1$ and $P(i, r) = P'(i, r)$ otherwise. \square

3 A Suboptimal Searching Algorithm

A schedule can be constructed by searching for free partners. At any round, people with small labels will have a high priority of getting partners. For each i , $1 \leq i \leq n$, the first person j (i.e. the smallest j) who has not danced with i in previous rounds and not been scheduled to any $k < i$ thus far in the round will be the partner of i ; if no such j can be found, then i will be idle in that round. The searching procedure stops in a round at which everyone is scheduled to be idle.

Let Q_n be the schedule produced by the searching algorithm. For example, Q_8 and Q_6 are

	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	1	4	3	6	5	8	7
3	4	1	2	7	8	5	6
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	1	4	3
7	8	5	6	3	4	1	2
8	7	6	5	4	3	2	1

	1	2	3	4	5	6	7
1	2	3	4	5	6	1	1
2	1	4	3	6	5	2	2
3	4	1	2	3	3	5	6
4	3	2	1	4	4	6	5
5	6	5	5	1	2	3	4
6	5	6	6	2	1	4	3

Notice that Q_8 is optimal as $t(Q_8) = 7$, and that Q_6 is the reduction of Q_8 by eliminating the rows 7 and 8 and replacing each entry $Q_8(i, r)$ by i if it is either 7 and 8. These facts are indeed true in general.

Proposition 2 For $n = 2^k$ with $k > 0$,

$$t(Q_n) = n - 1$$

and the schedule Q_n is optimal. For general $n > 1$,

$$t(Q_n) = 2^{\lceil \log_2 n \rceil} - 1,$$

where $k = \lceil \log_2 n \rceil$ is such that $2^{k-1} < n \leq 2^k$.

Proof. First consider the case $n = 2^k$ for some $k \geq 1$. In rounds 1 to $2^l - 1$, $l = 1, \dots, \lceil \log_2 n \rceil$, people are divided into groups of size 2^l , namely, $\{1, \dots, 2^l\}$, $\{2^l + 1, \dots, 2^{l+1}\}$, ..., $\{2^{(m-1)k} + 1, \dots, 2^{mk}\}$, where $m = n/2^l$, and each has been scheduled to dance with all the people within the same group. For example, at the first round (where $l = 1$), 1 dances with 2, 3 with 4 and so on. In the next 2^l rounds, each person in the first group will be scheduled to dance with each person in the second group, the third group with the fourth group, and so on. By induction, when $l = \lceil \log_2 n \rceil$, a schedule for everybody has been obtained. Therefore, $t(Q_{2^k}) = 2^k - 1$ and the schedule for $n = 2^k$ is optimal.

For general $n > 1$, the schedule Q_n can be constructed from $Q_{\bar{n}}$, where $\bar{n} = 2^{\lceil \log_2 n \rceil}$, by taking Q_n as the first n rows of $Q_{\bar{n}}$ and resetting entries $Q_{\bar{n}}(i, r)$ to i if they are greater than n . This is equivalent to considering that there are $2^{\lceil \log_2 n \rceil} - n$ dummy persons who join the party; any person i with $i \leq n$ will be idle if he or she is scheduled to dance with a dummy person. Therefore, $t(Q_n) = 2^{\lceil \log_2 n \rceil} - 1$. \square

Consequently,

$$t(Q_{2^{k-1}+1}) = t(Q_{2^{k-1}+2}) = \dots = t(Q_{2^k}) = 2^k - 1.$$

Hence, $t(Q_n)$ is bounded by $2^{\lceil \log_2 n \rceil} - 1 \leq 2n - 1$, which is linear in n .

4 A Suboptimal Divide-and-Conquer Algorithm

A schedule can also be constructed recursively by divide-and-conquer. The idea is to divide the group *evenly* into two groups such that the difference between numbers of people in the two groups is at most one. Then first schedule inner-group dances. This is done by using the same algorithm recursively to obtain two subschedules for the two groups. As the two groups do not intersect, the two subschedules can be run in parallel. Next schedule inter-group dances. This is done by pairing up the people across the two groups in $\lceil n/2 \rceil$ rounds, where $\lceil n/2 \rceil = n/2$ if n is even and $= (n+1)/2$ if n is odd. In case that n is odd, each round of these inter-group dances has one person idle. This algorithm reaches a base case when $n = 1$ or 2 , where, if $n = 1$, the only person in the group is scheduled to be idle and, if $n = 2$, the two persons are scheduled to dance together.

