Inter Process Communication Optimization in a Scalable Computing Cluster

Oren La'adan and Amnon Barak

Institute of Computer Science
The Hebrew University of Jerusalem
Jerusalem 91904, Israel

August 1998

Abstract

Scalable Computing Clusters (SCC) provide increased computing resources for the execution of high performance, demanding applications in multi-user, time-sharing environments, without the high costs of supercomputers or MPP's. In order to better utilize the multiple resources of SCC's it is necessary to develop means for cluster wide resource allocation and sharing.

This paper addresses the task clustering problem which deals with the assignment of tasks to nodes, to reduce the Inter Process Communication (IPC) overhead, while keeping the system balanced. The IPC optimization problem is a critical issue that affects the efficiency of execution of parallel programs in distributed memory machines. The main issue is to find a golden path between two conflicting objectives: keeping tasks separated, to increase parallelism, vs. clustering them, to reduce the IPC overhead.

We present a distributed, on-line, algorithm for the dynamic IPC optimization problem. Since this problem is NP-hard, the algorithm conducts a local search, to improve the assignment of tasks to nodes using a set of heuristics. This way, the algorithm continuously reassigns tasks to nodes by means of process migration, in order to reduce the "Average Slow Down" in the cluster. The allowable migrations are called "atomic operations".

The atomic operations are kept few and simple, to reduce the complexity, and "local" to enable distributed control. They can migrate a single task (single-tone), or two tasks simultaneously (dual-tone), and thus reduce the probability of reaching a local optimum. Simulations of clusters of processes with some common communication topologies has shown that the algorithms yield near-optimal assignments. These simulations has also demonstrated the usefulness and viability of the dual-tone operations as part of the atomic operations set.

Contents

Chapter 1

Introduction

Collections of computers, ranging from Scalable Clusters of (homogeneous) Computers (SCC) to Networks of (heterogeneous) Workstations (NOW) are becoming increasingly popular platforms for executing demanding applications, high performance and interactive computing [3]. Unlike Massively Parallel Processors (MPP's), which in most cases allow a single user per partition and a single process on each node, SCC's and NOW's are geared for time-sharing environments and simultaneous use by many users. In order to make SCC systems as easy to program, manage and use as an SMP, it is necessary to develop means for global (cluster-wide) resource allocation and sharing. These means can improve the overall performance by responding at run-time to uneven use of the resources, in order to better match between the resource requirements and availability. This is necessary for performance scalability, for efficient utilization of the resources and to support multiple users, with unpredictable requirements in a time-sharing parallel execution environment.

The general problem of multiple resource sharing in a parallel environment, and the development of effective techniques for resource allocation have gained much attention recently, mainly in the context of CPU load. Load balancing and load sharing algorithms for SCC's were extensively researched [5, 16, 26], and several schemes were implemented [7, 9, 30]. Other common resources which have been studied in this context include memory allocation [4, 8], and processor allocation (partitioning), mainly in MPP's [25]. Considerable research was also made on network and communication related resources, in particular off-line algorithms for graph partitioning and scheduling, and on-line algorithms for routing of virtual circuits, congestion and admission control [23].

The problem of resource allocation varies from simplified models, e.g., with static resource requirements, to more complex models, e.g., with dynamic parameters. Accordingly, the main parameters are the way resources are allocated, static vs. dynamic; the system's control, centralized vs. distributed; availability of knowledge about the system's resources, partial vs. complete; and the operation mode, static, dynamic or adaptive. Algorithms were developed for many of the above cases. These algorithms range from static, off-line optimizations or approximations, to dynamic on-line and competitive algorithms. In particular we note an ongoing work which formalizes a general framework for on-line allocation of different, multiple resources using the opportunity cost approach, which uses a common cost criteria [2, 6].

This paper addresses the problem of task allocation to nodes (a.k.a task clustering) to reduce the Inter Process Communication (IPC) overhead, while keeping the overall system load-balanced. The IPC overhead results from assigning communicating processes to different nodes. Should these processes be executed in the same node, then they would benefit from fast, shared-memory IPC. However, such a placement may overload that node, thus slowing the execution of these processes. The tradeoff is between load-balancing, which tends to disperse the processes (evenly) across many nodes, and IPC, with an opposite tendency, to group together communicating processes in common nodes. If the applications are modeled by a tasks graph denoting communications and dependencies among the tasks, then the main issue is how to partition the graph into sets of tasks and assign these sets to the nodes.

Since even simplified formulations of the task clustering problem are NP-hard [14, 22], many sub-optimal solutions have been proposed. These solutions have to sacrifice optimality for the sake of efficiency, either by using an approximation or by employing some heuristics. Common methodologies used for static assignment schemes can generally be divided to two general approaches: precedence scheduling and graph partitioning. In the precedence scheduling the application is usually modeled by a directed acyclic graph (DAG), and the goal of the assignment is to minimize the make-span (i.e. the total completion time). Several heuristics can be found in the literature for this model, that use various techniques such as critical path analysis [15], priority-based list scheduling [24], graph decomposition [20] and greedy strategies [21]. Graph partitioning is a more generic approach that does not limit the application's graph. The goal is to partition the vertices of the graph into 2 (or generally k) subsets of equal size, such that the bisection width (number of edges between the different sets) is minimized. In the context of the IPC optimization problem, this goal translates to minimizing the IPC overhead under load balancing constraints. Heuristics used for this problem implement local search strategies by considering perturbations of the current solution. These include greedy clustering heuristics [10, 18], Simulated Annealing [19] and Genetic algorithms [31]. In addition to their generality, these techniques can be used repeatedly and thus provide adaptive (on-line) assignment policies. Another approach is based on concepts drawn from Microeconomics to the resource allocation problem [13, 27]. In this approach both the resources and the applications are modeled as independent economic agents that attempt to selfishly optimize their satisfaction (utility). The main benefits are reduced complexity and flexibility through the choice of the utility. Unlike the previous approaches, these algorithms are intrinsically on-line and distributed.

In this paper we present distributed, on-line algorithms for the dynamic IPC optimization problem. Our study is aimed for a dynamic environment, where tasks arrive and leave the system in an unpredictable manner. Our algorithms are executed in a decentralized manner and they assume partial knowledge about the requirements and use of the global system resources. Our goal is to minimize the average slowdown of the tasks in the system. The approach employs a local (in each node) search for a better assignment of tasks, to reduce the inter-node communications overhead. This is done by a sequence of primitive (*atomic*) operations which reassign tasks to nodes to gradually reduce this overhead. Our approach benefits from the local search advantages: repeatedly improve the system's performance by migrating processes around. The (heuristics) rules for the migration policy are kept simple in order to reduce the complexity of the solution, yet powerful, to decrease the probability of reaching a local optimum. The rules are applied locally and are managed in a decentralized manner.

The performance of our algorithms is evaluated by simulations of several commonly used communication topologies between tasks. These simulations indicate that the algorithms improve the overall system's performance. In most of the test cases an optimal assignment was obtained. This indicates that our algorithms could improve the performance of any operating system that supports dynamic process migration, such as MOSIX [5, 7].

The paper is organized as follows: the next chapter begins with an intuitive overview followed by a formal description of the IPC optimization problem and its complexity. Chapter 3 presents the system model of the problem. This model provides a framework for the investigation of the IPC optimization problem. Chapter 4 presents the details of the optimization algorithm, the motivation for its design and the *atomic operations* underlying it. An analytical formulation and evaluation of the *atomic operations* is given in Chapter 5, demonstrating their role. Chapter 6 presents simulations results of the performance of the algorithm and their analysis. The conclusions and suggestions for future work are given in Chapter 7.

Chapter 2

Statement of the Problem

The IPC optimization problem in a SCC deals with how to assign processes to nodes in order to optimize the inter-process communication overhead while keeping the overall system load-balanced. The communication overhead results from assigning communicating processes to different nodes. Should these processes be executed in the same node, then they would benefit from fast, shared-memory IPC. However, such a placement may overload that node, thus slowing the execution of these processes. The tradeoff is between load-balancing, which tends to disperse the processes across the nodes and IPC, with an opposite tendency, to group together communicating processes in common nodes. We note that throughout this paper the terms "job" "process" and "task" are used interchangeably.

Formally, the IPC optimization problem in a SCC can be defined as follows: *given a set of processes with their CPU and communication requirements, place the processes to the nodes such that the overall performance is maximized.* This problem has many variants, differing by (among others) the underlying system model, the performance metric, on-line nature and so on. Many of these variants have been shown to be NP-complete or NP-hard. In this paper, we focus on the on-line variant, in a multi-user, time-sharing environment (such as a SCC), and we seek to minimize the response time.

Our computing cluster consists of a set of homogeneous nodes connected by a LAN with a full graph topology. At any given time, the cluster has a set of executing processes, which were previously assigned to the nodes according to some policy. We attempt to improve the overall performance by migrating processes from one node to another, to balance the load and at the same time reduce, as much as possible, the IPC overhead. This is achieved by an on-line, heuristic policy which continuously uses the process execution profiles in order to find an improved placement among the nodes.

Simultaneous optimization of both the load and the communication overhead is complicated since these two problems are contradictory by their nature. On one hand, load balancing is achieved by maximizing the distribution of the processes among the nodes, to utilize many CPU's. At the same time, the IPC overhead can be reduced if sets of communicating processes are assigned to common nodes, to benefit from fast, shared-memory communication. As such, load balancing acts as a repulsive force between the processes, while reducing the IPC overhead can be viewed as an attracting force between groups of processes [11]. As a result, there are scenarios where load balancing should prevail, e.g. when there is a low communication volume, while in other cases communication optimization should be the dominant consideration. The following example further explains this last point.

Example:

Consider a program consisting of two processes performing computation and communication. The two processes can either be placed in two different nodes or in the same node, as seen in Figure 2.1. In the first case, shown in Figure 2.1(a), the processes execute in parallel, while the communication is performed over the network. In the second case, shown in Figure 2.1(b), the computation time doubles, since the processes share the same CPU, but the communication via shared memory is much faster. The difference between shared memory and network communication is over an order of magnitude, due to the performance gap between current CPU speeds and the network bandwidth delivered to the application. Therefore, we assume that the communication via shared memory is negligible. Under this assumption, the best assignment policy in this case depends on the ratio between the computation and the communication times. If this ratio is greater than 1, then it is better to assign the processes to two nodes, while if it is less than 1, they should be placed together in one node, to maximize their throughput.

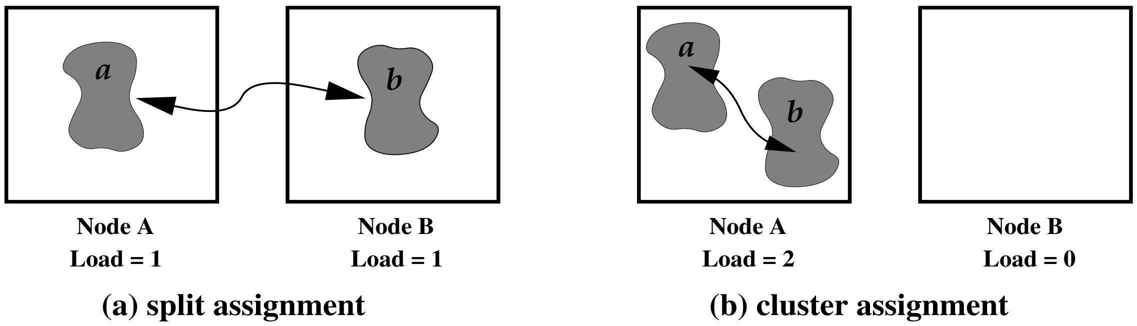

Figure 2.1: *Possible assignments of two processes to nodes*

The above example, though extremely simple, illustrates the essence of the IPC optimization problem: the intrinsic competition between load balancing and task clustering. The conclusion is that an approach involving independent algorithms for load balancing and IPC optimization is not sufficient. Rather, one must employ a *combined approach* for load balancing and IPC optimization.

Chapter 3

The System Model

This chapter presents the system model that provides a framework for our study of the IPC optimization problem. This system model consists of a cluster of loosely-coupled computers (nodes) that are connected by a Local Area Network (LAN), where each node is capable of time-sharing multiple processes. The nodes are autonomous and they support message passing communication among the processes. The system is dynamic with respect to resources requirements and availability.

3.1 The network model

Consider a cluster of N loosely-coupled nodes $P_i, \ldots, P_N$, that are connected by a LAN that supports a point-to-point communication. We assume that the network is homogeneous, with a topology of a fully connected graph. Denote the network latency by $\mathsf{T_L}$ and the network bandwidth by BW. Assume a linear communication time, such that sending a message of size M, costs $\mathsf{T_L} + \mathsf{M} \cdot \mathsf{T_W}$, where $\mathsf{T_W} = 1/\mathsf{BW}$.

Throughout this paper we assume an asynchronous communication, where *send* and *receive* operations are not necessarily synchronized. Sending a message requires $\mathsf{T_L} + \mathsf{M} \cdot \mathsf{T_W}$ time units, without the need to wait for the receiver to begin receiving it. This is a reasonable assumption for a system with enough buffer space to hold large messages that are not consumed. Receiving is asynchronous in the sense that a message may arrive at any time, and then it activates some kind of a message handler in the receiving process. It may happen that in the case of an explicit receive operation there is no data in the buffer. We assume that the requesting process will do some other (probably CPU related) work instead, and set a handler to be called on this event. Alternatively the process can *busy wait* by polling the buffer for the message.

