# LOAD-BALANCED TASK ALLOCATION IN LOCALLY DISTRIBUTED COMPUTER SYSTEMS

by

Hongjun Lu
Michael J. Carey

Computer Sciences Technical Report #633

February 1986

# Load-Balanced Task Allocation in Locally Distributed Computer Systems

*Hongjun Lu*
Computer Sciences Center
Honeywell, Inc.
Golden Valley, MN 55427

*Michael J. Carey*
Computer Sciences Department
University of Wisconsin
Madison, WI 53706

This research was partially supported by National Science Foundation Grant Number DCR-8402818, an IBM Faculty Development Award, and the Wisconsin Alumni Research Foundation.

# Load-Balanced Task Allocation in Locally Distributed Computer Systems

*Hongjun Lu*
Computer Sciences Center
Honeywell, Inc.
Golden Valley, MN 55427

*Michael J. Carey*
Computer Sciences Department
University of Wisconsin
Madison, WI 53706

## ABSTRACT

A task force is a distributed program which consists of a set of communicating tasks. This paper presents an algorithm for allocating the tasks in a task force to a set of execution sites in a locally distributed computer system. The main objective of this dynamic task allocation algorithm is to balance the load of the system, and the algorithm takes the current load status of the system into account when making its allocation decisions. In addition to balancing the system's load, the algorithm also tries to minimize the communications costs between the tasks to the extent possible within the constraints imposed by load balancing. The paper begins by quantitatively defining the load unbalance factor, and the problem of load-balanced task allocation is then defined. A heuristic algorithm for finding solutions to the problem is presented, and allocations generated by the heuristic algorithm are compared to those obtained via exhaustive search. The results of this comparison indicate that the heuristic algorithm finds the optimal allocation in most cases. The execution time and the complexity of the heuristic task allocation algorithm are also addressed.

## 1. INTRODUCTION

### 1.1. The Problem

Distributed computing systems have been an active research area in computer science for the last decade, and task allocation and load balancing have been a major issue associated with such systems. When a task or set of tasks are to be executed in a distributed system, they must be properly allocated to sites in the system. The problem of selecting appropriate sites is known as the *task allocation* problem, and it is often treated as being a static problem. That is, the problem is often defined according to the static characteristics of the tasks, such as their memory requirements, expected execution times, and communication patterns, and the static characteristics of the system, such as the number of processors available, the amount of memory associated with each processor, the processing speed of the processors, and the speed of the communications network. A number of solution methods have been developed for this type of task allocation problem, including integer programming methods [Gyly76, Pric79, Pric84], graph theoretic methods [Ston77a, Ston77b, Ston78, Rao79, Bokh79], branch-and-bound search methods [Ma82], and various heuristic methods [Gyly76, Chu80, Pric84]. The common feature of all

This research was partially supported by National Science Foundation Grant Number DCR-8402818, an IBM Faculty Development Award, and the Wisconsin Alumni Research Foundation.

these solution methods is that they operate using static information.

As Livny and Melman showed, the probability that at least one processor is idle while tasks are waiting at other sites in a distributed system (a "wait while idle" state) can be remarkably high over a wide range of network sizes and processor utilizations [Livn82]. *Load balancing* aims to reduce this probability by properly distributing and/or redistributing the load on the system's resources. As a result, the queuing times of the tasks will be reduced, and better system performance can be obtained. A mechanism for achieving a load-balanced system is *task migration*, which is the process of transferring partially executed tasks from one site to an idle or less heavily loaded site to continue execution there. However, the boundary between task allocation and task migration schemes can be somewhat fuzzy — in one class of task migration schemes, known as sender-initiated task migration schemes, tasks are migrated only upon arrival in the system, and only by the site at which they arrive. Recent results reported by Eager, Lazowska, and Zahorjan [Eage85] comparing such strategies with other strategies indicate that if the system load is light to moderate (which is when load balancing is usually the most effective), or if the cost of transferring executing tasks is significantly greater than the cost of transferring newly arrived tasks, then sender-initiated strategies are recommended. To distinguish this type of task allocation scheme from those based purely on static information, we refer to this type of dynamic, load-based approach to task allocation as *load-balanced task allocation.*

## 1.2. Load-Balanced Task Allocation

In this paper we propose a heuristic algorithm for solving a load-balanced task allocation problem. The particular problem that we consider here is unique as compared to most previous load balancing work, as we address the problem of selecting execution sites for an *entire distributed program* consisting of a number of tasks which are to be executed concurrently and which communicate with one another. Such distributed programs are known as *task forces* [Jone79], and we consider the problem of dynamically allocating the tasks in a task force to sites in a locally distributed computer system.

Our work on the dynamic task allocation problem originated from the problem of processing queries in a locally distributed database system in a load-balanced fashion. In distributed relational database systems, queries are often decomposed into sequences of *data moves* and *subqueries,* where each subquery is a set of database operations that are to be performed together at a single site. If the distributed database includes replicated data, it is likely to be possible to execute each subquery at one of several sites that contains the data which it references, and in this case it is necessary to somehow select a site at which to execute each subquery. Furthermore, research in the query processing area [Smit75, Yao79], including our own work on processing distributed joins in locally distributed database systems [Lu85a], indicates that pipelined execution of the subqueries of a query can improve performance significantly. When subqueries are executed in a pipelined fashion, they execute concurrently with intermediate results being transferred from subquery to subquery as they become available. The problem of allocating subqueries to sites in a locally distributed database system in a load-balanced fashion is a variant of the load-balanced task allocation problem.

The load-balanced task allocation problem addressed in this paper differs from previous work on task allocation in several ways. First, and most importantly, load-balanced task allocation is a dynamic problem. It takes a task force and the current load status of the system as inputs, and its main objective is to achieve a load-balanced system (as defined later) dynamically through proper task allocation. To our knowledge, previous task allocation work has not addressed this problem. Second, given load balancing as the primary objective, the communications cost for executing the tasks is minimized as the secondary objective. Achieving load balancing and minimizing the inter-task communications are conflicting goals in general [Chu80], and most previous task allocation work has been more concerned with minimizing communications costs. However, several researchers have reported that in locally distributed database systems, local processing costs are usually a more important factor than communications costs [Page83, Lu85a], and also that load balancing can have a bigger impact on system performance than communications costs do in such systems [Care85a]. Thus, our algorithm seeks to choose the allocation with the lowest communications cost from the set of allocations that provide an equivalent degree of load balancing. Third, each task in the task force is assumed to have a *feasible assignment set* that specifies the sites to which the task may be allocated. This constraint arises in our application (distributed query processing) due to the fact that each piece of data is stored only at a subset of the sites in the system. Finally, since load-balanced task allocation is dynamic in nature and is to be accomplished quickly at runtime, we require solution methods that are capable of solving the problem in a reasonably short time.

### 1.3. Paper Outline

The remainder of this paper presents our algorithm and its evaluation. Section 2 defines the load-balanced task allocation problem more precisely, and Section 3 describes the details of a heuristic algorithm that solves this problem. In Section 4, we present an evaluation of the optimality of the allocations generated by our heuristic task allocation algorithm, and we also discuss the complexity of the algorithm and its actual execution time. Two enhancements to the algorithm are also described in Section 4. Section 5 briefly discusses some simulation results that indicate the effectiveness of load-balanced task allocation. Finally, Section 6 presents our conclusions and outlines our plans for future work.

## 2. THE LOAD-BALANCED TASK ALLOCATION PROBLEM

The proper representation and accurate estimation of load information is a key problem in load balancing algorithms. The basic requirement for such representation and estimation is that it must be simple, effective, and efficient. The representation of load information should be simple enough so that the inter-computer exchange of load information introduces as little additional overhead to the system as possible. To be effective and efficient, the load estimate must be fairly accurate and able to be evaluated in a short time period in order to reflect the current status of the system.