Let D_n be the schedule generated by this divide-and-conquer algorithm. For any $n = 2^k$, it is not hard to see that the algorithm always generates an optimal schedule with $t(D_n) = n - 1$, which is identical to the schedule Q_n generated by the searching algorithm. However, the schedules for other n 's are different. For example, D_6 is

	1	2	3	4	5	6
1	2	3	1	4	5	6
2	1	2	3	5	6	4
3	3	1	2	6	4	5
4	5	6	4	1	3	2
5	4	5	5	2	1	3
6	6	4	6	3	2	1

Notice that $t(D_6) = 6$, while $t(Q_6) = 7$.

Formally, we define the number of rounds required by the algorithm as a function of the number of people at the dance party.

Definition 1 Define the function t by $t(D_0) = t(D_1) = 0$, $t(D_2) = 1$, and

$$t(D_{2n-1}) = t(D_{2n}) = t(D_n) + n \quad (n \geq 2).$$

The following proposition can easily be proved by induction.

Proposition 3

$$t(D_{2^k}) = 2^k - 1, \quad t(D_{2^{k+2}}) = 2^k + k \quad \text{and} \quad t(D_{2^{k-2}}) = 2^k - 2.$$

The function $t(D_n)$ is very interesting and is closely related to the number of factors of 2 in n . Though it is out of the scope of this paper to fully study the sequence, we can use this relation to obtain a formula for $t(D_n)$ and hence a bound on the number rounds required for the dance party. To do this, we define $\beta(n)$ as the number of factors of 2 in n . It is clear that $\beta(n) = 0$ for all odd n . Here are some values of $t(D_n)$ and $\beta(n)$ for even n :

n	6	8	10	12	14	16	18	20	22	24	26	28	30	32
$t(D_n)$	6	7	11	12	14	15	20	21	23	24	27	28	30	31
$\beta(n)$	1	3	1	2	1	4	1	2	1	3	1	2	1	5

It is interesting to compare $t(D_n)$, $\beta(n)$, and $t(D_{n+1})$. The following proposition states this relationship formally.

Proposition 4 For any $n > 1$,

$$t(D_{n+1}) - t(D_n) = \beta(n) + d(n)$$

where $d(n) = 1$ if $n = 2^k$ for some $k \geq 0$, and 0 otherwise.

Proof. For odd n , $t(D_{n+1}) = t(D_n)$ and $\beta(n) = d(n) = 0$ by definition, so the result follows. For even n , we can prove the result by induction on n . The base case for $n = 0$ and $n = 2$ is easily verified. For even $n > 2$ it follows directly from the definition that

$$\begin{aligned} t(D_{n+1}) - t(D_n) &= \left(\frac{n+2}{2} + t(D_{\frac{n+2}{2}})\right) - \left(\frac{n}{2} + t(D_{\frac{n}{2}})\right) \\ &= 1 + t(D_{\frac{n}{2}+1}) - t(D_{\frac{n}{2}}) \\ &= 1 + \beta\left(\frac{n}{2}\right) + d\left(\frac{n}{2}\right) \quad (\text{by induction}) \\ &= \beta(n) + d(n) \end{aligned}$$

since $1 + \beta(n/2) = \beta(n)$ and $d(n/2) = d(n)$ for even $n > 2$. □

Corollary 1

$$t(D_n) = \sum_{i=1}^{n-1} (\beta(i) + d(i)) = \left(\sum_{i=1}^{n-1} \beta(i)\right) + \lfloor \log_2(n-1) \rfloor + 1.$$

This relationship between $t(D_n)$ and $\beta(n)$ provides insight into the structure of both sequences. We will conclude this section by using this relationship to show that schedule produced by $t(D_n)$ is very close to optimal and give a non-recursive formula for $t(D_n)$. Recall that we have already shown that the schedule is optimal when $n = 2^k$.

To establish a bound on $t(D_n)$, we develop three simple lemmas.

Lemma 1

$$\sum_{i=1}^{2^k-1} \beta(i) = 2^k - k - 1.$$

Proof.

Recall that $t(D_{2^k}) = 2^k - 1$. By the previous corollary, we also have

$$\begin{aligned} \sum_{i=1}^{2^k-1} \beta(i) &= t(D_{2^k}) - (\lfloor \log_2(2^k - 1) \rfloor + 1) \\ &= 2^k - 1 - k. \end{aligned}$$

□

Lemma 2 For $1 \leq n < 2^k$, $\beta(2^k + n) = \beta(n)$.

Proof.

If n is odd, $\beta(2^k + n) = \beta(n) = 0$.