Lastly, we define how communication is performed in terms of CPU usage. Every communication operation involves data transfer between the node's memory and the network controller. We assume that this tasks is solely accomplished by the CPU. This means that communication consumes CPU cycles, and a communicating process competes for the CPU with the other processes on the same node. This assumption is appropriate for the *programmed I/O* communication concept. In *programmed I/O* the network controller and the CPU share a memory segment, and the CPU directly copies data to/from that segment. Another common method is *DMA* data transfer, in which special hardware is used to transfer data by "stealing" bus cycles. Since *DMA* require minimal CPU cycles, communications operations may be overlapped with CPU usage.

The above assumptions may be relaxed in several ways, leading to a more complex model. This opens a wide spectrum of areas for future research. In particular, our model employs two strong assumptions about the network: *programmed I/O* and asynchronous communications. The former determines the possible states with regard to CPU usage: the process *at any time* competes for CPU cycles (i.e. ready-to-run). This simplifies the load calculation (restricting it to an integer value) and relieves us from the burden of considering the exact scheduling scheme within each node. The second assumption allows us to disregard blocking of a process (pausing) while waiting for the other party to synchronize.

Extensions to this model may include changing the network topology, considering heterogeneous network links or a different model of the message transfer time. We consider these as "cosmetics" in the sense that they do not affect our optimization algorithm, but only the cost functions.

3.2 The execution model

Consider a sequence of jobs (processes), $J_1, J_2, \ldots, J_N$, that arrive at arbitrary times and are assigned to the nodes of an SCC. The set of executing processes varies dynamically with each arrival of a new process or the completion of another. We do not make any assumptions regarding the initial placement of newly created processes. Such initial placement may be based on pure load balancing considerations or any other heuristic.

At any given time an executing process may be in one of two states: *computing* or *communicating*. Hereinafter we shall use the following terminology: a process is *doing CPU* if it is computing, and is *doing IPC* if it is communicating. Computation always consumes CPU cycles. However, the communication times depends on the location of the participating processes. If these processes run on different nodes, then the communication cost is linear with the message size. For our study, we assume that each process needs that much of CPU cycles to accomplish its communications. On the other hand, if two communicating processes reside on the same node, then they benefit from fast, shared-memory IPC. For simplicity we assume that such communication completes immediately, without incurring any overhead.

In line with the above model, a process consumes CPU cycles unless it is doing IPC via shared memory. Since this kind of IPC is immediate, all the processes are competing for the

CPU at all times. We also assume fair scheduling in the nodes. Consequently, we define the *load* of a node as the number of processes executing on that node. The *load* represents the number of time slices a process has to wait between two consecutive time slices that it is given.

3.3 Runtime support

We assume that the operating system provides each node with information about the relevant resource utilization, e.g., the load, of some or all of the nodes in the system. Obviously, this information is required in order to provide a basis for decisions of the optimization algorithm. Examples of algorithms for information dissemination and information exchange are described in [1]. Similar algorithms are implemented in MOSIX [5].

In order to use a dynamic optimization algorithm which reassigns processes to nodes, the system must support process mobility. Such a tool allows a migration of a process from its current node to another node. This can either be preemptively, i.e. at any given time during its execution, or by an explicit request by the user or the process. In this paper we ignore the overhead incurred by process migrations.

During the life cycle of a process, the system keeps track of its execution profile. This essential information is continuously provided to the optimization algorithm. For each process the following attributes are collected and recorded:

- Accumulated CPU time
- Number of IPC operations to/from any other process
- Accumulated communication volume with any other process

3.4 The performance metric

Our goal is to assign processes to nodes to utilize the overall processing power of the system efficiently. This requires a definition of the objective function for the optimization, by means of a performance metric. Examples of common performance metrics are: *task throughput*, *average task response time* and *total elapsed time* [12]. We introduce a new performance metric, termed *Average Slow Down* (ASD), which is closely related to the *response time* metric.

We define the *slowdown* (SD) factor of a process as the ratio between its actual execution time and its optimal execution time (on an ideal system). This measure reflects the overhead incurred on the process by the existence of other processes and the network. The motivation is that reducing the slowdown of a process, improves its performance (and response time).

Hence, a decrease in the *average slowdown* over all processes, improves the performance of the system as a whole. Using this measure, the objective function for the optimization is to minimize the *average slowdown*.

Intuitively, minimizing the *slowdown* factors of the processes implies efficient overall utilization of the system. For instance, one can reduce the *slowdown* factor of all of the processes on a loaded node, by transferring one process to an idle node. Indeed, this yields better CPU utilization. Normally, however, improving the *average slowdown* of some processes results in a performance degradation of others, since they all compete for common resources of the system.

One point that requires further discussion is the *fairness* of the suggested performance metric. An assignment is said to be *fair* if the variance of the *slowdown* factors of the processes is small. An extremely unfair assignment may lead to starvation of some process with respect to a certain resource. For example, a pair of processes may exhibit *slowdown* factors of 3 and 3 in one assignment, or 1 and 5 in another. The average is the same in both cases, yet the latter is *unfair*. The fairness requirement is not inherent to our objective function. Rather, in some cases the criterion of minimal *average slowdown* implies an extremely unfair assignments.

It is interesting to show how the *average slowdown* metric behaves in the case of the known load-balancing problem. We show that the requirement of minimal *average slowdown* implies a fair, load balanced assignment. Such an assignment can be achieved, for example, with the load-matching algorithm described in [5].

Consider a system with P nodes $(P_0, P_1, ... P_P)$, that executes N CPU-bound processes. Let N_i be the number of processes that are executed on node P_i. Then we have:

$$\sum_{i=1}^{P} N_i = N \ .$$

A process executing on node P_i shares the CPU with $N_i - 1$ other processes, therefore experiencing *slowdown* of N_i. Consequently, the total *slowdown* (SD) factors on that node is:

$$SD(P_i) = N_i \cdot N_i = N_i^2 \ ,$$

and the average slowdown is:

$$ASD = 1/N \sum_{i=1}^{P} N_i^2 \ .$$

Claim 1 *For N a multiple of P, ASD reaches its minimal value iff $N_i = N/P$.*

Proof: First, observe that all the numbers are integers. If the claim doesn't hold, then there exist i, j and $\delta_i, \delta_j > 0$, such that:

$$N_i = N/P - \delta_i \ , \quad \text{and} \quad N_j = N/P + \delta_j \ .$$

Therefore we can move one process from P_j to P_i. Consequently, the value of the ASD is changed as follows:

$$\begin{aligned} ASD_{new} - ASD_{old} &= \left[(N/P + \delta_j - 1)^2 + (N/P - \delta_i + 1)^2\right] - \left[(N/P + \delta_j)^2 + (N/P - \delta_i)^2\right] \\ &= 1 - 2\delta_i + 1 - 2\delta_j < 0 \ . \end{aligned}$$

∎

Note that if N not a multiple of P, then the optimal solution must satisfy: $N_i = \lceil N/P \rceil$, or $N_i = \lfloor N/P \rfloor$, for every i. In both cases, this is obviously a fair, load-balanced assignment.

3.5 Slow-Down evaluation

This section gives quantitative definitions for the *slowdown factor* and the *Average Slow-Down (ASD)*, based on the system model that was presented in Section 3.2 and Section 3.1.

A process in the system is characterized by an execution profile. We assume that the system has no *a-priori* knowledge of this profile. Under this uncertainty, no system can guaranty that a certain operation will improve the performance. A simple heuristics is to assume that the process exhibits a "past repeat" execution pattern; that is, the process is expected to continue its execution with the same profile for a period of time that is at least equal to the time it has executed so far. It follows that the profile is not completely random, but rather of some regular pattern.

Let $\vec{RC_i} = \langle T_i, C_{i,1}, \ldots, C_{i,N} \rangle$ be a vector of length $N+1$ describing the execution profile of a process J_i. Then $\vec{RC_i}$ specifies T_i, the amount of CPU cycles required for computation, and $C_{i,j}$, the amount of CPU cycles needed for LAN communication with *any* of the other processes. We note that the values of $C_{i,j}$ are calculated according to the network parameters and the communication volume of the process.

The *slowdown* factor of a process J_i, with respect to $\vec{RC_i}$, is the ratio between the expected completion time and the completion time on an "ideal" reference system. For this purpose, the ideal reference system is a node with an infinite communication bandwidth, e.g., an SMP [28] with a sufficiently large number of processors, so that each processor executes a single process. In this architecture, each processor exhibit a load of 1 and there is no communication overhead due to the use of shared memory.

Consider the general case of a communicating process, running on a node with a load L. Let T_i be the number of CPU cycles required by the process for the computational work. Every communicating *peer* (J_j) that is mapped to a different node contributes $C_{i,j}$ CPU cycles. Those that are mapped to the same node do not incur any CPU overhead. The total requirement for CPU cycles due to communications is thereby determined by the current mapping of the *peers*, and is given by:

$$T_{ipc} = \sum_k C_{i,k} \cdot \overline{\delta}_{i,k} \ ,$$

where

$$\overline{\delta}_{i,k} = \begin{cases} 1 & \text{if } J_i \text{ and } J_k \text{ are mapped to the same node ,} \\ 0 & \text{otherwise .} \end{cases}$$

Since this process shares the CPU with $L-1$ other processes, the actual execution will be L times slower. Hence its expected execution time is:

$$T_{real} = L \cdot (T_{cpu} + T_{ipc}) \ .$$

Turning to the reference system, the process is using the CPU exclusively, and there is no communication overhead, thus $T_{ref} = T_i$.

Let $\widetilde{T}_{cpu}$ and $\widetilde{T}_{ipc}$ be the normalized computation and communication time respectively, obtained by dividing T_{real} by T_{ref}. Then the *slowdown* factor of this process, defined as the ratio between T_{real} and T_{ref}, satisfies:

$$SD(i) = \frac{T_{real}}{T_{ref}} = L \cdot \frac{T_{cpu} + T_{ipc}}{T_{ref}} = L \cdot (\widetilde{T}_{cpu} + \widetilde{T}_{ipc}) = L \cdot \left(1 + \frac{\sum_k C_{i,k} \cdot \overline{\delta}_{i,k}}{T_i}\right) \ .$$

Substituting a profile vector of a CPU-bound process ($\vec{RC} = \langle T_i, 0, \ldots, 0 \rangle$) into this formula gives the *slowdown* factor of L. Note that this is in line with the observation that such a process is executing L times slower than on an otherwise idle node.

It is interesting to point out that the *slowdown* factor reaches infinity when $T_i \to 0$, i.e., when the process is only communicating, and its peer is mapped to a different node. Indeed, the execution cost of such a process is zero if it is mapped to the same node as its peer, while it has some constant value otherwise. Hence the ratio is infinitely large.

Chapter 4

Optimization by Atomic Operations

In this chapter we develop an approximation algorithm for the IPC optimization problem. We begin by presenting the motivation for our algorithm, followed by a discussion of the basic building blocks for its operation. We then describe the algorithm and possible extensions to it.

4.1 Motivation

The complexity of the IPC optimization, being an NP-hard problem, is of major concern when developing the algorithm. It suggests that one should seek an approximation algorithm to the problem. Such an algorithm will have to sacrifice optimality for the sake of efficiency. The framework in which the algorithm is intended to operate is another important parameter. The algorithm should accept on-line decisions, with zero knowledge about the future and partial knowledge about the system. It should operate in a distributed manner, preferably with decentralized control to support scalable configurations.

Many of the algorithms previously suggested, are designed to deal with the off-line variant of the problem, where the communication patterns and computational demands are known a priory. See [21] and [22] for scheduling examples, and [18] for a graph partitioning method. Other approaches include local search algorithms, which require complete knowledge about the state of the system and centralized control. To the best of our knowledge, currently there are no on-line algorithms which deal with this problem in a distributed manner. Instead, combinatorial optimization algorithms were developed using a local search [17], hill-climbing and simulated annealing [19]. The last two use different heuristics to improve a given solution. The scheme continues until convergence to a local optimum. These heuristics do not meet our constraints, as they are centralized, and incur significant overhead, e.g., the simulated annealing algorithm, when executed frequently for an on-line dynamic system.

Our approach benefits from the local search advantages: repeatedly improve the system's performance by migrating processes around. The rules for the migration policy (heuristics) are simple, to reduce the complexity of the solution, yet powerful, to reduce the probability of reaching a local optimum. The key idea is to allow a limited number of simple operations, with the following properties:

- *Quality* - each operation improves the current setting of the processes
- *Locality* - each operation affects a small number of nodes
- *Autonomy* - decentralized control, executed independently by each node
- *Simplicity* - simple rules that do not incur large overhead

Below we describe the set of allowable operations and the motivation for inclusion of each of its elements.