In a number of research efforts, the load of a processing site has been defined as its degree of "busyness". According to this view, the load of a site $s_i$, $L(s_i)$, is represented by the number of tasks being currently served at that site, $N(s_i)$ [Ni81, Livn83]. That is,

$$LD(s_i) = N(s_i)$$

Under this definition, site $s_i$ is idle when $LD(s_i) = 0$. When $LD(s_i) > 1$, there are tasks waiting to be executed at site $s_i$. (This assumes that a processing site consists of a single server.)

In order to quantitatively describe the degree of "balancedness" of a system, Livny defined the *load unbalance factor* of a system, *UBF*, as the maximum value of the load differences between the $n$ sites in a system [Livn83]. That is,

$$UBF = \max_{1 \le j,k \le n} | LD(s_j) - LD(s_k) |$$

This definition is fine for use in allocating single tasks to sites as long as the tasks are not constrained as to the sites where they can execute (i.e., as long as their feasible assignment sets contain every site). However, if the tasks do have execution site constraints, this definition is no longer very useful. For example, if the possible execution sites for every task in a newly arrived task force happen to exclude the pair of sites with the current maximum and minimum loads in the system, then the *UBF* as defined above will remain constant regardless of how tasks are assigned to sites. Using the concept of *load variance*, the unbalance factor can instead be defined as the variance of the load distribution in the system as follows:

$$UBF = \frac{\sum_{j=1}^{n} (LD(s_j) - \overline{LD})^2}{n} = \frac{\sum_{j=1}^{n} (N(s_j) - \overline{N})^2}{n}$$

where $N(s_j)$ is the number of tasks at site $s_j$ and $\overline{N}$ is the average number of tasks per site after all query units are allocated.

Using this definition of the load unbalance factor, the load-balanced task allocation problem for a locally distributed system with $n$ sites, $\{s_1, \ldots, s_n\}$, can now be expressed as follows:

**Given:**

(1) A number of tasks $\{t_1, \ldots, t_m\}$, to be executed concurrently as a task force.

(2) A feasible assignment set $S_i = \{s_{i_1}, \ldots, s_{i_k}\}$ for each task $t_i$ $(1 \le i \le m)$.

(3) An *initial load vector* specifying the intial load at each site $s_j$, $1 \le j \le n$, as given by $LD(s_j)$.

**Find:** An allocation of tasks to processing sites such that:

(1) The unbalance factor under this allocation is minimized.

(2) The total communications cost, measured as the sum of the total data transfers between communicating tasks that are allocated to different sites, is also minimized to the extent possible without increasing the unbalanced factor of the allocation.

In the following discussion, we assume that the $m$ tasks are executed in a pipelined fashion (because of our application [Care85b, Lu85b]). In this case, communication only occurs between a pair of adjacent tasks in the pipeline, and the total communications cost can be roughly approximated by the number of nonlocal task pairs (i.e., pairs of adjacent subtasks $t_i$ and $t_{i+1}$ that are allocated to different sites). However, we will see later that these assumptions will not affect the generality of our algorithm, and that only a few minor modifications are required in order to apply the algorithm to task forces with more general communication patterns. Our only major assumption regarding the tasks in the task force is that they are all to be initiated simultaneously and executed concurrently. In other words, the allocation of a task to a site is assumed to introduce its load at that site immediately.

Since the objective of load balancing is quantitatively well defined, the load-balanced task allocation problem can be formalized [Lu85b]. However, there are several difficulties involved in trying to apply non-heuristic solution methods to solve the problem. Aside from the computational complexity when the number of sites becomes large, the main difficulty in trying to use an integer programming approach is that the two objective functions, minimizing the load unbalance factor and the communications cost, are in conflict. They therefore cannot be simply treated as a composite function. Another solution method, the branch-and-bound search method, is also unusable here because the main evaluation function, the unbalance factor *UBF*, cannot be computed during the expansion of a search tree — assigning a task to a site may either increase or decrease the *UBF*, so the unbalance factor of an allocation is computable only after all tasks have been assigned to sites. That is, all possible allocations would end up being generated, as in exhaustive search methods. Perhaps the most important consideration is that load-balanced task allocation is to be performed at runtime, so it should introduce as little overhead as possible. Heuristic methods requiring less computational effort and providing near-optimality are therefore preferable.

## 3. A HEURISTIC APPROACH

In this section, we present a heuristic algorithm that finds a solution to the load-balanced task allocation problem defined in the last section. The input of the algorithm includes a *feasible assignment set* for each task and an *initial load vector* that specifies the number of existing processes (i.e., tasks) at each site at the time the tasks are to be allocated. The originating site and result site of the task force, which are the site from which the task force originates and to which the results of its computation are to be returned, are also specified as input. These sites are included as input so that the communications costs for initiating tasks at remote sites and for returning results to the result site (assuming that there is a user awaiting results at that site) will be considered by the allocation algorithm. For this purpose, two dummy tasks, $t_0$ and $t_{m+1}$, are created. These tasks are assumed not to introduce any real load at their sites, but their communications costs will play a role in influencing the allocation decision if they are allocated remotely.

## 3.1. Task Allocation Heuristics

Before we present the algorithm itself, we first describe a set of heuristics that are used in the algorithm. Three main heuristics are employed by the algorithm — a heuristic to control the order in which tasks are considered for allocation, a heuristic to avoid considering sites with particularly heavy loads, and a heuristic to help ensure that a good allocation site is selected for each task.

### 3.1.1. Heuristic 1: Order of Allocation

In order to guarantee fast execution, our algorithm operates by allocating tasks to sites one by one without backtracking or reconsidering previous task allocation decisions. As a result, the order in which the tasks in a task force are considered for allocation is a critical factor affecting the optimality of the resulting allocations. The heuristic that we use to control the allocation order in our algorithm is that tasks with less flexibility about where they can run should be considered ahead of any tasks with greater flexibility. The notion of the *degree of freedom* of a task is used for this purpose, and two freedom-related metrics are used by the algorithm.

The first degree of freedom metric is the *static freedom* of a task, which is defined as the number of sites where it can be allocated (i.e., as the cardinality of its feasible assignment set). If the static freedom of a task is one, it has no freedom at all — it must be allocated to the only site in its feasible assignment set. A task with a higher static freedom value will be relatively more flexible than one with a lower value because there are more site choices available. The second freedom metric has a more dynamic flavor. The *dynamic freedom* of a task is defined as the sum of the current load of the sites in its feasible assignment set (where the current load of a site is the sum of its initial load plus the number of tasks in the task force which have been assigned to the site thus far). In cases where two tasks have the same static freedom value, the task whose feasible assignment sites are more heavily loaded (i.e., the task with the larger dynamic freedom measure) will usually have fewer execution site options.

Using these freedom measures, tasks are allocated by our heuristic algorithm one by one in order of their degree of freedom. That is, tasks with fewer site choices are allocated earlier, and the degree of static freedom is the primary consideration. Tasks are considered in increasing order of static freedom. In the event of a tie, the task with the largest dynamic freedom measure is chosen, as its candidate sites are more heavily loaded. Note that these freedom measures are opposite in how they must be interpreted, as a larger static freedom value implies more site choices, whereas a larger dynamic freedom value implies fewer choices.

### 3.1.2. Heuristic 2: Elimination of Full Sites

A heuristic used to reduce the task allocation search space and hence the amount of computation required for allocation is that a site with a current load which is larger than the expected post-allocation average load is considered to be full. Sites that become full will not be assigned further tasks (except if such a site is the only site in the feasible assignment set for a given task). Thus, whenever a site is full, whether due to its initial load or to the assignment of a new task to the site, it is deleted from the feasible assignment set of all tasks. Since this changes both the static and dynamic degrees of freedom for the affected tasks, their degrees of freedom must be updated throughout the allocation process.

### 3.1.3. Heuristic 3: Site Selection Criteria

After determining the task to be considered next, up to four metrics will be used to guide the selection of a site from among the sites in its feasible assignment set). These four metrics are:

(1) The *current load* of a candidate site: This is the total number of tasks currently assigned to the site, including both the tasks in the initial load of the site plus any tasks that have already been assigned to it during the allocation process for the task force.