If n is even we can prove the result by induction on n . The base case of $n = 2$ is easily verified. For $n \geq 2$, $2^{k-1} > \frac{n}{2} \geq 1$, and hence by induction we have that

$$\beta(2^k + n) = 1 + \beta(2^{k-1} + \frac{n}{2}) = 1 + \beta(\frac{n}{2}) = \beta(n).$$

□

Lemma 3

$$\sum_{i=1}^{n-1} \beta(i) = n - \beta(n) - b(n)$$

where $b(n)$ is the number of 1's in the binary representation of n .

Proof.

By induction on n . The base case of $n = 1$ is easily verified.

Let $k = \lfloor \log_2(n) \rfloor$, and put $m = n - 2^k$. If $m = 0$, then $b(n) = 1$ and by Lemma ?? we have that

$$\begin{aligned} \sum_{i=1}^{n-1} \beta(i) &= \sum_{i=1}^{2^k-1} \beta(i) \\ &= 2^k - k - 1 \\ &= n - \beta(n) - b(n). \end{aligned}$$

If $m \geq 1$, note that by construction we have $2^k > m \geq 1$ and hence by Lemmas ?? and ?? we have by induction that

$$\begin{aligned}
\sum_{i=1}^{n-1} \beta(i) &= \sum_{i=1}^{2^k+m-1} \beta(i) \\
&= \sum_{i=1}^{2^k-1} \beta(i) + \sum_{i=2^k}^{2^k+m-1} \beta(i) \\
&= 2^k - k - 1 + \beta(2^k) + \sum_{i=1}^{m-1} \beta(i) \\
&= 2^k - k - 1 + k + m - \beta(m) - b(m) \\
&= 2^k + m - \beta(2^k + m) - (b(m) + 1) \\
&= n - \beta(n) - b(n).
\end{aligned}$$

□

Theorem 1

$$t(D_n) = n + \lfloor \log_2(n-1) \rfloor + 1 - \beta(n) - b(n).$$

The proof follows immediately from the corollary and Lemma ??:

$$\begin{aligned}
t(D_n) &= \sum_{i=1}^n \beta(i) + \lfloor \log_2(n-1) \rfloor + 1 \\
&= n - \beta(n) - b(n) + \lfloor \log_2(n-1) \rfloor + 1.
\end{aligned}$$

□ As a result we can compute $t(D_n)$ directly given the binary representation of n . The number of rounds required by the algorithm is linear in n and very close to the optimal value, differing only by $\lfloor \log_2(n-1) \rfloor + 2 - \beta(n) - b(n)$.

5 An Optimal Factorization Algorithm

It turns out that optimal schedules of the dance party problem can be found by relating the problem to a graph theory problem, namely, *one- and near-one-factorizations of complete graphs* [?].

Given a graph $G = (V, E)$, a *one-factor* of G is a set of pairwise vertex-disjoint edges that partitions the set of vertices V , and a *one-factorization* of G is a set of one-factors, F , that partition the set of edges E . Obviously, in order for a graph to have a one-factorization, it is necessary that it have an even number of vertices. If G has an odd number of vertices, the corresponding concepts are near-one-factor and near-one-factorization. A *near-one-factor* of G is a set of pairwise disjoint edges covering all vertices of G but one. A *near-one-factorization* of G is a set of near-one-factors that partitions the set of edges E .

By identifying the people at the dance party as the vertices V of a graph G and using the dance relationship to form (undirected) edges E , finding an optimal schedule for the dance party problem can be cast as the one- or near-one-factorization problem of a complete graph, depending on whether n is even or odd; the reason that the resulting graph of the dance party problem is complete is that each person needs to dance with each other exactly once. A dance round corresponds to a one- or near-one-factor, and a schedule for the entire party corresponds to a one- or a near-one-factorization of the graph.

A standard way to construct a one-factorization $F = \{F_0, \dots, F_{2n-1}\}$ of a complete graph K_{2n} with $2n$ vertices $v_0, v_1, \dots, v_{2n-1}$ is as follows [?]: for $i = 0, \dots, 2n - 1$,

$$F_i = \{(v_0 v_{i+1})\} \cup \{(v_{i+j+1} v_{i-j+1}) \mid j = 1, \dots, n - 1\},$$

where in the subscripts $\underline{k} = k \bmod (2n - 1)$. This construction can be illustrated by the following circle figure, where the sections are equally spaced on the circle with vertex 0 at the center. The factor F_0 shown consists of one radical line and $n - 1$ chords. The factors F_1, \dots, F_{2n-2} are obtained by rotating the nodes of the whole figure counter-clockwise through an angle $\theta = 2\pi/(2n - 1)$, then 2θ and so on. A near-one-factorization of the complete graph K_{2n-1} can be constructed \wr from a one-factorization of the complete

graph K_{2n} by deleting all the edges involving the vertex $2n - 1$.