4.2 Optimization by atomic operations

In this section we present a set of operations, called *atomic operations* for process assignment and reassignments (migration), to reduce the IPC overhead. The first set of such operations involve only one process. We call this class *single-tone* atomic operations. An example of single-tone operation was presented in Chapter 2. In this example, it was shown that a simple consideration based on the CPU/IPC ratio of the process execution profile was sufficient to determine which of possible 2 alternative assignments was better. More formally, the *single-tone* decision class includes all the migrations that involve a single process. For a migrated process i, denote the origin node and the destination node S_i and D_i respectively. Then this class includes all two-tuples of the form $\langle S_i, D_i \rangle$. If $S_i = D_i$ then there is no migration, implying that $S_i \neq D_i$ is the only operation in this class. This operation shall be called: *single migration.*

As we shall see, in the general case, *single-tone* considerations are not sufficient and more complex operations, involving migration of more than one process are required. For example, consider the case of two identical pairs of communicating processes initially assigned to two nodes, as shown in Figure 4.1. If the assignment is such that each communicating pair is split between the two nodes, as shown in Figure 4.1(a), then this assignment may result in a large communication overhead over the other alternative, which places each communicating pair together in one node, as shown in Figure 4.1(b). In this last case the processes benefit from fast local communication due to the use of shared memory communication. To sum, the second assignment is clearly better than the first.

Suppose now that in the above example, the initial assignment was as in Figure 4.1(a). Then, there is no incentive for any single process to migrate to the other node, since this may cause the system to become imbalance. On the other hand, if two processes exchange their

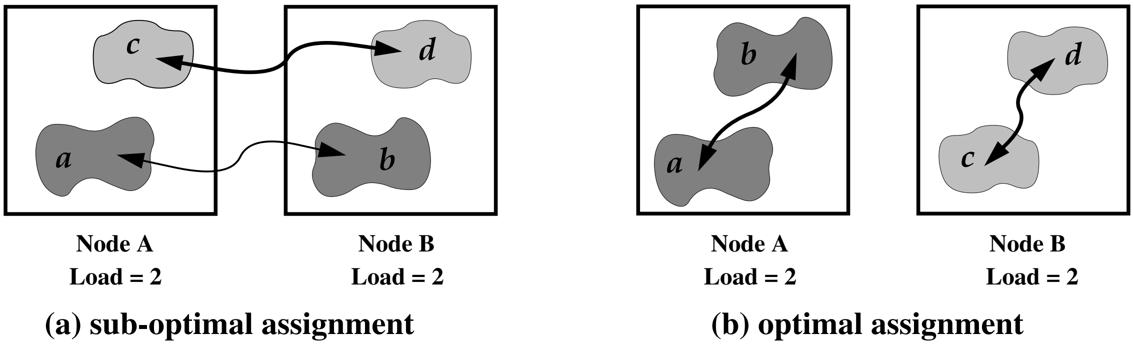

Figure 4.1: *Exchanging processes C and B eliminates the communication overhead.*

locations simultaneously then the communication overhead will diminish, thus improving the overall performance, while still preserving the balanced load among the 2 nodes. We call this operation *processes exchange*. Note that it involves two processes, and requires both nodes to coordinate their migrations. We define this class of decisions: *dual-tone* operations.

Formally, the *dual-tone* decision class includes all migrations that involve two processes. It contains all four-tuples of the form $\langle\{S_i, D_i\}, \{S_j, D_j\}\rangle$, with $i \neq j$, $S_i \neq D_i$ and $S_j \neq D_j$. The operations in this class can be further sub classified into 6 different types, based on combinations of the originating and destination nodes. Below are the definitions of the different types:

1. *Processes exchange:* $S_i = D_j$ and $S_j = D_i$
   Two processes *exchange* their positions.

2. *Double migration:* $S_i = S_j$ and $D_j = D_i$
   Two processes migrate together from one node to the same target node.

3. *Processes merge:* $S_i \neq S_j$ and $D_j = D_i$
   Two processes from two different nodes migrate to a common third node.

4. *Processes split:* $S_i = S_j$ and $D_j \neq D_i$
   Two processes from the same node migrate to two different nodes.

5. *Processes shift:* $S_i \neq S_j$, $S_i = D_j$ and $S_j \neq D_i$
   One process migrates, and another (from a different node), takes its place.

6. *Mutual migration:* $S_i \neq S_j$, $D_j \neq D_i$, $S_i \neq D_j$ and $S_j \neq D_i$
   Two processes from two different nodes migrate to two other, different nodes.

We note that the above operations can also be characterized by the number of nodes involved. *Double migration* and *processes exchange* involve 2 nodes. *Processes split*, *processes merge* and *processes shift* involved 3 nodes. *Mutual migration* requires cooperation between four nodes. In the following sections we elaborate on each of the dual-tone operations and we demonstrate their usefulness.

4.2.1 Processes exchange

Process exchange is needed whenever it is necessary to simultaneously swap 2 processes between 2 nodes. This case was explained earlier in this chapter and in Figure 4.1.

4.2.2 Double migration

In the *double migration* operation, two communicating processes that are executing on the same node are migrated together to another (usually, less loaded) node. Figure 4.2(a) shows a typical case in which this operation is applied. The figure shows 2 communicating pairs in Node A. In this case, if a single process moves to the idle Node B, it will benefit from fast computation but at the same time suffer from higher IPC overhead. Moreover, its peer will also be affected by the additional IPC overhead. Applying a *double migration* allows this pair to execute on a less loaded node without incurring additional IPC overhead. The resulting placement is shown in Figure 4.2(b). Note that the remaining processes on the sending machine also have better performance, due to a reduced load there.

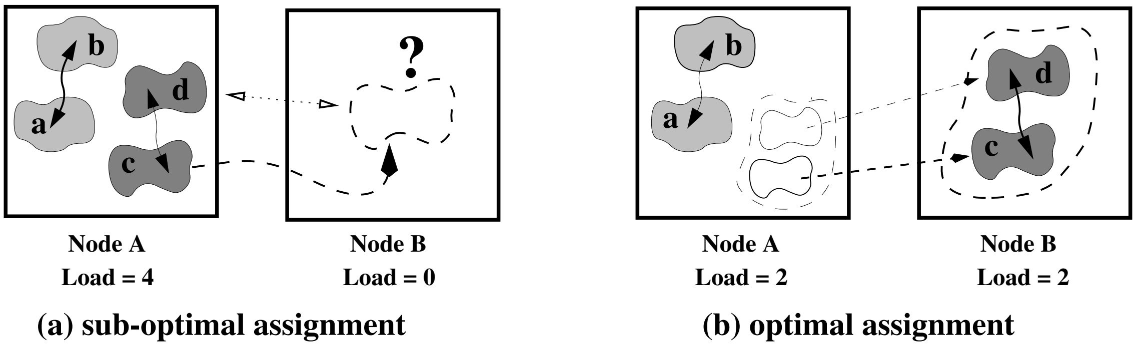

Figure 4.2: *Double process migration: a single process would not migrate (a), but 2 communicating processes can migrate together to reduce the load (b).*

4.2.3 Processes merge

Processes merge, see Figure 4.3, is used to migrate a pair of communicating processes from two different nodes to a third node. This can be beneficial when a separated communicating pair would abstain a single migration or a process-exchange due to load/IPC considerations. For example, consider the setting shown in Figure 4.3(a). Both processes a and b perform light local IPC in addition to their mutual, higher volume IPC. Without a mutual migration, both process a and b are bound to stay in their current nodes. If process a is migrated to Node B, to join process b in order to reduce their mutual IPC overhead, than the load of Node B is increase, thus slowing down the execution of both process a, b and d, as well as incurring additional overhead due to IPC with process c. Another alternative is that process a is migrated to Node C, to benefit from a reduced load, but it could then exhibit an increased IPC overhead. Migrating both processes a and b to Node C, see Figure 4.3(b), combines both advantages: reduced load for all the processes and possibly decrease in the IPC overhead over the original setting.

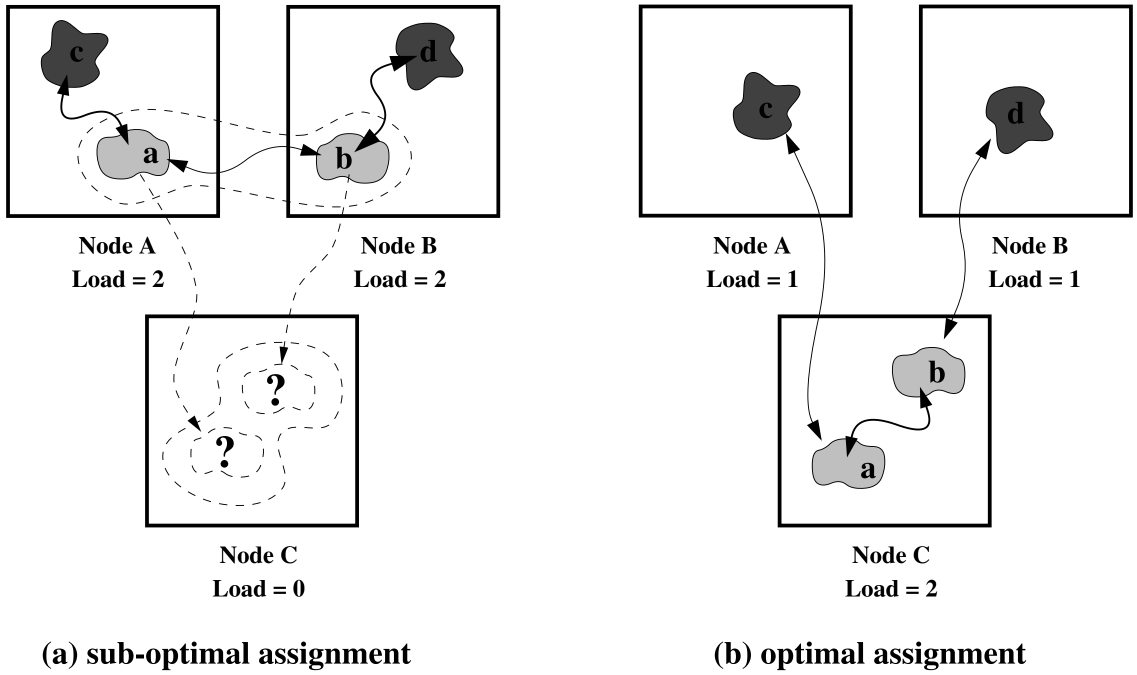

Figure 4.3: *Processes merge: a single process would not migrate (a), but 2 heavily communicating processes could migrate together to cancel their mutual IPC overhead (b).*

4.2.4 Processes split

The *processes split* operation can be viewed as the reverse operation of *processes merge*: two communicating processes that are mapped to the same node, migrate to two different, other nodes. This operation can be advantageous, for instance, if the system state is like that shown in Figure 4.3(b). Assume a set of communication and computation parameters which justify a mutual migration of process a and process b to the state shown in Figure 4.3(a). Any single migration, of either process a or process b to Node A or Node B respectively would just increase the overall overhead. Also, migrating of process c to Node C would not be possible due to load considerations. In this case *processes split* is advantageous since it attempts to compensate for the increase in the IPC overhead of both processes a and b (due the split) by reducing the IPC overhead with processes c and d respectively. A migration out of Node C may be also motivated by a larger load there.

4.2.5 Processes shift

The idea behind the *processes shift* operation is similar to that of *process exchange*: migrate an IPC process to its peer node, and transfer some other process to another node, to prevent an increase of the load. In this case, the "other node" is a third node, to which the "other process" is more attracted due to IPC considerations. Such a scenario is shown in Figure 4.4(a). Even though process c will not voluntarily migrate to node C, it should do so for the benefit of processes a and b, thereby improving the overall system performance, see Figure 4.4(b).

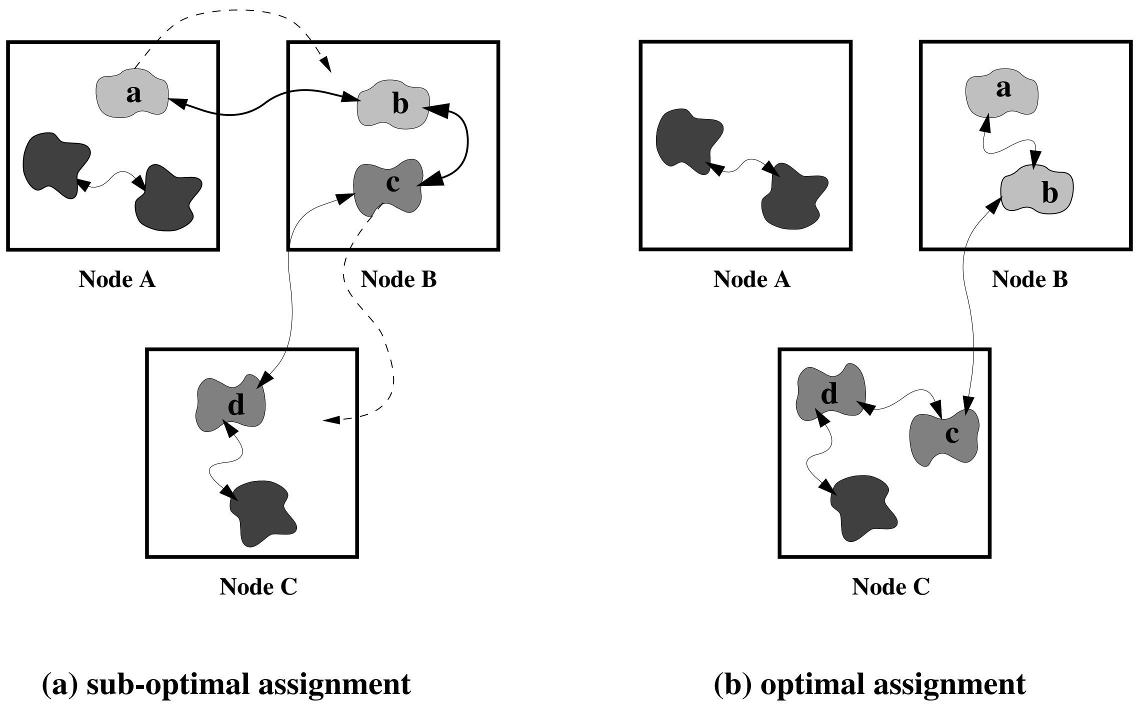

Figure 4.4: *Processes shift: a single process would not migrate (a), but shifting another process away results in a better setting (b).*

4.2.6 Mutual migration

The *mutual migration* operation should be treated differently from the other *dual-tone* operations. As opposed to the other operations, it is redundant in the sense that it does not offer additional functionality. For every tuple $OP_M = \langle\{S_i, D_i\}, \{S_j, D_j\}\rangle$ that constitutes a *mutual migration* operation, there is a sequence of two *single migrations*, namely $OP_{S,1} = \langle S_i, D_i\rangle$, and $OP_{S,2} = \langle S_j, D_j\rangle$, that would achieve the same result. Moreover, if OP_M reduces the ASD of the system, then either $OP_{S,1}$ or $OP_{S,2}$ (or both) reduce the ASD . In fact, if only one of these operations improves (w.l.g $OP_{S,1}$), then we should better avoid OP_M, and only apply $OP_{S,1}$ instead. If both operations are beneficial, then applying them sequentially would attain the same effect. Note however, that there may be an improved *single migration* operation without any improvement by a matching *mutual migration*.