(2) The *potential load* of a candidate site: This is the number of unassigned tasks that have the site in their feasible assignment set.

(3) The *benefit* of a possible assignment: This is the communications cost that can be avoided by this assignment. In our case, since communications occur only between adjacent tasks in the pipeline, the benefit of assigning task $t_i$ to site $s_j$ is the number of adjacent tasks ($t_{i-1}$ or $t_{i+1}$) that have already been allocated to the site (either 0, 1, or 2). If task $t_{i-1}$ (or $t_{i+1}$) is already allocated to site $s_j$, and $t_i$ is then allocated to the same site $s_j$, the communications cost between $t_{i-1}$ (or $t_{i+1}$) and $t_i$ will become zero. If neither $t_{i-1}$ or $t_{i+1}$ has been allocated to site $s_j$, then the assignment of task $t_i$ to site $s_j$ will not lead to a savings in the communications cost.

(4) The *potential benefit* of a possible assignment: This measures the communications costs that might be eliminated by this assignment. In our case, this is the number of unassigned tasks which have the site in their feasible assignment set and which would form a consecutive sequence of tasks including the task being allocated if they were also assigned to the site. For example, consider the allocation of task $t_i$ to site $s_j$. If $s_j$ is in the feasible assignment set of $t_{i-1}$ (or $t_{i+1}$), and this task has not been allocated to any site yet, there is a future possibility that the communications cost between $t_i$ and $t_{i-1}$ (or $t_{i+1}$) could become zero. This would happen if $t_i$ is first assigned to $s_j$, and then $t_{i-1}$ (or $t_{i+1}$) is later assigned to the same site. Compared with assigning $t_i$ to a site $s_k$ that is not in the feasible assignment set of $t_{i-1}$ or $t_{i+1}$, then, assigning $t_i$ to $s_j$ has the potential of being beneficial.

Since load balancing is the main objective, the evaluation order for these four metrics during the allocation process is: minimum current load, maximum benefit, minimum potential load and maximum potential benefit. First, an attempt is made to allocate task $t_i$ to the site with the minimum current load among its feasible assignment sites. In the event of a tie, the benefit of assigning $t_i$ to each site with the minimum load is calculated, and the site with the maximum benefit is selected. In case of another tie, then the site with the minimum potential load is chosen. Finally, the maximum potential benefit site is considered if there is more than one site with the same minimum potential load. It is important to notice that not all four metrics are calculated for every task. For most tasks, only one or two metrics will need to be evaluated. Furthermore, the size of the candidate site set gets smaller and smaller as each metric is considered in turn. This is advantageous for minimizing the cost of the site selection process.

## 3.2. The Basic Load-Balanced Task Allocation Algorithm

Given the heuristics just described, the basic load-balanced task allocation algorithm can now be described as follows:

(1) Allocate the dummy task $t_0$ to the task force's site of origin, and allocate the other dummy task $t_{m+1}$ to the result site for the task force.

(2) Compute the static and dynamic freedom metrics for each task $t_i$, $1 \le i \le m$.

(3) Select the next task to be allocated as the one with the least assignment flexibility using the degree of freedom metrics.

(4) Select an allocation site $s_j$ for this task ($t_i$) by choosing the site from its feasible assignment set with the least current load, considering the benefit, potential load, and potential benefit metrics in turn as necessary (to break any ties).

(5) Increment the load of site $s_j$, and recompute the freedom metrics of any unallocated tasks that have $s_j$ in their feasible assignment set. (If $s_j$'s load has exceeded the expected average, simply delete $s_j$ from their feasible assignment sets.)

(6) If any unassigned tasks remain, go to step 3 and repeat the process.

Figure 3.1 shows an example input for the load-balanced task allocation algorithm. The example task to be allocated consists of 5 tasks, and there are 8 sites in the system. The algorithm is applied as follows:

i. The total initial load is 12, so the expected average load after allocation is $\lceil (12+5) / 8 \rceil = 3$. Site $s_1$ is therefore full initially, so it is deleted from the feasible assignment sets of $t_1$ and $t_4$.

ii The static degrees of freedom for the tasks are $\{1, 3, 3, 4, 1, 5, 1\}$. The dummy tasks $t_0$ and $t_6$ are allocated to $s_3$ first.

iii. $t_4$ is allocated to $s_6$, the only site in its feasible assignment set.

iv. Both $t_1$ and $t_2$ have the same static degree of freedom value of 3. The feasible assignment set of $t_1$ is $\{s_2, s_5, s_7\}$, with the relevant part of the current load vector being $\{1, 2, 1\}$, and $t_2$'s is $\{s_2, s_4, s_5\}$, with the relevant part of the current load vector being $\{1, 2, 2\}$. Thus, $t_2$ has less dynamic freedom, so $t_2$ is allocated next.

v. $t_2$ is allocated to site $s_2$, as $s_2$ is the site with the minimum load (of 1) in its feasible assignment set. (This increases $s_2$'s load by 1.)

An Example Input:

| | |
|---|---|
| number of tasks | 5 |
| number of sites in the database | 8 |
| originating site | $s_3$ |
| result site | $s_3$ |
| initial load vector | {4, 1, 1, 2, 2, 0, 1, 1} |
| feasible assignment set | |
| $t_1$ | $\{s_1, s_2, s_5, s_7\}$ |
| $t_2$ | $\{s_2, s_4, s_5\}$ |
| $t_3$ | $\{s_3, s_5, s_7, s_8\}$ |
| $t_4$ | $\{s_1, s_6\}$ |
| $t_5$ | $\{s_3, s_4, s_5, s_6, s_7\}$ |

Figure 3.1: An example for load-balanced task allocation.

vi. $t_1$ is now allocated to site $s_7$ for the same reason as in v.

vii. Since $t_3$ is next in increasing order of static freedom, it is considered next. Its feasible assignment set and the relevant part of the current load vector are $\{s_3, s_5, s_7, s_8\}$ and {1, 2, 2, 1}, respectively. The minimum load sites for $t_3$ are $\{s_3, s_8\}$. As for benefits, benefit($t_3$,$s_3$) = benefit($t_3$,$s_8$) = 0. Finally, potential_load($s_3$) = 2, but potential_load($s_8$) = 1. $t_3$ is thus allocated to site $s_8$.

viii. Finally, $t_5$ is considered: Its feasible assignment set is $\{s_3, s_4, s_5, s_6, s_7\}$, and its corresponding load vector is {1, 2, 2, 1, 2}. Both $s_3$ and $s_6$ have the same current load of 1, but benefit($t_5$, $s_3$) = 0 and benefit($t_5$, $s_6$) = 1 (since $t_4$ was already allocated to $s_6$ in (iii)). Thus, $t_5$ is allocated to site $s_6$.

ix. The final assignment is:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| task | ($t_0$) | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ | ($t_6$) |
| execution site | ($s_3$) | $s_7$ | $s_2$ | $s_8$ | $s_6$ | $s_6$ | ($s_3$) |

After assignment, the load vector is {4, 2, 1, 2, 2, 2, 2, 2}. The *UBF* for this assignment is 0.61, and its total communications cost is 5. Comparing this result with the optimal allocation, which can be obtained by the exhaustive search method described in the next section, it turns out that this is actually an optimal allocation under the problem definition.

## 4. PERFORMANCE AND ENHANCEMENTS OF THE BASIC ALGORITHM

The algorithm described in the last section is a greedy algorithm in the sense that tasks are assigned one by one, without looking ahead or backtracking. However, the potential load and potential benefit metrics are used to improve its optimality. Also, the task allocation order and the order in which the four metrics are considered was carefully designed. In order to evaluate the quality of the allocations generated by our heuristic algorithm, we conducted a study in which we compared its allocation decisions to those of an exhaustive search method which always finds the optimal allocation. In this section, we describe the results of this study, and we also present two enhancements that can be used to improve the optimality of the algorithm at little additional cost.