For example, the one-factorization of K_6 constructed as above is

$$F_0 = \{(01)(25)(34)\}, F_1 = \{(02)(31)(45)\}, F_2 = \{(03)(42)(51)\}, F_3 = \{(04)(53)(12)\}, F_4 = \{(05)(14)(23)\}.$$

The near-one-factorization of K_5 reduced from the one-factorization of K_6 is

$$F_0 = \{(01)(34)\}, F_1 = \{(02)(31)\}, F_2 = \{(03)(42)\}, F_3 = \{(04)(12)\}, F_4 = \{(14)(23)\}.$$

Based on the above constructions of one- and near-one-factorizations, a schedule G_n for the dance party can be defined as follows: for any i , $0 \leq i < n$, and any r , $0 \leq r < 2\lfloor(n+1)/2\rfloor - 1$,

$$G_n(i+1, r+1) = G'_n(i, r) + 1,$$

where

$$G'_{2m}(i, r) = \begin{cases} r+1 & \text{for } i=0 \\ 0 & \text{for } i=r+1 \\ (2r-i+1)(\text{mod } (2m-1)) + 1 & \text{otherwise} \end{cases}$$

and

$$G'_{2m-1}(i, r) = \begin{cases} i & \text{if } G'_{2m}(i, r) = 2m-1 \\ G'_{2m}(i, r) & \text{otherwise.} \end{cases}$$

For example, the schedules G_6 and G_5 are

	1	2	3	4	5
1	2	3	4	5	6
2	1	4	6	3	5
3	6	1	5	2	4
4	5	2	1	6	3
5	4	6	3	1	2
6	3	5	2	4	1

	1	2	3	4	5
1	2	3	4	5	1
2	1	4	2	3	5
3	3	1	5	2	4
4	5	2	1	4	3
5	4	5	3	1	2

Finally,

Proposition 5 *For any $n > 0$,*

$$t(G_{2n}) = 2n - 1 \quad \text{and} \quad t(G_{2n-1}) = 2n - 1,$$

and hence G_n is optimal.

Proof. This is because a one-factorization of the complete graph K_{2n} has $2n - 1$ factors and a near-one-factorization of the complete graph K_{2n-1} has $2n - 1$ near-one-factors. These numbers are also the lower bounds for $T(n)$, as given in Proposition 1. \square

According to this proposition, the optimal number of total rounds for the dance party problem is

$$T(n) = \begin{cases} n - 1 & \text{if } n \text{ even} \\ n & \text{if } n \text{ odd.} \end{cases}$$

6 Application to Implementing All-Gather Operation

Given a schedule P , the `all-gather` operation in a complete (fully-connected) network of n processors can be implemented as follows for a processor with ID `my_id`, $1 \leq \text{my_id} \leq n$:

```
for r = 1 to t(P) do
  j = P(my_id, r);
  if j < my_id then
    receive a message from processor j
    send a message to processor j
  else if j > my_id then
    send a message to processor j
    receive a message from processor j
```

```
    end if
end for
```

The semantics of `receive` requires it to block until a message is received. `send` does not complete until `receive` acknowledges it. This enforces the distributed synchronization.

The “sequential” schedule S_n given in Section 2 and the optimal schedule G_n in the previous section have been used to implement the `all-gather` operation in [?] for parallelizing the class of iterative algorithms over clusters of workstations connected via ethernet. As the communication schedule is calculated in advance, the above implementation is particularly important and advantageous. Computer experiments show that the implementation based on the optimal schedule G_n has better performance (about 15% to 20% faster for transmitting messages of length less than 1000 bytes) than the one based on the sequential schedule S_n , even though ethernet allows only one message at any time to be transmitted on an ethernet bus segment. Actual timings are given in [?].

References

- [1] V. Kumar, A. Grama, A. Gupta, G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms* (Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994).
- [2] Message Passing Interface Forum, *MPI: A message-passing interface standard*, Computer Science Department, Technical Report CS-94-230, University of Tennessee, Knoxville, TN, April 1994.
- [3] R. Butler and E. Lusk, *User's Guide to the p4 Programming System*, Argonne National Laboratory, October 1992.

- [4] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Programming* (The MIT Press, Cambridge, MA, 1994).
- [5] W. D. Wallis, *Combinatorial Designs*, (Marcel Dekker, 1988).
- [6] X. Wang and E. K. Blum, "Parallel Execution of Iterative Algorithms on Workstation Clusters", Submitted to *Journal of Parallel and Distributed Computing*, 1995.