The reason for this relationships is that our communications model implies locality of operations. Since the communication is asynchronous, a migration affects only those processes which are currently mapped to either of the nodes participating in the migration. We conclude that *mutual migration* is a degenerate case of multiple *single migrations*. Consequently we exclude it from the set of atomic operations used by the algorithm.

4.3 Extending the atomic operations set

The previous sections introduced the set of atomic operations, containing *single-tone* and *dual-tone* considerations. These operations constitute the building blocks of the optimization algorithm, as described below. Unfortunately, these operations are not sufficient to obtain an optimal, robust solution. For example, consider a system with two nodes and two communicating processes with a ring topology of size 3 that are initially assigned to one node, see Figure 4.5(a). Evidently, the optimal assignment in this case is shown in Figure 4.5(b). However if the communication is massive enough, both *single-tone* and *dual-tone* migrations are not sufficient to transfer the system from the initial setting to the optimal setting.

The example above can be optimized by successively applying two atomic operations: a *double migration* and a *a single migration*. This is a second-order consideration, in the sense that two operations must be decided upon *in advance* and *together* while considering migrations. Migrating the 3 processes together may also be viewed as a *triple migration* operation belonging to the *tri-tone* class, similar to the definition of *double migration*.

This exemplifies that the selected set of atomic operations is incomplete. It can be naturally extended to include *tri-tone*, *quad-tone* and, in general, *N-tone* operations. Note that adding more complex operations is equivalent to allowing *N-order* considerations, i.e. using the basic set of atomic operations as defined with look-ahead decisions enabled. In practice it is a complicated, as well as costly procedure to implement. This is more so, in a distributed system, which has only partial knowledge about the state of the processes in remote nodes. Furthermore, if the algorithm were to examine *N-order* operations for every

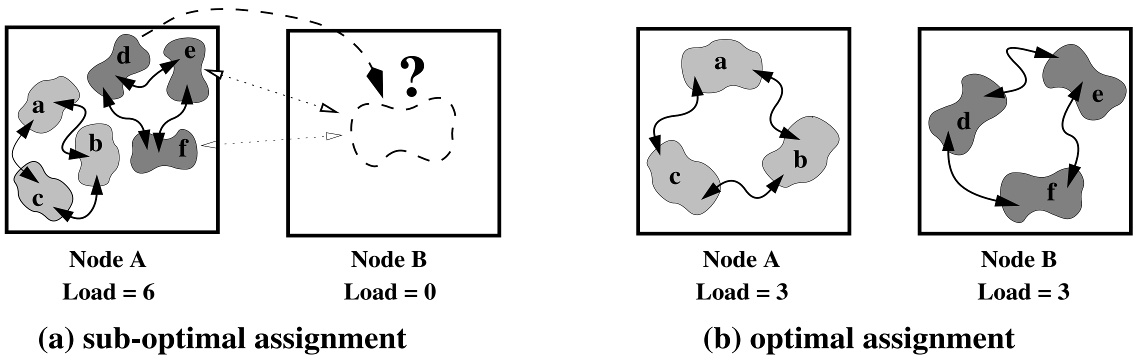

Figure 4.5: *Neither a single migration nor a double migration would be adequate (a), but 3 communicating processes can migrate together (b).*

N, then it would actually perform a global search, instead of a local search, resulting in an exponential complexity.

4.4 The optimization algorithm

The main goal of the algorithm is to optimize the IPC overhead while balancing the loads of the participating processors. This is accomplished by continuous attempts to reduce the average slowdown of the system. The main tool of the algorithm is the set of atomic operations presented above. These operations are applied by the algorithm to transfer the system from its current process state to an improved one.

The algorithm continuously records the execution profile of the running processes. For each process, it maintains the amount of CPU cycles consumed by the process and the amount of IPC performed with every peer. Also, the load of the processor is calculated, and this information is disseminated to other nodes by an appropriate algorithm, e.g. as in MOSIX [7], such that each node has a (partial) load information about other nodes. Each node executes the same algorithm locally, based on the recorded execution profiles of the local processes and on the available load information of other processors.

A migration consideration is triggered either periodically, or when a change in the system state is detected. For instance, this could be a change in the load of some nodes, or when a new node join the cluster. Once triggered, the algorithm scans all the processes on the node seeking for a candidate for migration, calculating the expected outcome of each of the atomic operations. If an operation that reduces the average slowdown is found, it is applied. In case of multiple choices, the operation which implies the largest performance improvement is chosen.

The algorithm is symmetrical in the sense that all of the nodes execute the same algorithm. It is also distributed, in the sense that the reduction of the average slowdown is performed independently by any pair (or triplet) of nodes. Other properties of the algorithm are that it responds to changes in the loads of the nodes, the runtime characteristics of the processes, and the number of nodes.

We note that the proposed algorithm is an approximation for the optimization problem, and the reason for this is two-fold; the use of the hill-climbing optimization paradigm and the limited set of atomic operations. As we shall show, for many cases the algorithm is indeed powerful and useful, as demonstrated by our examples in Chapter 6.

4.5 Implementation notes

In this section we examine the complexity of the implementation of using the atomic operations. In a real system the execution of the optimization algorithm itself incurs non-negligible overhead. Every consideration made, e.g., testing for *double migration*, consumes both CPU cycles and network bandwidth. It is therefore beneficial to analyze the complexity (or cost) of the atomic operations. Based on this analysis the runtime overhead can be reduced by considering expensive operations (in terms of overhead) less frequently than inexpensive ones.

We characterize cost of the atomic operations by the number of processes that are migrated and the number of nodes that participate. By all means the simplest operation is the *single migration*, which involves a single process and two nodes. It consists of a scan through the list of known nodes (in the origin node) and considering migration of each of the processes in that node.

The next level of complexity includes *double migration* and *processes exchange*, involving two processes and two nodes. It is possible to further distinguish between them by considering the type of interaction which is required between the two participating nodes. The first case is handled similarly to the *single migration*, except that the origin node should consider every pair of processes. The second case is more complicated, since it requires that the two nodes interact. For example, one node initiates connection and sends the other node data about its processes. The other node uses this information to consider *processes exchange* and replies. This interaction incurs additional non-trivial overhead, and is harder to design.

For the remaining atomic operation, two processes and three nodes are engaged, making them the most expensive operations to perform. They require complicated interaction schemes between the nodes. Furthermore, it means that for every (local) process chosen on the sending node, another node is selected, which in turn chooses another process (there), and only then the third node is selected for consideration. Hence, in addition to the augmented interaction overhead, the consideration is repeated many times using a large number of combinations.

Chapter 5

Evaluation of the Atomic Operations

This chapter presents quantitative measures for the use of the atomic operations. For each such operation, we develop an expression which reflects the change in the *Average Slow Down (ASD)* of the system, for arbitrary configurations. We then illustrate the use of these expressions in several scenarios, some of which presented in the previous chapter.

Throughout this chapter we shall use the following notation:

J_j -	Job number j,
P_A -	Node A,
$P(j)$ -	Node hosting job j,
$SD(j)$ -	Slowdown of job j,
$RC(J_j)$ -	Resource profile of J_j, i.e. $RC(J_j) = \langle T_i, C_{i,1}, \ldots C_{i,N} \rangle$,
$Jobs(P_A)$ -	Set of jobs running on P_A,
$\underline{Dep}(j)$ -	Set of jobs dependent† of job j,
$\overline{Dep}(j)$ -	Set of jobs independent† of job j,
$\underline{Dep}_A(j)$ -	Set of jobs‡ in P_A, that are dependent of j, i.e. $\underline{Dep}_A(j) = \underline{Dep}(j) \cap Jobs(P_A)$,
$\overline{Dep}_A(j)$ -	Set of jobs‡ in P_A, that are independent of j, i.e. $\overline{Dep}_A(j) = \overline{Dep}(j) \cap Jobs(P_A)$,
$L(p)$ -	The load on node p.

† Two jobs are dependent iff they communicate.
‡ Not including job j itself.

5.1 Single migration

Let P_A and P_B denote two nodes that are involved in a single migration operation. Assume that job J_i is migrated from P_A to P_B. Obviously, the set of other jobs that are affected by this migration is those that are already present in P_A or in P_B (in the discussion below we exclude J_i from $Jobs(P_A)$). Jobs that depend on J_i are treated differently from jobs that are independent of J_i. This is also true for jobs that are mapped to other nodes. Therefore, there are four different subsets of jobs to examine:

- $Dep_A(J_i) = \{j \in Jobs(P_A) : j \neq J_i \wedge j \text{ depends on (communicates with) } J_i\}$;
- $\overline{Dep}_A(J_i) = \{j \in Jobs(P_A) : j \neq J_i \wedge j \text{ is not dependent of } J_i\}$;
- $Dep_B(J_i) = \{j \in Jobs(P_B) : j \text{ depends on (communicates with) } J_i\}$;

and

- $\overline{Dep}_B(J_i) = \{j \in Jobs(P_B) : j \text{ is not dependent of } J_i\}$.

In order to find the change in the ASD , we first calculate the change in the slowdown factors for each of the above subsets, as well as for the migrating job itself. Then, since these are the only jobs that are affected by the migration, and since $ASD = \sum_j SD(j)$, we simply sum up the differences. We have:

- $\forall j \in \overline{Dep}_A(J_i):$ In this case there is one change - the load decreases by one unit, since j is independent of J_i. The slowdown factors before and after the migration are respectively: $SD_{old}(j) = L_A \cdot (\tilde{T}_{cpu} + \tilde{T}_{ipc})$ and $SD_{new}(j) = (L_A - 1) \cdot (\tilde{T}_{cpu} + \tilde{T}_{ipc})$. Hence the change is: $SD_{new}(j) \leftarrow SD_{old}(j) \cdot (L_A - 1)/L_A$. To further highlight this difference we denote it by $\Delta(j)$. We get:

$$\Delta(j) = -\frac{SD(j)}{L_A};$$

- $\forall j \in \overline{Dep}_B(J_i):$ This is similar to the previous case, except that we use the load of P_B, which increases. Therefore: $SD_{new}(j) \leftarrow SD_{old}(j) \cdot (L_B + 1)/L_B$. We get:

$$\Delta(j) = +\frac{SD(j)}{L_B};$$

- $\forall j \in Dep_A(J_i):$ The load decreases by one unit. In addition since J_i is no longer in P_A, T_{ipc} increases by $C_{J_i,j}$. The old slowdown is: $SD_{old}(j) = L_A \cdot (\tilde{T}_{cpu} + \tilde{T}_{ipc})$, while the new slowdown is: $SD_{new}(j) = (L_A - 1) \cdot (\tilde{T}_{cpu} + \tilde{T}_{ipc} + \frac{C_{J_i,j}}{T_j})$. Therefore: $SD_{new}(j) \leftarrow SD_{old}(j) \cdot \frac{L_A - 1}{L_A} + \frac{C_{J_i,j}}{T_j} \cdot (L_A - 1)$. The net change is therefore:

$$\Delta(j) = -\frac{SD(j)}{L_A} + \frac{C_{J_i,j}}{T_j} \cdot (L_A - 1);$$

- $\forall j \in Dep_B(J_i)$: This case is similar to the previous case, except that considering node P_B instead makes two differences: the load increases, and in addition T_{ipc} (of job j) decreases (since the IPC with J_i is performed locally via shared memory). Therefore: $SD_{new}(j) \leftarrow SD_{old}(j) \cdot \frac{L_B+1}{L_B} - \frac{C_{J_i,j}}{T_j} \cdot (L_B+1)$.