## 4.1. Optimality of the Algorithm

In our study, task allocations obtained using the heuristic algorithm were compared with their corresponding optimal allocations. A task force generator was used to generate inputs like that in the example of Figure 3.1. Both an exhaustive search program that always finds the optimal allocation and our heuristic algorithm were then applied to these task forces. The optimality of the heuristic algorithm can be characterized by the percentage of its allocations that have the same *UBF* values and total communications costs as the optimal allocations over a number of tests.

The parameters that controlled the task generator were the number of tasks to be allocated at the same time (i.e., the number of tasks comprising the task force), the number of sites in the system, the maximum initial load and the average size of the feasible assignment set of a task. The maximum initial load parameter controls the initial unbalance of the system, with the initial load of each site assumed to be uniformly distributed between zero and this maximum value. The average size of the feasible assignment set determines the number of sites a task can be allocated to. (In a database environment, this is used to model the extent of data replication, as a subquery can only be allocated to a site where the data it references is stored).

In order to find the optimal allocation for an input, an exhaustive search tree was constructed according to the feasible assignment sets of the tasks. Figure 4.1 shows the search tree for the example input of Figure 3.1. Each level of the tree corresponds to the possible allocations of one of the tasks, with each node at a level representing one site in the feasible assignment set of the task. (The dummy tasks $t_0$ and $t_6$ are not shown in the tree since each always has just one possible allocation site.) This tree is evaluated after all possible assignment sites have been considered by computing the unbalance factor and the communications cost along each branch of the tree. The branch that gives the minimum unbalance factor and communications cost represents the optimal allocation.

Three groups of test were conducted in the study in order to investigate the optimality of the heuristic algorithm:

(1) The first group of tests studied the general behavior of the algorithm. The number of tasks ($m$) was varied from 3 to 6. The number of sites in the system ($n$) was varied from 4 to 12. The maximum load was randomly chosen in the range from 4 to 12, and the average size of the feasible assignment sets was chosen to be the half of the number of sites.

(2) In the second group of tests, the maximum initial load was varied to investigate the performance of the heuristic algorithm when the initial unbalance factor was changed. The number of tasks ($m$) was fixed at 5, and the number of sites ($n$) was fixed at 8. The average size of the feasible

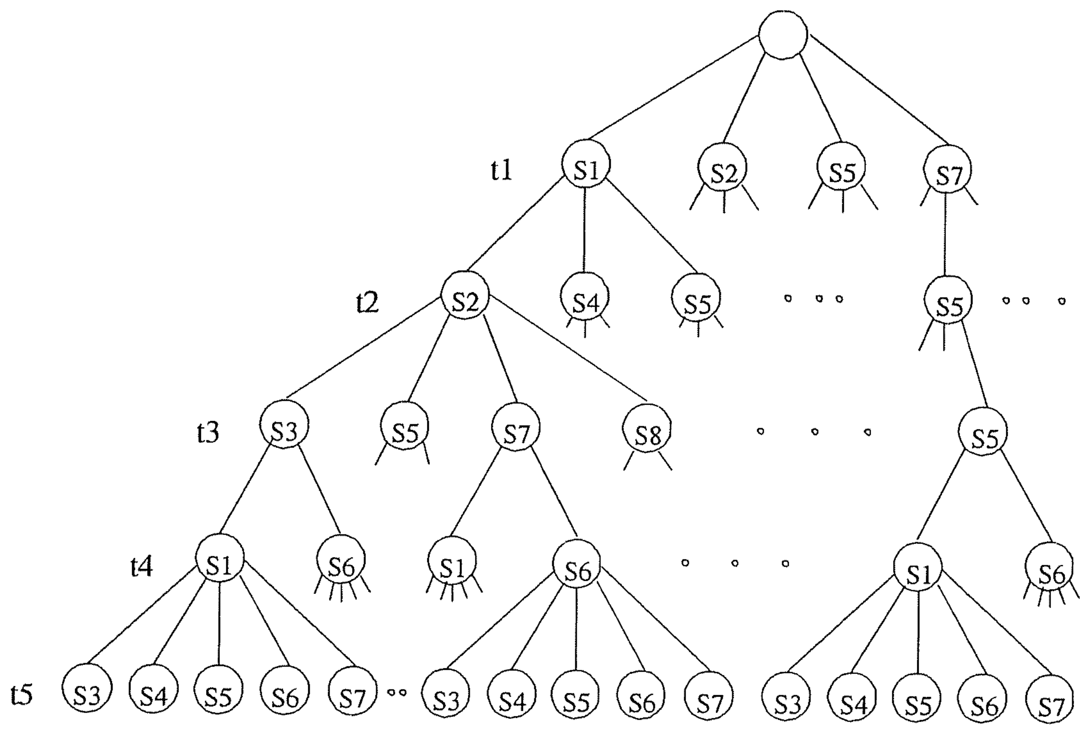

Figure 4.1: A search tree for finding the optimal allocation.

assignment sets was fixed at 4 for this group of tests.

(3) In the third group of tests, the size of the feasible assignment sets was varied. The number of tasks ($m$) and sites ($n$) were fixed at 5 and 8, respectively. The average size of the feasible assignment sets was set to 8 (meaning that a task can be allocated to any site in the system), then to 4 (meaning that the number of sites that a task can be allocated to is half of the total number of sites in the system), and finally to 2 (in which case there are only two choices about where to allocate a task).

The test results are summarized in Tables 4.1 to 4.3. In these tables, the percentage of optimal allocations obtained by using the heuristic algorithm is given for each test. Also, the percentage of allocations that are optimal under each of the objective function alone is given. (These allocations either have the same unbalance factor as the optimal allocations but a larger communications cost, or have the same or lower communications cost but a larger unbalance factor). The percentages shown in Table 4.1 were computed by averaging the results obtained using five different (even) values of $n$ between 4 and 12 (inclusive). The tables indicate that, in most cases, the heuristic algorithm generated the optimal allocations. The percentage of optimal allocations is at least 75%, and typically higher, for the runs in

| Optimality of the Heuristic Algorithm (Test 1) | | | | | |
|---|---|---|---|---|---|
| m | n | total runs | Percentage of the Optimal Allocations (%) | | |
| | | | optimal allocations | optimal on UBF | optimal on CC |
| 3 | 4-12 | 500 | 96.8 | 99.2 | 97.6 |
| 4 | | | 90.8 | 98.2 | 92.4 |
| 5 | | | 83.6 | 98.0 | 85.6 |
| 6 | | | 75.8 | 94.8 | 80.6 |

Table 4.1: Optimality in general (Test 1).

| Optimality of the Heuristic Algorithm (Test 2, m=5, n=8) | | | | |
|---|---|---|---|---|
| max. init. load | total runs | Percentage of the Optimal Allocations (%) | | |
| | | optimal allocations | optimal on UBF | optimal on CC |
| 2 | 100 | 79 | 98 | 81 |
| 8 | | 85 | 98 | 87 |
| 16 | | 93 | 100 | 93 |

Table 4.2: Optimality versus the initial load (Test 2).

| Optimality of the Heuristic Algorithm (Test 3, m=5, n=8) | | | | |
|---|---|---|---|---|
| avg. num. feasible sites | total runs | Percentage of the Optimal Allocations (%) | | |
| | | optimal allocations | optimal on UBF | optimal on CC |
| 2 | 100 | 97 | 98 | 99 |
| 4 | | 85 | 98 | 87 |
| 8 | | 26 | 100 | 26 |

Table 4.3: Optimality versus the number of feasible sites (Test 3).

Test 1. About 95% of the allocations obtained in Test 1 are optimal if only the unbalance factor is considered, which is encouraging since load balancing is our primary objective. As for the communications costs, more than 80% of the allocations have the same (or lower) cost values as the optimal allocations.