$$\Delta(j) = +\frac{SD(j)}{L_B} - \frac{C_{J_i,j}}{T_j} \cdot (L_B - 1);$$

- For $j = J_i$: The load changes from L_A to (L_B+1). In addition, new overhead is created due to IPC with $j \in Dep_A(J_i)$), while the corresponding overhead with $j \in Dep_B(J_i)$ is eliminated. Therefore:
$SD_{new}(J_i) \leftarrow SD_{old}(J_i) \cdot \frac{L_B+1}{L_A} + (L_B+1) \cdot \left(\sum_{j\in Dep_A(J_i)} \frac{C_{J_i,j}}{T_i} - \sum_{j \in Dep_B(J_i)} \frac{C_{J_i,j}}{T_i}\right)$. Thus:

$$\Delta(J_i) = \frac{L_B - L_A + 1}{L_A} \cdot SD(J_i) + (L_B+1) \cdot \left(\sum_{j\in Dep_A(J_i)} \frac{C_{J_i,j}}{T_i} - \sum_{j \in Dep_B(J_i)} \frac{C_{J_i,j}}{T_i} \right).$$

The total change in the ASD of the system, due to the migration of J_i is the sum of the changes in the slowdown factors of all the jobs, that is:

$$\begin{aligned}
\Delta_{ASD} &= \frac{L_B - L_A + 1}{L_A} \cdot SD(J_i) - \sum_{Jobs(P_A)} \frac{SD(j)}{L_A} + \sum_{Jobs(P_B)} \frac{SD(j)}{L_B} \\
&+ \sum_{Dep_A(J_i)} \left((L_A - 1) \cdot \frac{C_{J_i,j}}{T_j} + (L_B+1) \cdot \frac{C_{J_i,j}}{T_i} \right) \\
&- \sum_{Dep_B(J_i)} \left((L_B+1) \cdot \left[\frac{C_{J_i,j}}{T_j} + \frac{C_{J_i,j}}{T_i} \right] \right).
\end{aligned}$$

Example: to illustrate the usefulness of the last expression, consider the system shown in Figure 2.1(b). Assume that the two processes have the following resource profile: $\langle T, 0, C\rangle$ and $\langle T, C, 0\rangle$ respectively. In this case, $L_A = 2$ and $L_B = 0$. The slowdown factors of both processes is $L_A \cdot T/T = 2$. If one of the processes migrates to node P_B, then the loads would become: $L_A = L_B = 1$, and the corresponding slowdown factors are: $L \cdot (T+C)/T = 1 + C/T$. A migration is worthwhile `iff` the difference in the ASD is negative, i.e. $2 \cdot (1 + C/T) - 2 \cdot 2 = 2 \cdot C/T - 2 < 0$. The same inequality results by substituting the appropriate values in the above formula. We get:

$$\Delta_{ASD} = \frac{0-2+1}{2} \cdot 2 - \frac{2}{2} + (2-1) \cdot \frac{C}{T} + (0+1) \cdot \frac{C}{T} = -2 + 2 \cdot \frac{C}{T} < 0 \ ,$$

or

$$\Delta_{ASD} < 0 \quad \Longleftrightarrow \quad C/T > 1 \ .$$

This means that as long as the IPC overhead requires more CPU than the computation, the processes should run in the same node, otherwise, they should run in different nodes. Note that this is in line with the intuitive conclusion presented in Chapter 2 for this case.

5.2 Double migration

In this section we develop expressions for the difference in the *ASD* for the dual-tone operations, starting with the *double migration*. We consider a migration of a pair of communicating processes, J_i and J_k from node P_A to node P_B. This case is quite similar to the *single migration* case, with two important differences. First, the load changes by 2 units on each node (instead of 1 unit in the single migration case), and second, the cost of the IPC between J_i and J_k remains zero, since they move together to the same node.

The decrease in the load on node P_A by 2 units reduces the slowdown factor of the jobs that are running there. On the other hand, IPC between J_i (or J_k) and some of these jobs (which was previously local) will now impose network traffic, thus increasing the slowdown factors. The exact opposite holds for jobs that are mapped to node P_B, on which the load increases and IPC overhead may lessen. We divide the jobs on P_A and P_B into 8 disjoint subclasses, in a similar manner to the subclasses of the *single migration*. These classes are:

- $\forall j \in \overline{Dep}_A(J_i) \cap \overline{Dep}_A(J_k) : \Delta(j) = -2 \cdot \frac{SD(j)}{L_A}$;
- $\forall j \in Dep_A(J_i) \cap \overline{Dep}_A(J_k) : \Delta(j) = -2 \cdot \frac{SD(j)}{L_A} + \frac{C_{J_i,j}}{T_j} \cdot (L_A - 2)$;
- $\forall j \in \overline{Dep}_A(J_i) \cap Dep_A(J_k) : \Delta(j) = -2 \cdot \frac{SD(j)}{L_A} + \frac{C_{J_k,j}}{T_j} \cdot (L_A - 2)$;
- $\forall j \in Dep_A(J_i) \cap Dep_A(J_k) : \Delta(j) = -2 \cdot \frac{SD(j)}{L_A} + \frac{C_{J_i,j}}{T_j} \cdot (L_A - 2) + \frac{C_{J_k,j}}{T_j} \cdot (L_A - 2)$;
- $\forall j \in \overline{Dep}_B(J_i) \cap \overline{Dep}_B(J_k) : \Delta(j) = +2 \cdot \frac{SD(j)}{L_B}$;
- $\forall j \in Dep_B(J_i) \cap \overline{Dep}_B(J_k) : \Delta(j) = +2 \cdot \frac{SD(j)}{L_B} - \frac{C_{J_i,j}}{T_j} \cdot (L_B + 2)$;
- $\forall j \in \overline{Dep}_B(J_i) \cap Dep_B(J_k) : \Delta(j) = +2 \cdot \frac{SD(j)}{L_B} - \frac{C_{J_k,j}}{T_j} \cdot (L_B + 2)$;
- $\forall j \in Dep_B(J_i) \cap Dep_B(J_k) : \Delta(j) = +2 \cdot \frac{SD(j)}{L_B} - \frac{C_{J_i,j}}{T_j} \cdot (L_B + 2) - \frac{C_{J_k,j}}{T_j} \cdot (L_B + 2)$;
- For $j \in \{J_i, J_k\}$, the change in the slowdown factors of the migrating jobs is derived as in the *single migration*, except that in this case the load difference is 2 units. We get

$$\Delta(J_n) = \frac{L_B - L_A + 2}{L_A} \cdot SD(J_n) + (L_B + 2) \cdot \left(\sum_{j \in Dep_A(J_n)} \frac{C_{J_n,j}}{T_n} - \sum_{j \in Dep_B(J_n)} \frac{C_{J_n,j}}{T_n} \right).$$

The total change in the system's *ASD* for the *double migration* is thus given by:

$$\begin{aligned}
\Delta_{ASD} \quad &= \quad \frac{L_B - L_A + 2}{L_A} \cdot [SD(J_i) + SD(J_k)] - \sum_{Jobs(P_A)} 2 \cdot \frac{SD(j)}{L_A} + \sum_{Jobs(P_B)} 2 \cdot \frac{SD(j)}{L_B} \\
&+ \sum_{Dep_A(J_i)} \left((L_A - 2) \cdot \frac{C_{J_i,j}}{T_j} + (L_B + 2) \cdot \frac{C_{J_i,j}}{T_i} \right) \\
&- \sum_{Dep_B(J_i)} \left((L_B + 2) \cdot \left[\frac{C_{J_i,j}}{T_j)} + \frac{C_{J_i,j}}{T_i} \right] \right) \\
&+ \sum_{Dep_A(J_k)} \left((L_A - 2) \cdot \frac{C_{J_k,j}}{T_j} + (L_B + 2) \cdot \frac{C_{J_k,j}}{T_k} \right) \\
&- \sum_{Dep_B(J_k)} \left((L_B + 2) \cdot \left[\frac{C_{J_k,j}}{T_j)} + \frac{C_{J_k,j}}{T_k} \right] \right) .
\end{aligned}$$

Example: consider the set of processes shown in Figure 4.2(a). Assume that each pair of processes has the following resource profile: $\langle T, 0, C \rangle$ and $\langle T, C, 0 \rangle$ respectively. In this case the slowdown factors of all processes is the load on node A, i.e. 4, hence the *ASD* is 16. Applying a *single migration* will change this slowdown factors as following: for the (single) migrating process and its counterpart the slowdown factor becomes: $1 + C/T$ and $3 + 3 \cdot C/T$ respectively. For the other pair of processes, the slowdown factor is reduced to exactly 3. Therefore, the difference in the *ASD* due to a *single migration* is:

$$\Delta_{ASD} = [3 + 3 + (1 + C/T) + (3 + 3 \cdot C/T)] - (4 + 4 + 4 + 4) = -6 + 4 \cdot C/T,$$

or

$$\Delta_{ASD} < 0 \quad \Longleftrightarrow \quad C/T < 3/2 .$$

In other words, a migration would be avoided if the communication volume is substantially more dominant than the computation.

Alternatively, we can use *double migration* to migrate a pair of communicating processes together to node P_B, thereby avoiding undesired IPC overhead. In this case, the slowdown factors of all processes would be 2, and $\Delta_{ASD} = -8$. To verify this, if we substitute the appropriate values in the above expression for *double migration*, we get:

$$\Delta_{ASD} = \frac{0 - 4 + 2}{4} \cdot (4 + 4) - \left(2 \cdot \frac{4}{4} + 2 \cdot \frac{4}{4} \right) = -4 - (2 + 2) = -8 .$$

The meaning of this result is that a *double migration* always improves the state of the system, simply because it balances the load among the nodes without increasing the IPC. This conclusion further justifies the introduction of this dual-tone operation.

5.3 Processes exchange

This case deals with an exchange operation between two jobs, J_i and J_k, that are initially assigned to two nodes P_A and P_B respectively. Unlike the previous case, in this case the load in the two nodes remains unchanged. The only change to the ASD is due to changes in the IPC type, local vs. via the network. Jobs on P_A that communicate with J_i will suffer from the migrations, while jobs of P_B will benefit from it. A similar (but opposite) argument holds for the other migrating job, J_k. Note that The migration does not change the IPC between the migrating jobs themselves, since they remain mapped to different nodes. The change in the slowdown factors, for the various combinations, is summarized below:

- $\forall j |\ j \in Dep_A(J_i) \wedge j \notin Dep_A(J_k): \quad \Delta(j) = \ + \frac{C_{J_i,j}}{T_j} \cdot L_A$;
- $\forall j |\ j \notin Dep_A(J_i) \wedge j \in Dep_A(J_k): \quad \Delta(j) = \ - \frac{C_{J_k,j}}{T_j} \cdot L_A$;
- $\forall j |\ j \in Dep_A(J_i) \wedge j \in Dep_A(J_k): \quad \Delta(j) = \ + \frac{C_{J_i,j}}{T_j} \cdot L_A \ - \frac{C_{J_k,j}}{T_j} \cdot L_A$;
- $\forall j |\ j \in Dep_B(J_i) \wedge j \notin Dep_B(J_k): \quad \Delta(j) = \ - \frac{C_{J_i,j}}{T_j} \cdot L_B$;
- $\forall j |\ j \notin Dep_B(J_i) \wedge j \in Dep_B(J_k): \quad \Delta(j) = \ + \frac{C_{J_k,j}}{T_j} \cdot L_B$;
- $\forall j |\ j \in Dep_B(J_i) \wedge j \in Dep_B(J_k): \quad \Delta(j) = \ - \frac{C_{J_i,j}}{T_j} \cdot L_B \ + \frac{C_{J_k,j}}{T_j} \cdot L_B$;
- For $j \in \{J_i, J_k\}$, the change in the slowdown factor is due to changes in the IPC type with the other jobs, i.e. local or networked. In addition, being executed on a different node than before the migration may result in a different load. We get:

  $\Delta(J_i) = \frac{L_B - L_A}{L_A} \cdot SD(J_i) + \sum_{j \in Dep_A(J_i)} \left(\frac{C_{J_i,j}}{T_i} \cdot L_B \right) - \sum_{j \in Dep_B(J_i)} \left(\frac{C_{J_i,j}}{T_i} \cdot L_B \right)$;

  and

  $\Delta(J_k) = \frac{L_A - L_B}{L_B} \cdot SD(J_k) - \sum_{j \in Dep_A(J_k)} \left(\frac{C_{J_k,j}}{T_k} \cdot L_A \right) + \sum_{j \in Dep_B(J_k)} \left(\frac{C_{J_k,j}}{T_k} \cdot L_A \right)$.

The total ASD change due to a *processes exchange* operation is therefore given by:

$$\begin{aligned}
\Delta_{ASD} \;=\;& \frac{L_B - L_A}{L_A} \cdot SD(J_i) + \frac{L_A - L_B}{L_B} \cdot SD(J_k) \\
&+ \sum_{Dep_A(J_i)} \left(\frac{C_{J_i,j}}{T_j} \cdot L_A + \frac{C_{J_i,j}}{T_i} \cdot L_B \right) - \sum_{Dep_B(J_i)} \left(\frac{C_{J_i,j}}{T_j} \cdot L_B + \frac{C_{J_i,j}}{T_i} \cdot L_B \right) \\
&- \sum_{Dep_A(J_k)} \left(\frac{C_{J_k,j}}{T_j} \cdot L_A + \frac{C_{J_k,j}}{T_k} \cdot L_A \right) + \sum_{Dep_B(J_k)} \left(\frac{C_{J_k,j}}{T_j} \cdot L_B + \frac{C_{J_k,j}}{T_k} \cdot L_A \right).
\end{aligned}$$

Example: consider the system shown in Figure 4.1(a). Assume execution pattern for each pair of processes as in the previous examples. Since the configuration is symmetric, the slowdown factor of all processes is the same, and equals to $2 \cdot (1 + C/T)$. A migration of a single process would change the slowdown factors as following: for the migrating process and its counterpart it will become exactly 3; for the other process on the same node it will be $3 \cdot (1 + C/T)$, and for the remaining process $(1 + C/T)$. The change in the ASD is therefore:

$$\Delta_{ASD} = [3 + 3 + 3 \cdot (1 + C/T) + (1 + C/T)] - 4 \cdot [2 \cdot (1 + C/T)] = 2 - 4 \cdot C/T \ ,$$

which implies that:

$$\Delta_{ASD} < 0 \quad \Longleftrightarrow \quad C/T > 1/2 \ .$$

This means that a *single migration* will not be beneficial unless the IPC is significantly more dominant than the computation.

Evidently using *processes exchange* will bring the system to a better configuration, shown in Figure 4.1(b). The symmetry of the system is conserved by this operation, hence the slowdown factor is the same for all the processes, and its value is 2. The change in the ASD now is: $4 \cdot 2 - 4 \cdot [2 \cdot (1 + C/T)] = -8 \cdot C/T$. Substituting the appropriate values in the previous formula yields;

$$\Delta_{ASD} = \frac{2-2}{2} \cdot 2 + \frac{2-2}{2} \cdot 2 - \left(\frac{C}{T} \cdot 2 + \frac{C}{T} \cdot 2\right) - \left(\frac{C}{T} \cdot 2 + \frac{C}{T} \cdot 2\right) = -8 \cdot \frac{C}{T} \ .$$

We conclude that in the given system, any positive value of C will raise the requirement for *processes exchange* migration, to improve the performance.

5.4 Processes merge

Unlike the previous operations discuss so far, this atomic operation involves three nodes. Denote the nodes by P_A, P_B and P_C. We consider migrations of job J_i from P_A and job J_k from P_B, both to node P_C. The slowdown factor of the jobs in the sending nodes changes in the same way as in the case of *single migration*: one job is migrated out. The jobs in the destination node can be treated as a *double migration*, since two jobs are migrated to that node and the load increases by two units. For the same reason, the migrating jobs can be treated similarly, with the exception that any IPC between J_i and J_k becomes local. Following is a summary of the various cases:

- $\forall j \in \overline{Dep}_A(J_i): \ \Delta(j) = -\frac{SD(j)}{L_A} \qquad \forall j \in Dep_A(J_i): \ \Delta(j) = -\frac{SD(j)}{L_A} + \frac{C_{J_i,j}}{T_j} \cdot (L_A - 1);$
- $\forall j \in \overline{Dep}_B(J_k): \ \Delta(j) = -\frac{SD(j)}{L_B} \qquad \forall j \in Dep_B(J_k): \ \Delta(j) = -\frac{SD(j)}{L_B} + \frac{C_{J_k,j}}{T_j} \cdot (L_B - 1);$
- $\forall j \in \overline{Dep}_C(J_i) \cap \overline{Dep}_C(J_k)$: $\Delta(j) = +\ 2 \cdot \frac{SD(j)}{L_C}$;

- $\forall j \in Dep_C(J_i) \cap \overline{Dep}_C(J_k)$: $\Delta(j) = +\, 2 \cdot \frac{SD(j)}{L_C} - \frac{C_{J_i,j}}{T_j} \cdot (L_C + 2)$;
- $\forall j \in \overline{Dep}_C(J_i) \cap Dep_C(J_k)$: $\Delta(j) = +\, 2 \cdot \frac{SD(j)}{L_C} - \frac{C_{J_k,j}}{T_j} \cdot (L_C + 2)$;
- $\forall j \in Dep_C(J_i) \cap Dep_C(J_k)$: $\Delta(j) = +\, 2 \cdot \frac{SD(j)}{L_C} - \frac{C_{J_i,j}}{T_j} \cdot (L_C + 2) - \frac{C_{J_k,j}}{T_j} \cdot (L_C + 2)$;
- For $j \in \{J_i, J_k\}$:

$$\begin{aligned}\Delta(J_i) = {} & \tfrac{L_C - L_A + 2}{L_A} \cdot SD(J_i) - \tfrac{C_{J_i,J_k}}{T_i} \cdot (L_C+2) + \\ & \left(\textstyle\sum_{j \in Dep_A(J_i)} \tfrac{C_{J_i,j}}{T_i} - \sum_{j \in Dep_C(J_i)} \tfrac{C_{J_i,j}}{T_i}\right) \cdot (L_C + 2)\ ;\end{aligned}$$

and

$$\begin{aligned}\Delta(J_k) = {} & \tfrac{L_C - L_B + 2}{L_B} \cdot SD(J_k) - \tfrac{C_{J_k,J_i}}{T_k} \cdot (L_C+2) + \\ & \left(\textstyle\sum_{j \in Dep_B(J_k)} \tfrac{C_{J_k,j}}{T_k} - \sum_{j \in Dep_C(J_k)} \tfrac{C_{J_k,j}}{T_k}\right) \cdot (L_C + 2)\ .\end{aligned}$$

The total change in the ASD due to a *processes merge* operation is thus given by:

$$\begin{aligned}\Delta_{ASD} \;=\; & \frac{L_C - L_A + 2}{L_A} \cdot SD(J_i) + \frac{L_C - L_B + 2}{L_B} \cdot SD(J_k) - (L_C + 2) \cdot \left(\frac{C_{J_i,J_k}}{T_i} + \frac{C_{J_k,J_i}}{T_k}\right) \\ - & \sum_{Jobs(P_A)} \frac{SD(j)}{L_A} - \sum_{Jobs(P_B)} \frac{SD(j)}{L_B} + \sum_{Jobs(P_C)} 2 \cdot \frac{SD(j)}{L_C} \\ + & \sum_{Dep_A(J_i)} \left[\frac{C_{J_i,j}}{T_j} \cdot (L_A - 1) + \frac{C_{J_i,j}}{T_i} \cdot (L_C + 2)\right] \\ + & \sum_{Dep_B(J_k)} \left[\frac{C_{J_k,j}}{T_j} \cdot (L_B - 1) + \frac{C_{J_k,j}}{T_k} \cdot (L_C + 2)\right] \\ - & \sum_{Dep_C(J_i)} \left(\frac{C_{J_i,j}}{T_j} + \frac{C_{J_i,j}}{T_i}\right) \cdot (L_C + 2) \\ - & \sum_{Dep_C(J_k)} \left(\frac{C_{J_k,j}}{T_j} + \frac{C_{J_k,j}}{T_k}\right) \cdot (L_C + 2)\ .\end{aligned}$$

It is not difficult to demonstrate the necessity of the *processes merge* operation, by showing a process setup that only this operation improves. For example, this is the case with the configuration given in Figure 4.3.

5.5 Processes split

Processes split can be viewed as the reverse operation of *processes merge*. For the purpose of the analysis, assume the same notations for the nodes and the jobs as in the previous case, with only the direction of the migration being reversed. The analysis for all the jobs, except the jobs that migrate, is similar to the *processes merge*case, simply by reversing all the signs in the expression (e.g. replace the term $(L_A - 1)$ with $(L_A + 1)$, since the load on node P_A increases instead of decreasing). This is also true for the migrating jobs, except that the term $(L_C + 2)$ should be replaced by $(L_A + 1)$, and $(L_B + 1)$ for jobs J_i and J_k respectively, since they migrate to those nodes. Performing these modification to the formula of the *processes merge* yields the following formula for the change in the ASD :

$$
\begin{aligned}
\Delta_{ASD} \;&=\; \frac{L_A - L_C + 1}{L_C} \cdot SD(J_i) + \frac{L_B - L_C + 1}{L_C} \cdot SD(J_k) + (L_A + 1) \cdot \frac{C_{J_i,J_k}}{T_i} \\
&+\; (L_B + 1) \cdot \frac{C_{J_k,J_i}}{T_k} + \sum_{Jobs(P_A)} \frac{SD(j)}{L_A} + \sum_{Jobs(P_B)} \frac{SD(j)}{L_B} - \sum_{Jobs(P_C)} 2 \cdot \frac{SD(j)}{L_C} \\
&-\; \sum_{Dep_A(J_i)} \left(\frac{C_{J_i,j}}{T_j} + \frac{C_{J_i,j}}{T_i} \right) \cdot (L_A + 1) \;-\; \sum_{Dep_B(J_k)} \left(\frac{C_{J_k,j}}{T_j} + \frac{C_{J_k,j}}{T_k} \right) \cdot (L_B + 1) \\
&+\; \sum_{Dep_C(J_i)} \left[\frac{C_{J_i,j}}{T_j} \cdot (L_C - 2) + \frac{C_{J_i,j}}{T_i} \cdot (L_A + 1) \right] \\
&+\; \sum_{Dep_C(J_k)} \left[\frac{C_{J_k,j}}{T_j} \cdot (L_C - 2) + \frac{C_{J_k,j}}{T_k} \cdot (L_B + 1) \right] .
\end{aligned}
$$

Like the previous case, it is not difficult to find an example that illustrates the necessity of *processes split* operation.

5.6 Processes shift

The *processes shift* operation also involves three nodes, but in a different manner than the previous two operations. Let J_i and J_k be jobs migrating from node P_A to node P_B, and from node P_B to node P_C respectively. P_A is a sending node while P_C is a destination node. The load on both of these nodes changes by one unit. Therefore we can evaluate the change in the ASD of the jobs mapped to these nodes in the same way it was done for the sending and destination nodes of the *single migration* case. Also, the calculation for node P_B is like the *exchange operation* case, since one job leaves out and another job moves in. We get that:

- $\forall j \in \overline{Dep}_A(J_i) :\ \Delta(j) = -\frac{SD(j)}{L_A}$ $\quad \forall j \in Dep_A(J_i) :\ \Delta(j) = -\frac{SD(j)}{L_A} + \frac{C_{J_i,j}}{T_j} \cdot (L_A - 1);$
- $\forall j \in \overline{Dep}_C(J_k) :\ \Delta(j) = +\frac{SD(j)}{L_C}$ $\quad \forall j \in Dep_C(J_k) :\ \Delta(j) = +\frac{SD(j)}{L_C} - \frac{C_{J_k,j}}{T_j} \cdot (L_C + 1);$

- $\forall j|\ j \in Dep_B(J_i) \wedge j \notin Dep_B(J_k)\ \Delta(j) = -\ \frac{C_{J_i,j}}{T_j} \cdot L_B$;
- $\forall j|\ j \notin Dep_B(J_i) \wedge j \in Dep_B(J_k)\ \Delta(j) = \qquad + \frac{C_{J_k,j}}{T_j} \cdot L_B$;
- $\forall j|\ j \in Dep_B(J_i) \wedge j \in Dep_B(J_k)\ \Delta(j) = +\ \frac{C_{J_i,j}}{T_j} \cdot L_B + \frac{C_{J_k,j}}{T_j} \cdot L_B$;
- For $j \in \{J_i, J_k\}$:
  $\Delta(J_i) = \frac{L_B - L_A}{L_A} \cdot SD(J_i) + \left(\sum_{j \in Dep_A(J_i)} \frac{C_{J_i,j}}{T_i} - \sum_{j \in Dep_B(J_i)} \frac{C_{J_i,j}}{T_i}\right) \cdot L_B$;

  and

  $\Delta(J_k) = \frac{L_C - L_B + 1}{L_B} \cdot SD(J_k) + \left(\sum_{j \in Dep_B(J_k)} \frac{C_{J_k,j}}{T_k} - \sum_{j \in Dep_C(J_k)} \frac{C_{J_k,j}}{T_k}\right) \cdot (L_C + 1)$.

The total change in the *ASD* due to a *processes shift* operation is therefore given by:

$$\begin{aligned}
\Delta_{ASD} &= \frac{L_B - L_A}{L_A} \cdot SD(J_i) + \frac{L_C - L_B + 1}{L_B} \cdot SD(J_k) \\
&- \sum_{Jobs(P_A)} \frac{SD(j)}{L_A} + \sum_{Jobs(P_C)} \frac{SD(j)}{L_C} \\
&+ \sum_{Dep_A(J_i)} \left[\frac{C_{J_i,j}}{T_j} \cdot (L_A - 1) + \frac{C_{J_i,j}}{T_i} \cdot L_B\right] - \sum_{Dep_B(J_i)} \left(\frac{C_{J_i,j}}{T_j} + \frac{C_{J_i,j}}{T_i}\right) \cdot L_B \\
&+ \sum_{Dep_B(J_k)} \left(\frac{C_{J_k,j}}{T_j} \cdot L_B + \frac{C_{J_k,j}}{T_k} \cdot (L_C + 1)\right) - \sum_{Dep_C(J_k)} \left(\frac{C_{J_k,j}}{T_j} + \frac{C_{J_k,j}}{T_k}\right) \cdot (L_C + 1)\ .
\end{aligned}$$

As with all the other atomic operations, it is also possible to present a relatively simple example that highlights the unique contribution of the *processes shift.* Note that this operation may be especially handy if the system is almost balanced, where changing the loads by more than one unit is unfavorable. This prevents the use of either *double migration*, *processes merge* or *processes split.* In this context, *processes shift* is complementary to the *processes exchange* operation.