As the number of sites and tasks is increased, however, the number of optimal allocations found by the heuristic algorithm decreases. The explanation for this is that larger numbers of sites and tasks increase the number of possible allocations, so the probability of selecting an optimal allocation using a greedy algorithm decreases. It is then more likely that a nearly optimal allocation will be chosen instead of the true optimum. This is especially clear in the case where each task is capable of being executed at any of the sites in the system, where the heuristically obtained allocations frequently result in higher communications costs. When a task can be allocated to any site, the exhaustive search method can take advantage of this flexibility to decrease the communications cost, but this is much less true for the heuristic algorithm. On the contrary, having more choices was seen to reduce the chances of the heuristic algorithm generating an optimal allocation.

## 4.2. Enhancing the Basic Algorithm

Two enhancements have been designed to further minimize the communications cost, and hence to improve the overall optimality of the resulting allocation of tasks to sites. Both enhancements start from the allocation obtained using the basic algorithm. Local adjustments are then made to the allocation to decrease its communications cost while keeping the unbalance factor constant.

*Enhancement 1.* Enhancement 1 is applied to each task individually. If a task $t_i$ was assigned to site $s_j$, its feasible assignment set is searched to find another site $s_k$ so that, if $t_i$ is instead assigned to $s_k$, the unbalance factor of the system is not affected but the communications cost decreases. Figure 4.2 shows an example of this enhancement. After applying the basic load-balanced task allocation algorithm, $t_3$ is allocated to site $s_3$. In this case, $s_6$ is in the feasible assignment set of $t_3$, and the initial load of both $s_3$ and $s_6$ is 0. Applying Enhancement 1 will reallocate $t_3$ to $s_6$, which will not affect the unbalance factor but will reduce the communications cost.

Input :

| | |
|---|---|
| number of tasks | 3 |
| number of sites | 6 |
| originating site | $s_5$ |
| result site | $s_5$ |
| initial load vector | {2, 5, 0, 4, 5, 0} |

feasible assignment set

| | |
|---|---|
| $t_1$ | $\{s_1, s_3, s_4\}$ |
| $t_2$ | $\{s_2, s_6\}$ |
| $t_3$ | $\{s_2, s_3, s_5, s_6\}$ |

allocation sites after applying the basic algorithm:

| task | $(t_0)$ | $t_1$ | $t_2$ | $t_3$ | $(t_4)$ |
|---|---|---|---|---|---|
| allocated site | $(s_5)$ | $s_3$ | $s_6$ | $s_3$ | $(s_5)$ |

allocation sites after applying Enhancement 1:

| task | $(t_0)$ | $t_1$ | $t_2$ | $t_3$ | $(t_4)$ |
|---|---|---|---|---|---|
| allocated site | $(s_5)$ | $s_3$ | $s_6$ | $s_6$ | $(s_5)$ |

Figure 4.2: An example of Enhancement 1.

Tables 4.4 to 4.6 show the results for the three optimality tests with Enhancement 1 applied following the application of the heuristic load-balanced task allocation algorithm. It can be seen that there is an improvement in terms of the communications cost. In the case with 4 sites in the feasible assignment sets on the average, the number of optimal allocations increased by 2-10%, while in the case where a task can be allocated to any site in the system, the number of optimal allocations nearly doubled.

*Enhancement 2.* Enhancement 1 adjusts the allocation site for each task individually from within its feasible assignment set. Enhancement 2 takes a more global view. The main idea of this enhancement is to group as many tasks as possible together at the same site without affecting the *UBF*. This enhancement looks at each pair of adjacent tasks $(t_i, t_{i+1})$, where $t_i$ and $t_{i+1}$ have been allocated to two different sites $s_i$ and $s_k$, and it tries to find a third task $t_j (j > i+1)$ which was also allocated to site $s_i$ (i.e., to the same site as $t_i$). If sites $s_i$ and $s_k$ are in the feasible assignment sets of $t_{i+1}$ and $t_j$ respectively, then it

**Optimality of the Heuristic Algorithm (Test 1)**

| m | n | total runs | Percentage of the Optimal Allocations (%) | | |
|---|---|---|---|---|---|
| | | | optimal allocations | optimal on UBF | optimal on CC |
| 3 | 4-12 | 500 | 98.0 | 99.2 | 98.8 |
| 4 | | | 94.0 | 98.2 | 95.6 |
| 5 | | | 89.0 | 98.0 | 91.0 |
| 6 | | | 79.2 | 95.0 | 85.4 |

Table 4.4: Optimality in general (Test 1, Enhancement 1).

**Optimality of the Heuristic Algorithm (Test 2, m=5, n=8)**

| max. init. load | total runs | Percentage of the Optimal Allocations (%) | | |
|---|---|---|---|---|
| | | optimal allocations | optimal on UBF | optimal on CC |
| 2 | 100 | 84 | 98 | 86 |
| 8 | | 88 | 98 | 90 |
| 16 | | 98 | 100 | 98 |

Table 4.5: Optimality versus the initial load (Test 2, Enhancement 1).

**Optimality of the Heuristic Algorithm (Test 3, m=5, n=8)**

| avg. num. feasible sites | total runs | Percentage of the Optimal Allocations (%) | | |
|---|---|---|---|---|
| | | optimal allocations | optimal on UBF | optimal on CC |
| 2 | 100 | 98 | 98 | 100 |
| 4 | | 94 | 98 | 96 |
| 8 | | 43 | 100 | 43 |

Table 4.6: Optimality versus the number of feasible sites (Test 3, Enhancement 1).

is possible to reverse the choice of sites for $t_{i+1}$ and $t_j$ without affecting the *UBF* (since the number of tasks at $s_k$ and $s_j$ will not change). However, this switch eliminates the communications cost between $t_i$ and $t_{i+1}$. For each task $t_i$, $1 \le i \le m-2$, for which $t_{i+1}$ is at a different site, Enhancement 2 looks for such a $t_j$. Figure 4.3 shows an example of this enhancement in action. The allocation for $t_0 - t_5$ before applying Enhancement 2 is $\{s_5, s_2, s_5, s_5, s_2, s_5\}$. It is obvious that reversing the allocation sites for $t_2$ and $t_4$ will not change the *UBF* value, but this will decrease the overall communications cost.

Tables 4.7 to 4.9 shows the test results repeated with both enhancements employed. The results show that the optimality of the algorithm is further improved, and that Enhancement 2 is especially helpful in the case where the size of the feasible assignment set is equal to the number of site in the system (where the optimal allocation is now always found). The optimality of the heuristic approach proposed here has thus been shown to be quite good. While the test results may vary with different input data, it is expected that the general trends will remain the same.

---

Input :

| | |
|---|---|
| number of tasks | 4 |
| number of sites | 6 |
| originating site | $s_5$ |
| result site | $s_5$ |
| initial load vector | {4, 1, 4, 5, 1, 3} |
| feasible assignment set | |
| $t_1$ | $\{s_1, s_2, s_3, s_6\}$ |
| $t_2$ | $\{s_1, s_2, s_5, s_6\}$ |
| $t_3$ | $\{s_3, s_5\}$ |
| $t_4$ | $\{s_2, s_3, s_5, s_6\}$ |

allocation sites after applying the basic algorithm:

| task | $(t_0)$ | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $(t_5)$ |
|---|---|---|---|---|---|---|
| allocated site | $(s_5)$ | $s_2$ | $s_5$ | $s_5$ | $s_2$ | $(s_5)$ |

allocation sites after applying Enhancement 2:

| task | $(t_0)$ | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $(t_5)$ |
|---|---|---|---|---|---|---|
| allocated site | $(s_5)$ | $s_2$ | $s_2$ | $s_5$ | $s_5$ | $(s_5)$ |

Figure 4.3: An example of Enhancement 2.