Chapter 6

Performance Evaluation

This chapter presents simulations of several test cases using the algorithms for the IPC optimization problem. These algorithms represent the atomic operation heuristics for finding an approximation to the solution of this problem. Our simulations are effective, straightforward means to evaluate the atomic operations, without the development of a full scale implementation. In order to provide some insight to the algorithms performance, a simulation program was designed and implemented. The chapter begins by a description of this simulator, followed by description of the the evaluation methodology, then we present the simulation results of several test cases and their analysis.

6.1 The simulator

The goal of the simulator was to imitate the execution of a set of parallel applications in a configuration that consists of a homogeneous computing cluster with several nodes, each running a time-sharing, Unix-like multitasking environment.

The simulator was written in *C++* [29]. It consists of three layers: the simulator core, the migration policy and an application generator.

6.1.1 The simulator core

The simulator core represents a distributed system model, similar to that described in Chapter 3. Each node is represented by an instance of class `Node`. A `Node` emulates a Unix-like node, including "ready-to-run" and "delayed" queues. Processes in the system are represented by instances of class `Proc`. Each `Proc` has an execution profile, determined by a configuration file, which describes its behavior throughout time.

The simulator core is responsible for spawning and executing processes in the nodes. The processes are managed using fair-share, round-robin scheduling, i.e., each process receives $1/load$ fraction of the CPU. The simulation is conducted in time-steps, where in each time-step every node executes its current running process for a CPU time slice. When given a time slice, a process can either compute or communicate with another process. If the communication is local, i.e., both processes reside on the same node, then it completes immediately. Otherwise, the communication time is a linear function of the message size, as described in Section 3.1.

Whenever a process is migrated from one node to another, it is placed in the "delayed" queue. The delay period is the time that takes to complete the migration. It is computed as a startup time plus the time to transfer the process over the network (depending on the process size). When the migration ends, the process is placed in the "ready-to-run" queue at its new node and it resumes its execution.

6.1.2 The migration policy

The migration policy is triggered by the simulator core periodically, in pre-specified intervals or when a change in the system state is detected. This layer implements the algorithm described in Section 4.4. The policy operates on every node, managing internal book-keeping. This policy controls the optimization decisions, such as when to migrate, which atomic operation to apply, which processes participate and what are the destination nodes.

The migration policy is capable of performing all the atomic operations that were defined in Section 4.2. To allow flexible use of the simulator, each of this operations can be disabled or enabled at configuration time. This way we can study the effect of using only a subset of the atomic operations. For example, we could disable all the *dual-tone* operations, in order to be able to compare the algorithm to a policy that allows migration of only a single process at a time.

6.1.3 The application generator

The application generator is responsible for generating and representing user applications. Parallel applications may vary in their communication topologies and computation requirements. These properties may significantly affect the behavior of the optimization algorithm. The application generator was designed to enable a flexible configuration of simple and complex applications, reflecting the execution of a large range of applications.

Applications are generated by spawning processes in the system, as was pre-determined in a special file. This file specifies the characteristics of the applications and optionally their activation time. This allows to introduce dynamics to the system, and watch the response of the algorithm. For each application, one of four initial assignment method is selected:

- Round-robin assignment;

- Random assignment;
- Clustered assignment - all processes start in the same node;
- Pre-specified assignment - when the optimal assignment is known.

6.1.4 Simulator output

The output of the simulator include the total execution time of the application(s), the total number of migrations performed, the number of atomic operations that were applied and their types. The execution time for multiple applications is taken as the time until all the processes complete their execution. The output also include the initial and final assignments and the corresponding average slowdown factors, as defined in Section 3.4.

In addition to the above, the simulator can also produce trace information, describing the entire execution with several levels of details. This allows for inspection of the way the algorithm behaves. In particular one can follow the optimization path selected by the algorithm and compare it to other alternatives.

6.2 Test cases examined

This section describes the test cases that were chosen to evaluate the performance of the algorithm. For simplicity, in all of the test cases the application is assigned a regular, fixed execution profile. This mitigates the job of determining an optimal assignments. The optimal assignment that is derived serves as a basis for comparison with our dynamic migration algorithm. The first, static test cases are used to examine the basic capabilities and limitations of both the algorithm and the atomic operations set.

It is also of high importance to evaluate the performance of the algorithm in dynamic cases, to study its on-line capabilities. Dynamics is introduced by spawning different applications at different (simulation) times. As with the static cases, the dynamic cases were designed so that we can easily determine the optimal assignment at any time.

One of the most important parameters of the simulated applications is their granularity, i.e. computation to communication ratio (C/T). The optimal assignment can vary depending on this parameter. In the case of $C/T \ll 1$, the problem reduces to a load balancing problem, which is solved by an even distribution of processes among the nodes. In practice, even highly communication message-passing applications experience only moderate values ($C/T \approx 1$). Hence, the extreme case of ($C/T \gg 1$) is of no practical importance. In our test cases, we chose the value of C/T in the range of $0 \leq C/T \leq 1$. That is, the heaviest communicating jobs experience up to the same amount of communication compared to pure computation.

6.2.1 Static cases

Although the suggested optimization algorithm is particularly useful for dynamic cases, it still helps in static cases, since it saves programmers the burden of conducting a pre-run analysis for deriving an optimal static assignment. The static cases are also useful for the study of the behavior of the atomic operations.

In the set of static cases we included the following test cases:

- **No communication -** all the processes are CPU-bound, without IPC between the processes;
- **Ring topology -** of communicating processes, where each process communicates with two adjacent processes;
- **Torus topology -** a two-dimensional cyclic mesh, in which each process communicates with four adjacent processes;
- **Disjoint sets -** a mixture of several disjoint rings, toruses and or CPU-bound jobs.

6.2.2 Dynamic cases

Much of the complexity of the IPC optimization problem is attributed to the dynamic properties of the model. In such an environment, where jobs with unknown topologies and resource profiles arrive and terminate in an unpredictable manner, the state of the system rapidly changes, even after an optimal assignment is obtained. This requires that the algorithm responds to the fluctuations and continuously adapts to the system's state, with different number and types of jobs every time.

The problem with evaluating the performance of the algorithm in the dynamic cases is finding an optimal assignment policy in order to have a lower bound to compare to. To overcome this problem the dynamic cases are designed as a sequence of static applications, arriving at some rate to the cluster. If the period between two successive entries of applications to the system is long enough, then the optimal assignment is one that corresponds to a static case of disjoint sets that includes the set of jobs executing *at that time*. Consequently, we derived a hand-crafted optimal assignment for these cases.

6.3 CPU bound processes

The first case examined is a set of CPU bound processes, with no inter-process communications. All the processes are independent of each other, reducing the IPC optimization problem to a load balancing problem. As shown in Section 3.4, the optimal assignment is to disperse the processes among the nodes evenly.

We tested the algorithm with two initial placement strategies: round-robin (RR) and clustered (CL), in which all the processes are created in one (arbitrary) node. We note that the RR placement is optimal in this case.

No. of nodes	*8 processes*			*32 processes*		
	RR	CL		RR	CL	
	[time]	[time]	No. Migs.	[time]	[time]	No. Migs.
4	20000	20146	7	80000	80198	33
8	10000	10296	7	40000	40302	34
16	10000	10296	7	20000	20682	32

Table 6.1: *Assignment of processes with no communication*

Table 6.1 presents the results of the executions for clusters of 4, 8 and 16 nodes, with 8 and 32 CPU-bound processes. In the table, columns 2 and 5 show the total completion time (in simulation time units) of the (optimal) RR assignment, for 8 and 32 processes respectively. Columns 3-4 and columns 6-7 give the corresponding time units and the number of migrations for a clustered assignment (CL),

From the table it can be seen that the total completion time in CL assignments is slightly larger than the RR optimal completion times. The algorithm requires some time until it stabilizes and reaches the optimal assignment, since it operates on a periodic basis. This (fixed) overhead is the reason for the small performance gap. We note that in all the test cases, the algorithm reached the optimal assignment with a minimal number of migrations. This result remains the same even if we restrict the algorithm to use only *single migrations*. The added value of the *dual-tone* operations in this case is that fewer decisions are required to optimize the system.

6.4 Processes with a ring topology

In a *ring* topology, each process communicates with exactly two adjacent processes. Such a communication pattern is frequent for many parallel applications, e.g. finite-element solvers. The use of a ring topology is particular attractive since in this case it is relatively easy to find the optimal assignment. For example, Figure 6.1 depicts an optimal assignment of a program with 8 processes and a *ring* topology, into a system with four nodes. The assignment in this figure is optimal for low to moderate values of C/T. A round robin assignment of the same program is depicted in Figure 6.2. Clearly it is inferior to the optimal assignment.

In the simulation, we executed the algorithm with sets of 8 and 16 processes with a ring topology, on clusters of 4 and 8 nodes. For each case we examined three initial placement policies: Round Robin (RR), clustered (CL) and random (RND). We used three different values for C/T, to reflect different communication loads. Since the optimization path taken

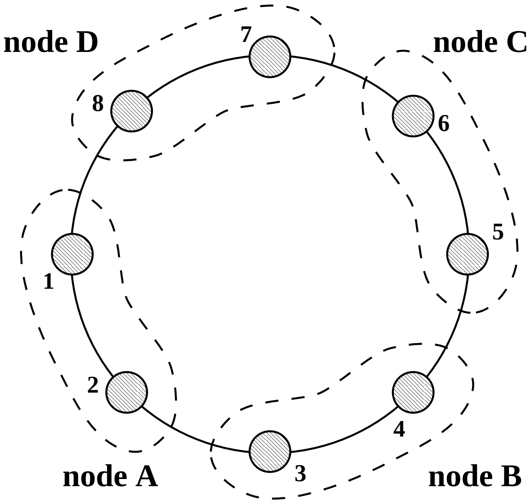

Figure 6.1: *Optimal assignment of 8 processes with a ring topology to 4 nodes*

by the algorithm is non-deterministic, we repeated each execution multiple times (50 for RR and CL, and 200 times for RND), then calculated the average results and compared it to the optimal assignment (OPT).

Table 6.2 presents the results for a low volume of communication, where $C/T = 0.1$. The table presents the *Average Slow Down* (*ASD*) obtained by the algorithm for each case, normalized (divided) by the optimal *ASD* . The results are shown once for the execution of the algorithm with the entire set of the (*dual and single tone*) atomic operations (ASD_{dual}), and once with only the *single tone* operations (ASD_{single}). In the table, columns 2-3 list the results for a ring with 8 processes running on 4 nodes, for ASD_{dual} and ASD_{single} respectively. Columns 4-5 and 6-7 list the corresponding results for a ring of 16 processes running on 4 and 8 nodes respectively.

Assignment Policy	**Configuration**					
	Ring-8, 4 nodes		Ring-16, 4 nodes		Ring-16, 8 nodes	
	ASD_{dual}	ASD_{single}	ASD_{dual}	ASD_{single}	ASD_{dual}	ASD_{single}
Optimal	1	1	1	1	1	1
Round-robin	1	1.091	1.023	1.143	1.011	1.091
Clustered	1	1.025	1.006	1.012	1.006	1.014
Random	1	1.028	1.009	1.052	1.006	1.031

Table 6.2: *Normalized ASD for $C/T = 0.1$ (Dual- and Single-tone) for a Ring topology*

From the table it can be seen that in all the cases the performance of the algorithm is within 2.5% of the optimal value. On average, the percentage of cases in which an optimal assignment was obtained (hereinafter called *optimality percentage*) of the algorithm is 70%.

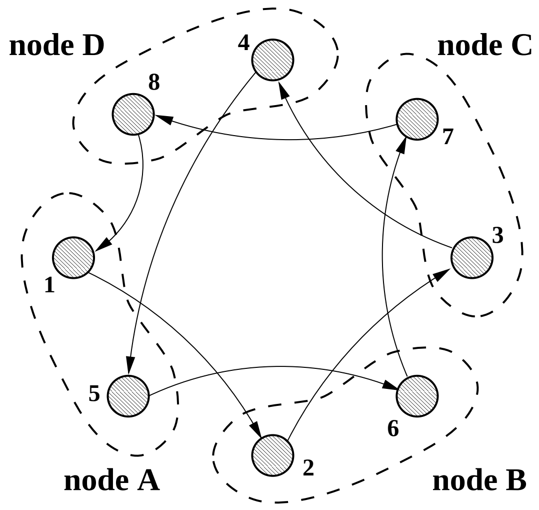

Figure 6.2: *Suboptimal RR assignment of 8 processes with a ring topology to 4 nodes*

This can be explained by the low value of C/T, which implied relatively small *optimality percentage*. Yet, for the same reason the performance penalty in these cases is insignificant. This point is further discussed below.

Observe that the effectiveness and necessity of the *dual-tone* operations is reflected from the performance results of the reduced algorithm, with only the *single-tone* operation enabled. These results are consistently worse than the algorithm with both the *single-* and *dual-tone* operations for all categories. In particular, note that the reduced algorithm has a 100% failure in reaching an optimal configuration, for RR and CL initial placements (that is, a 0% *optimality percentage*). This further emphasizes the importance of the *dual-tone* operations, in light of the fact that the *single-tone* operation is not sufficiently powerful to reach optimality in these cases.