---

| **Optimality of the Heuristic Algorithm (Test 1)** | | | | | |
|---|---|---|---|---|---|
| m | n | total runs | Percentage of the Optimal Allocations (%) | | |
| | | | optimal allocations | optimal on UBF | optimal on CC |
| 3 | 4-12 | 500 | 99.0 | 99.2 | 99.8 |
| 4 | | | 98.0 | 98.2 | 97.6 |
| 5 | | | 92.8 | 98.0 | 94.8 |
| 6 | | | 85.2 | 95.0 | 90.2 |

Table 4.7: Optimality in general (Test 1, Enhancements 1 & 2).

| **Optimality of the Heuristic Algorithm (Test 2, m=5, n=8)** | | | | |
|---|---|---|---|---|
| max. init. load | total runs | Percentage of the Optimal Allocations (%) | | |
| | | optimal allocations | optimal on UBF | optimal on CC |
| 2 | 100 | 85 | 98 | 87 |
| 8 | | 94 | 98 | 96 |
| 16 | | 99 | 100 | 99 |

Table 4.8: Optimality versus the initial load (Test 2, Enhancements 1 & 2).

| **Optimality of the Heuristic Algorithm (Test 3, m=5, n=8)** | | | | |
|---|---|---|---|---|
| avg. num. feasible sites | total runs | Percentage of the Optimal Allocations (%) | | |
| | | optimal allocations | optimal on UBF | optimal on CC |
| 2 | 100 | 97 | 98 | 99 |
| 4 | | 94 | 98 | 96 |
| 8 | | 100 | 100 | 100 |

Table 4.9: Optimality versus the number of feasible sites (Test 3, Enhancements 1 & 2).

## 4.3. Execution Time and Algorithm Complexity

In addition to the optimality of the allocations generated by the heuristic algorithm, the execution time and complexity of the algorithm are also important concerns. As mentioned earlier, dynamic task allocation will take place at runtime, so its cost must be as small as possible. A test was conducted to measure the CPU time of the algorithm and its enhancements. For this test, the number of sites was fixed at 8, and the average size of the feasible assignment set was varied. The test was run on a VAX 11/780, and CPU times were obtained using functions provided by UNIX. The results are shown in Table 4.10. It can be seen that the execution times are fairly small. With 5 tasks in the task force and 8 sites in the system, even if each task can be allocated to any of the sites, the total CPU time is less than 41 milliseconds. In our database application [Care85b, Lu85b], query task forces are typically small like this, and the time required to initiate a compiled query will easily dominate these (unoptimized) task allocation times.

Another consideration, particularly for larger task forces and systems, is how execution time increases when the number of tasks and sites is increased. The complexity of the basic algorithm can be

| **Execution Time of the Basic Allocation Algorithm (in msec)** | | | | |
|---|---|---|---|---|
| number of tasks | average size of the feasible assignment set | | | |
| | 2 | 4 | 6 | 8 |
| 3 | 17.0 | 21.0 | 22.2 | 24.7 |
| 4 | 23.9 | 26.1 | 28.4 | 31.3 |
| 5 | 27.1 | 33.4 | 34.6 | 40.9 |

| **Execution Time of Enhancements 1 (in msec)** | | | | |
|---|---|---|---|---|
| number of tasks | average size of the feasible assignment set | | | |
| | 2 | 4 | 6 | 8 |
| 3 | 2.1 | 2.3 | 2.7 | 3.0 |
| 4 | 2.2 | 1.9 | 3.1 | 4.6 |
| 5 | 2.0 | 2.7 | 3.4 | 3.7 |

| **Execution Time of Enhancements 2 (in msec)** | | | | |
|---|---|---|---|---|
| number of tasks | average size of the feasible assignment set | | | |
| | 2 | 4 | 6 | 8 |
| 3 | 1.2 | 0.3 | 0.8 | 0.7 |
| 4 | 0.7 | 1.1 | 0.8 | 0.7 |
| 5 | 1.0 | 0.6 | 1.0 | 0.8 |

Table 4.10: The execution time of the allocation algorithm.

| **Complexity of the Basic Algorithm** | |
|---|---|
| Step | Complexity |
| 2 | $O(n)$ |
| 3 | $O(m \log_2 m)$ |
| 4 | $O(n)$ |
| 5 | $O(m)$ |
| TOTAL | $O(max(n,m \log_2 m)$ |

Table 4.11: Complexity analysis per task.

analyzed as follows: Before selecting allocation sites for the tasks, the degree of freedom of each task is calculated (in step 2 of the algorithm). The next task to be allocated is then selected from among all unallocated tasks (in step 3). Site selection then chooses from among all the sites in the feasible assignment set of the task (in step 4), so the number of sites considered is $n$ or less. After a site has been selected, the system load vector and the degree of freedom of the remaining tasks must be updated (in step 5), and the list of tasks must be reordered (sorted) according to their new freedom values. This process is repeated until all $m$ tasks have been allocated. Table 4.11 summarizes the per-task complexity of the basic load-balanced task allocation algorithm. Every task is processed in this way, so the total complexity of the algorithm is $O(max(mn, m^2 \log_2 m))$.

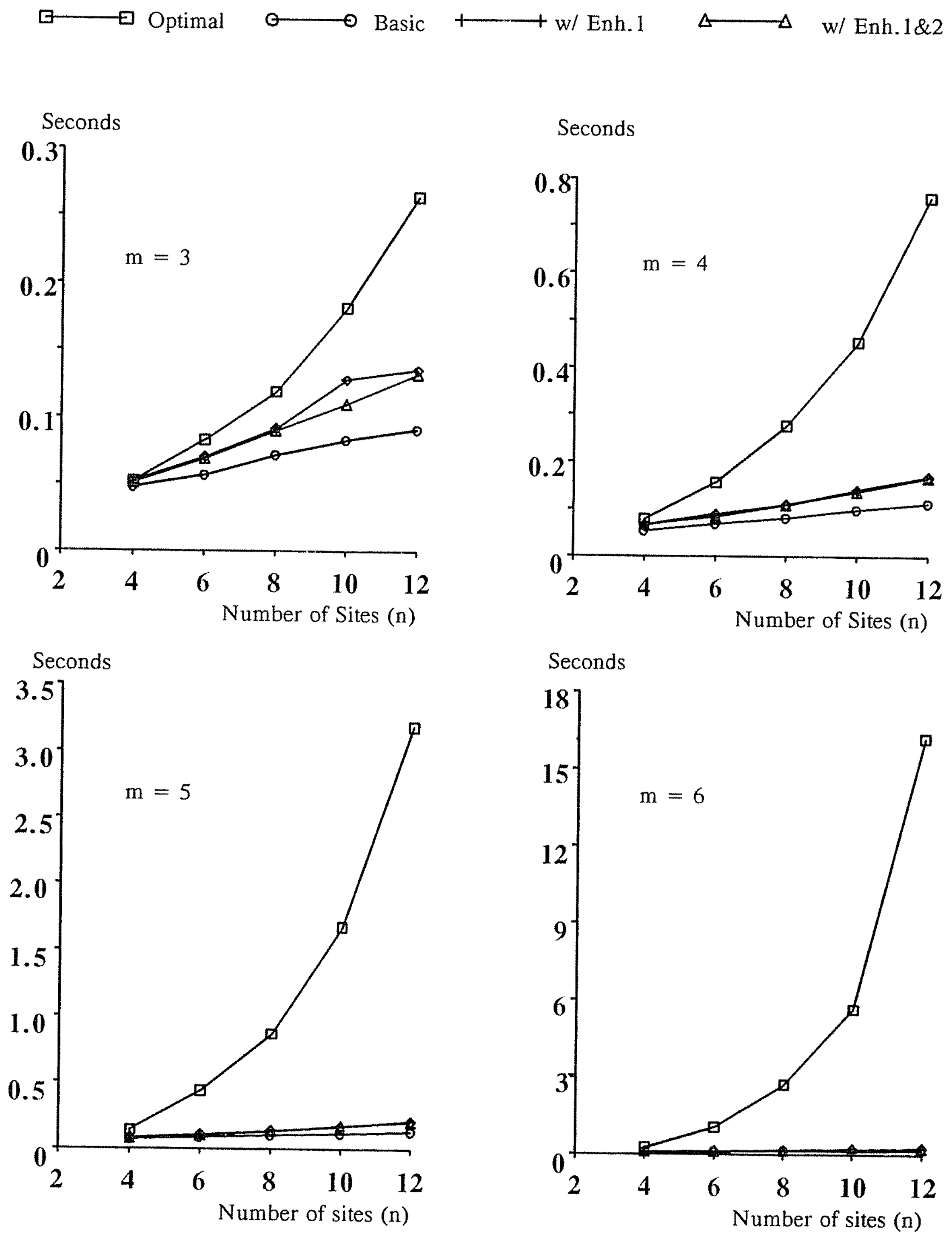

Figure 4.4: The elapsed time of the different algorithms.

In order to see how the heuristic algorithm's execution times compare to those of the exhaustive search algorithm, the elapsed time of both algorithms were measured during the optimality tests presented in Section 4.1. The results are shown in Figure 4.4. These tests were performed on a VAX 11/750 running UNIX with just one user on the system. The figures show that the elapsed time of the heuristic algorithm is indeed basically linear in $n$ when $m$ is fixed, but that the exhaustive search algorithm's time is not. When the number of queries and sites are small, the heuristic algorithm has only a small advantage with regard to execution time. However, when the number of tasks and sites increases, the elapsed time of the exhaustive search increases dramatically. For example, when $m=3$ and $n=4$, its elapsed time is 52.1 milliseconds. For $m=6$ and $n=12$, it becomes 16.2 seconds, or over 300 times longer. In contrast, the execution time of the heuristic algorithm is basically linear in $mn$ here, going from 47.3 milliseconds to 243 milliseconds for these same values of $m$ and $n$.

Another observation from Figure 4.4 is that the two enhancements do increase the computation time somewhat. Thus, applying the enhancement procedures should perhaps be optional. In most cases the basic algorithm performs just fine and it is not necessary to apply the enhancements. In some cases, such as when the number of sites is large or when tasks have a large degree of assignment flexibility, either or both of the enhancement procedures can be applied.

## 4.4. Other Task Force Structures

Because the algorithms and ideas presented in this paper resulted from work on load balancing for distributed database systems [Care85b, Lu85b], our discussions of task forces and our algorithm descriptions have assumed a linear, pipelined task force structure. The ideas discussed here are applicable to more general task force topologies, however. Both the basic load-balanced task allocation algorithm and the first enhancement to the algorithm can be applied directly to more general task force structures, and the complexity of both algorithms will be unaffected. The only difference will be that the communications cost factors are likely to take on a larger range of values (which is not a problem). The second enhancement does have to be slightly modified, however, as it is dependent on the assumption of a pipelined sequence of tasks. To generalize this enhancement, rather than considering pairs of adjacent tasks $(t_i, t_{i+1})$, it will have to consider $t_i$ and each task $t_k$ in the set that $t_i$ communicates with. For each such task $t_k$, it will then have to look for other tasks (outside the set) for exchanging sites with. This will increase the complexity of the second enhancement by a factor dependent on the average number of neighbors for the tasks in the task force. Finally, future work is needed to evaluate the optimality of the plans generated via our heuristic algorithm for other task force topologies, as all of our optimality results were obtained by studying pipelined sequences of tasks.

## 5. AN EXAMPLE APPLICATION

As mentioned in the first section, the load-balanced task allocation algorithm presented here was motivated by the load balancing problem for locally distributed database systems. We have designed and evaluated a load-balanced approach to query processing for locally distributed database systems, and

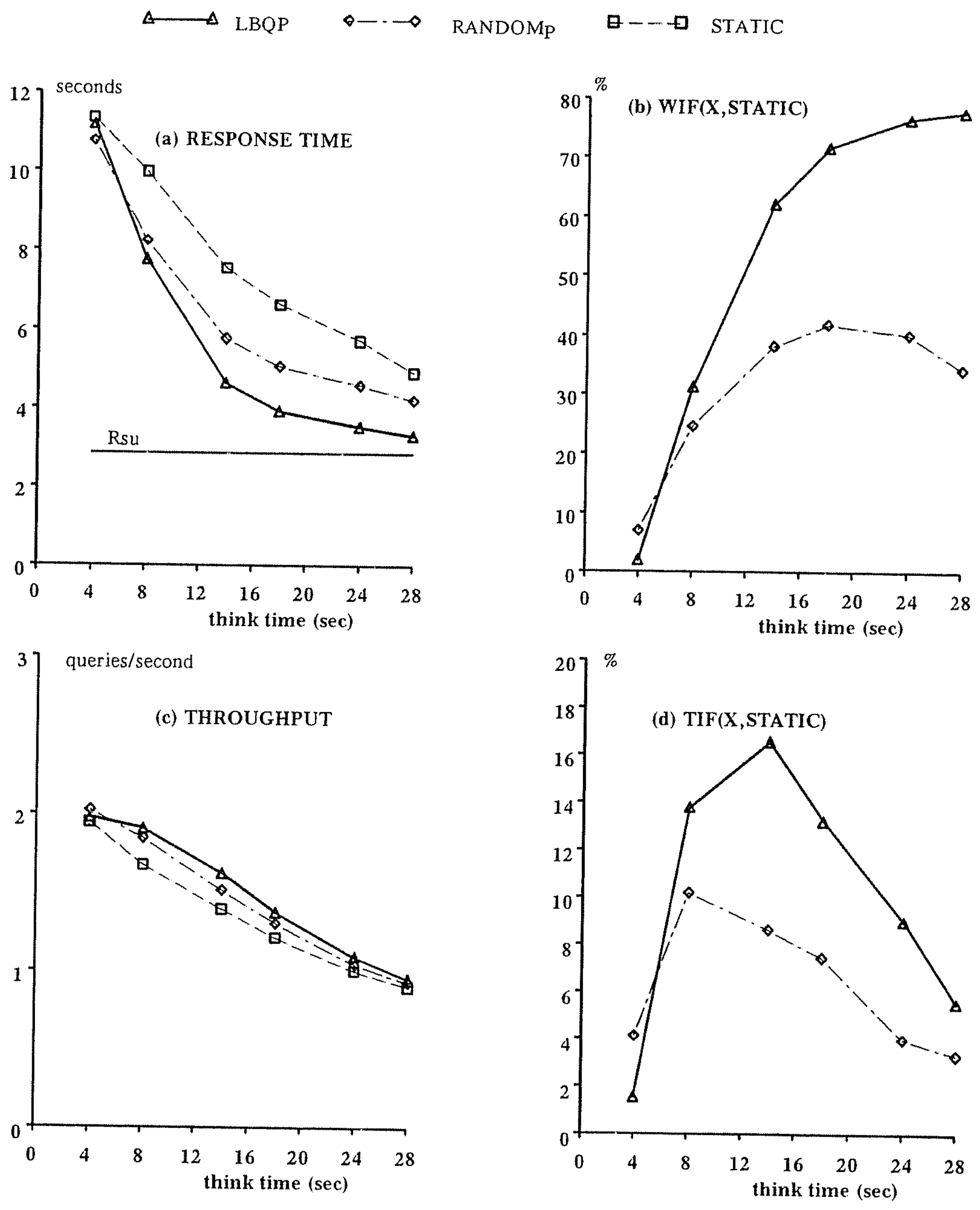

Figure 5.1: Example of performance gains using dynamic task allocation.

this work is reported in [Care85b, Lu85b]. In our approach, query compilation is used to produce a statically-optimized logical query processing plan for a query (i.e., a plan which does not specify which copy of a relation will be accessed), and then a dynamic optimization phase converts this plan into an executable physical plan at runtime. The load-balanced task allocation algorithm described in this paper is used in the dynamic allocation phase in order to select a processing site for each subquery of the overall query from among those sites that have copies of the relation needed by the subquery.

A simulation study was conducted to investigate the performance gains that might be expected in a system employing our load-balanced approach to query processing. In particular, the study addressed the impact of using our load-balanced query allocation algorithm. The simulation model used in the study was a closed queuing network model of a locally distributed database system. User queries were represented as sequences of query units, where each query unit runs at some site, access a single relation, and passes data to the next query unit in the query (in a pipelined fashion).

When a query is initiated by a terminal in our model, a slight variant of the load-balanced task allocation algorithm is used to decide the processing site for each of its query units. In the simulation, load-balanced query allocation was compared with other two allocation algorithms: *static allocation* (*STATIC*) and *random allocation* ($RANDOM_P$). The static allocation algorithm allocates a query unit to its predecessor's processing site if possible. If that site is not in the feasible assignment set of the query unit, i.e., if the referenced relation is unavailable there, a statically predetermined copy of the relation is selected. The random allocation algorithm operates similarly, except it randomly selects a copy of the relation instead of using a predetermined copy in the event that a local copy is unavailable. The performance metrics employed in the study were the response time of user queries, the waiting time improvement factor of each algorithm $X$ with respect to the static allocation algorithm, denoted $WIF(X, STATIC)$ and defined as the percentage of the decrease in the queuing time of the query, the throughput, and the throughput improvement factor of each algorithm $X$ with respect to the static allocation algorithm, $TIF(X, STATIC)$.

The results of the study indicated that load-balanced task allocation offers better performance than either the static or random allocation strategies, improving the response time and waiting time for queries, and even improving the overall throughput of the system in many cases [Care85b, Lu85b]. Waiting time reductions of 50% or more were typical under moderate CPU loads, and throughput improvements in some cases reached 10-30%. Figure 5.1 shows some example results from one of the tests that we conducted. The database system in this test consisted of 8 sites. The user queries (i.e., the task forces) consisted of 3 query units (or tasks) with 4 sites in their feasible assignment set on the average. Readers interested in the details of our approach to load balancing in a locally distributed database system, and/or in the details of the associated performance experiments, are encouraged to consult [Care85b, Lu85b] — the results presented here are intended only to convey some idea of what can be expected from load-balanced task allocation in terms of performance.

## 6. CONCLUSIONS

This paper has presented a new algorithm for allocating task forces to sites in a locally distributed computer system. The algorithm is novel in that it considers the current system load, and it uses load balancing as the main objective for driving the allocation procedure. The paper began by introducing a quantitative measure of the "balanceness" of a system. The load-balanced task allocation problem was then defined based on this measure and on the communications cost between the tasks to be allocated. A heuristic algorithm that carefully selects the allocation order of tasks and the sites to which tasks are allocated was then presented. A comparison between the allocations generated by the heuristic algorithm and those obtained using an exhaustive search algorithm indicated that optimal allocation plans were found in the vast majority of cases. Two enhancements of the basic algorithm were also developed and shown to further improve the optimality of the algorithm (in particular, to improve its optimality with respect to communications costs). The execution time of the algorithm was measured in several tests, and the complexity of the algorithm was considered; both indicated that the algorithm is acceptably efficient. Finally, some experimental results from a study of the use of load-balanced task allocation in a distributed database application were very briefly summarized, and the results indicated that the approach indeed offers performance gains over static task allocation schemes.

A number of opportunities exist for future work. The algorithms and ideas presented in this paper resulted from work on load balancing in locally distributed database systems [Care85b, Lu85b]. As a result, the discussions of task forces in this paper have been based on a pipelined sequence of tasks, and the evaluations of the optimality of our heuristic algorithm assumed this communications structure. While the ideas discussed here can definitely be applied to more general task force topologies, future work is needed to evaluate the optimality of the plans generated via our heuristic algorithm for other topologies. It would also be interesting to study the performance of various task force topologies using our approach to task allocation. Finally, we have not addressed the question of how load information is to be exchanged among the sites, making the integration of our ideas with an appropriate information exchange policy a problem for future work as well.

## REFERENCES

[Bokh79] S. H. Bokhari, Dual processor scheduling with dynamic reassignment, *IEEE Transactions on Software Engineering,* SE-5, 4 (July 1979), 341-349.

[Care85a] M. J. Carey, M. Livny and H. Lu, Dynamic task allocation in a distributed database system, *Proceedings of the 5th International Conference on Distributed Computing Systems,* May 1985.

[Care85b] M. J. Carey and H. Lu, *Load balancing in a locally distributed database system,* Computer Sciences Technical Report #625, University of Wisconsin-Madison, December 1985.

[Chu80] W. W. Chu, L. J. Holloway, M. T. Lan and K. Efe, Task allocation in distributed data processing, *IEEE Transactions on Computers,* C-29, 11 (November 1980), 57-69.

[Eage85] D. L. Eager, E. D. Lazowska and J. Zahorjan, A comparison of receiver-initiated and sender-initiated dynamic load sharing, *Proceedings of the ACM-SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* August 1985.

[Gyly76] V. B. Gylys and J. A. Edqards, Optimal partitioning of workload for distributed systems, *Digest of Papers, COMPCON Fall 1976,* September 1976, 353-357.

[Jone79] A. K. Jones, R. J. Chansler, I. Durham, K. Schwans and S. R. Vegdahl, StarOS: a multiprocessor operating system for the support of task forces, *Proceedings of the 7th ACM Symposium on Operating Systems Principles,* December 1979, 117-127.

[Livn82] M. Livny and M. Melman, Load balancing in homogeneous broadcast distributed systems, *Proceedings of ACM Computer Network Performance Symposium*, April 1982, 47-55.

[Livn83] M. Livny, *The study of load balancing algorithms for decentralized distributed processing systems*, Ph.D. Dissertation, Weizmann Institute of Science, August 1983.

[Lu85a] H. Lu and M. J. Carey, Some experimental results on distributed join algorithms in a local area network, *Proceedings of the 11th International Conference on Very Large Data Bases, Stockholm, August 1985.*

[Lu85b] H. Lu, "Distributed query processing with load balancing in local area networks", Ph.D Dissertation, Computer Sciences Department, University of Wisconsin-Madison, December 1985.

[Ma82] P. R. Ma, E. Y. S. Lee and M. Tsuchiya, A task allocation model for distributed computing systems, *IEEE Transactions on Computers*, **C-31**, 1, January 1982.

[Ni81] L. M. Ni and K. Abani, Nonpreemptive load balancing in a class of local area networks, *Proceedings of the 1981 Computer Networking Symposium*, December 1981, 113-118.

[Page83] T. W. J. Page, *Distributed query processing in local network databases*, Masters Thesis, University of California, Los Angeles, 1983.

[Pric79] C. C. Price, *A nonlinear multiprocessor scheduling problem*, Ph.D. Dissertation, Texas A & M University, May 1979.

[Pric84] C. C. Price and S. Krishnaprasad, Software allocation models for distributed computing systems, *Proceedings of the 4th International Conference on Distributed Computing Systems*, San Francisco, California, May 1984, 40-48.

[Rao79] G. S. Rao, H. S. Stone and T. I. Hu, Assignment of tasks in a distributed processor systems with limited memory, *IEEE Transactions on Computers*, **C-28**, 4, April 1979.

[Smit75] J. M. Smith and P. Y. T. Chang, Optimizing the performance of a relational algebra database interface, *Communications of the ACM*, **18**, 10 (October 1975), 568-579.

[Ston77a] H. S. Stone, Multiprocessor scheduling with the aid of network flow algorithms, *IEEE Transactions on Software Engineering*, **SE-3**, 1 (January 1977), 85-93.

[Ston77b] H. S. Stone, *Program assignment in three-processor systems and tricutset partitioning of graphs*, Technical Report No. ECE-CS-77-7, Department of Electrical Engineering, University of Massachusetts at Amherst, 1977.

[Ston78] H. S. Stone, Critical load factors in two processor distributed systems, *IEEE Transactions on Software Engineering*, **SE-4**, 3 (May 1978), 254-258.

[Yao79] S. B. Yao, Optimization of query evaluation algorithms, *ACM Transactions on Database Systems*, **4**, 2 (June 1979), 133-155.