We note that the main reason for the relatively small deviations from the optimal values of the various cases, is the small value of C/T. In order to examine the influence of the this parameter, we executed the same cases with larger values of C/T. Table 6.3 presents the results for $C/T = 0.6$, and Table 6.4 presents the results for $C/T = 1$.

From these tables we observe two trends. Firstly, the performance of the algorithm improves with the increase of C/T. This observation is strengthened by the *optimality percentage* data for $C/T = 0.6$ and $C/T = 1$, which are on average 90.5% and 99.3% respectively, for the algorithm with all the atomic operations. Consequently the value of ASD_{dual} obtained was better as well. However, this was compensated for by the increased performance penalty due to the additional communication weigh, such as in the case of clustered assignment of Ring-16 on 8 nodes (Table 6.3).

The second trend that can be seen is the growing performance gap between the algorithm with the complete set of atomic operations and the reduced algorithm (with only the *single-*

Assignment Policy	Configuration					
	Ring-8, 4 nodes		Ring-16, 4 nodes		Ring-16, 8 nodes	
	ASD_{dual}	ASD_{single}	ASD_{dual}	ASD_{single}	ASD_{dual}	ASD_{single}
Optimal	1	1	1	1	1	1
Round-robin	1	1.375	1.013	1.057	1	1.375
Clustered	1	1.078	1.005	1.024	1.033	1.104
Random	1	1.012	1.001	1.023	1.020	1.029

Table 6.3: *Normalized ASD for $C/T = 0.6$ (Dual- and Single-tone) for a Ring topology*

Assignment Policy	Configuration					
	Ring-8, 4 nodes		Ring-16, 4 nodes		Ring-16, 8 nodes	
	ASD_{dual}	ASD_{single}	ASD_{dual}	ASD_{single}	ASD_{dual}	ASD_{single}
Optimal	1	1	1	1	1	1
Round-robin	1	1.500	1.020	1.062	1	1.5
Clustered	1	2	1	2.666	1	4
Random	1	1	1	1.029	1	1

Table 6.4: *Normalized ASD for $C/T = 1$ (Dual- and Single-tone) for a Ring topology*

tone operations). This trend is particularly prominent in the cases of round-robin and clustered initial assignments. The *optimality percentage* of the reduced algorithm remained 0% in almost all the cases for round-robin and clustered assignments, with the case of round-robin of 16-Ring on 4 nodes being the only exception.

One point that deserves further discussion is the *optimality percentage* of the algorithm for small C/T, which is lower than what one might expect. For low values of C/T the problem resembles the load balancing problem where load considerations dominate, and practically overrule communications considerations when determining the optimization path. Consider a possible result of such an optimization path, as shown in Figure 6.3, where a *ring* of length 16 is assigned on an 8 nodes cluster. It is clear that this is a sub-optimal assignment, that can be improved by, for example, a 3-phase shifting of processes ($5 \mapsto C, 7 \mapsto D, 9 \mapsto E$ and $11 \mapsto nodeH$). Nevertheless the *dual-tone* operations are too weak to improve this particular state. The low C/T ratio discourages creation of a load-imbalance which might lead later to a better assignment. On the other hand, the additional communication overhead incurred on processes 5 and 12 is small as well, and so is the overall performance penalty.

In the last example, when the value of C/T is increased, process 5 prefers to join, for example, node C, to reduce communication overhead, even at the cost of an increased load. In addition, the role of communications considerations increases, and better decisions are made along the optimization path, yielding an improved assignment.

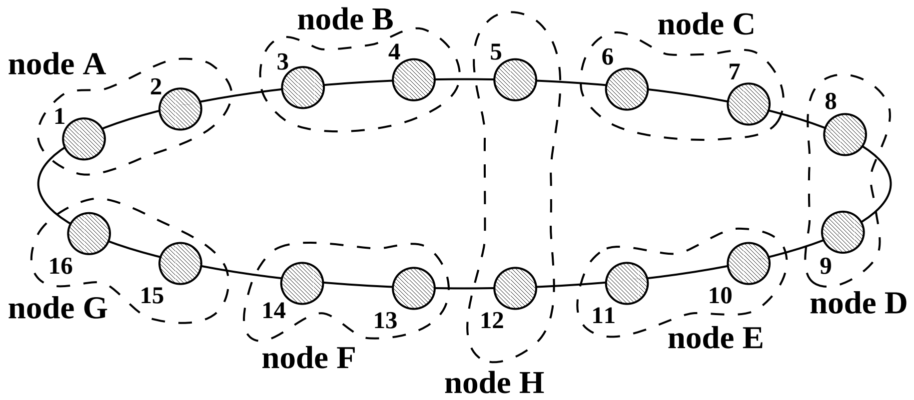

Figure 6.3: *Sub-optimal assignment demonstrating weakness of the dual-tone operations*

6.5 Processes with a torus topology

A *torus* is a cyclic two dimensional mesh or the Cartesian product of two rings i.e., it can be considered as a generalization of a ring. A torus communication topology is frequent among parallel applications. Figure 6.4 depicts a 16 (4×4) parallel job with a torus topology, optimally assigned to a 4 nodes system.

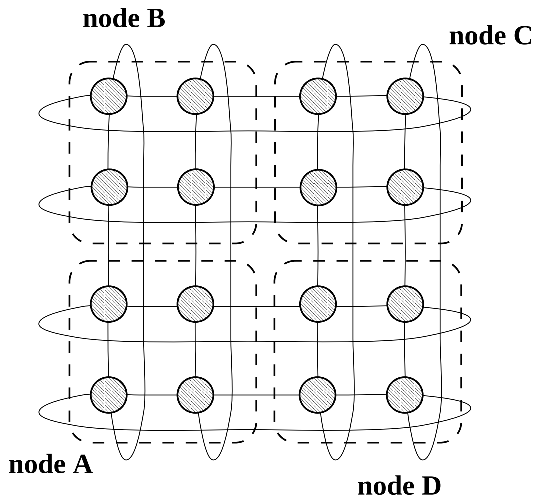

Figure 6.4: *Optimal assignment of a* 4×4 *torus on 4 nodes*

In the simulations, we executed the algorithm for toruses of 2×4 and 4×4 processes, on clusters of 4 and 8 nodes. In each case we used the RR, CL and RND initial placements, we executed the algorithm many times and then averaged the obtained results. We then compared the results to the optimal assignment (OPT). These executions were conducted for 2 different values of C/T.

Tables 6.5 presents the results of the executions for $C/T = 0.6$. The corresponding results for $C/T = 0.8$ are shown in 6.6. From the tables it can be seen that the results and the outcome are similar to those of the ring topology.

Assignment Policy	**Configuration**					
	Torus 2×4 on 4 nodes		Torus 4×4 on 4 nodes		Torus 4×4 on 8 nodes	
	ASD_{dual}	ASD_{single}	ASD_{dual}	ASD_{single}	ASD_{dual}	ASD_{single}
Optimal	1	1	1	1	1	1
Round-robin	1	1.188	†	†	1.003	1.158
Clustered	1	1	1.012	1	1	1.008
Random	1	1.023	1.014	1.033	1	1.024

†In this case round-robin is an optimal assignment

Table 6.5: *Normalized ASD for $C/T = 0.4$ (Dual- and Single-tone) for a Torus topology*

Assignment Policy	**Configuration**					
	Torus 2×4 on 4 nodes		Torus 4×4 on 4 nodes		Torus 4×4 on 8 nodes	
	ASD_{dual}	ASD_{single}	ASD_{dual}	ASD_{single}	ASD_{dual}	ASD_{single}
Optimal	1	1	1	1	1	1
Round-robin	1	1	†	†	1.004	1.182
Clustered	1	1	1.007	1	1	1.009
Random	1	1	1.008	1.029	1	1.028

†In this case round-robin is an optimal assignment

Table 6.6: *Normalized ASD for $C/T = 0.8$ (Dual- and Single-tone) for a Torus topology*

6.6 Static disjoint sets of processes

Static disjoint sets of processes simulate the concurrent execution of several independent applications. For simplicity each application is either a ring, a torus or a set of CPU-bound processes. In the preceding cases the application that was executed maintained an ordered (regular) structure, and in a sense symmetric, with equal jobs. Simulation of combinations of several applications gives a more complete analysis of the algorithms. We simulated three scenarios with mixed topologies, as discussed below.

6.6.1 Case *I*

The configuration of case I is given in Table 6.7. It consists of five concurrent applications, with a total of 20 processes that started simultaneously, on a cluster of 8 nodes. The algorithm was executed with three possible initial placement policies: Round-Robin (RR), Clustered (CL) and Random (RND). The optimal placement was obtained off-line. The results were averaged over 50 executions, and compared against the optimal placement.

Applic. No.	Topology	Size	C/T
1	Torus	3×2	0.3
2	Torus	3×2	0.3
3	Ring	3	0.2
4	Ring	3	0.2
5	CPU-bound	2	N/A
Total no. of jobs: 20 Cluster size: 8 nodes			

Table 6.7: *Simulation configuration for Case I*

Assignment Policy	*Dual tone*		*Single tone*	
	ASD_{dual}	OP_{dual}	ASD_{single}	OP_{single}
Optimal	1	100%	1	100%
Round-robin	1	100%	1.060	20%
Clustered	1	100%	1.056	0%
Random	1	100%	1.055	25%

Table 6.8: *Normalized ASD (Dual- and Single-tone) and optimality percentage for case I*

Table 6.8 presents the results, using the *ASD* and the *optimality percentage* (OP) obtained from the simulations. From the table it can be seen that the algorithm performs well on a set of disjoint applications. Observe the improved performance of the algorithm with the *dual-tone* operations compared to the algorithm with only the *single-tone* operations. However, it is interesting to see that the performance difference between the two algorithms is relatively small. This can be attributed to the specific configuration and parameters of this case. This was later verified by simulation of an example in which the C/T ratio was increased for the two toruses (designated 1 and 2 in Table 6.7) to 0.7. In this case the normalized *ASD* for a clustered initial placement was 1.88, and the *optimality percentage* remained approximately the same.

6.6.2 Case *II*

The configuration of case II is given in Table 6.9. It consists of seven concurrent applications, with a total of 24 processes that started simultaneously, on a cluster of 16 nodes. The algorithm was executed with the same initial placement policies as in case I, the results were averaged over 50 executions, and compared against the optimal placement, that was derived off-line.

Applic. No.	Topology	Size	C/T
1	Torus	3×2	0.3
2	Ring	4	0.5
3	Ring	3	0.4
4	Ring	3	0.4
5	Ring	3	0.2
6	Ring	3	0.2
7	Ring	2	0.5
Total no. of jobs: 24 Cluster size: 16 nodes			

Table 6.9: *Simulation configuration for Case II*

Assignment Policy	*Dual tone*		*Single tone*	
	ASD_{dual}	OP_{dual}	ASD_{single}	OP_{single}
Optimal	1	100%	1	100%
Round-robin	1	100%	1.019	0%
Clustered	1	100%	1.121	0%
Random	1	100%	1.018	0%

Table 6.10: *Normalized ASD (Dual- and Single-tone) and optimality percentage for case II*

Table 6.10 presents the *ASD* and the *optimality percentage* (OP) obtained from the simulations, for the four initial placement policies. By comparing the results to the results of the previous case, it can be seen that the performance of the algorithm remain unchanged with the increase in the number of jobs. We note that in Case II the number of jobs is a multiple of the cluster size. This apparently has no effect on the algorithm with both the *single-tone* and *dual-tone* operations, but seems to impair the results obtained by the algorithm with only the *single-tone* operations. This can be explained by the fact that the loads on the nodes are about the same. Therefore migrations that change the load difference between two nodes by two units (e.g. *single migration*) are unfavorable. Rather, *processes exchange* is more likely to be applicable.

6.6.3 Case *III*

The configuration of case III is given in Table 6.11. It consists of nine concurrent applications, with a total of 32 processes that started simultaneously, on a cluster of 16 nodes. The algorithm was executed with the same four initial placement policies as in the previous cases. As before, the results were averaged over many executions, and compared against the optimal placement.

Applic. No.	Topology	Size	C/T
1	Torus	3×2	0.3
2	Ring	3	0.4
3	Ring	3	0.4
4	Ring	3	0.2
5	Ring	3	0.2
6	Ring	2	0.5
7	Ring	2	0.5
8	Ring	2	0.5
9	CPU-bound	8	N/A
Total no. of jobs: 32 Cluster size: 16 nodes			

Table 6.11: *Simulation configuration for Case III*

Assignment Policy	*Dual tone*		*Single tone*	
	ASD_{dual}	OP_{dual}	ASD_{single}	OP_{single}
Optimal	1	100%	1	100%
Round-robin	1.001	90%	1.227	0%
Clustered	1.001	65%	1.009	18%
Random	1.001	45%	1.014	13%

Table 6.12: *Normalized ASD (Dual- and Single-tone) and optimality percentage for case III*

Table 6.12 presents the *ASD* and the *optimality percentage* (OP) obtained from the simulations. The results confirm the trends that emerged from the first two cases. First, the algorithm performs well on a large system with many jobs. Second, there is a clear performance difference between the algorithm with the complete set of atomic operations and the reduced algorithm (with only the *single-tone* operations), indicating the importance of the *dual-tone* operations. Note that increasing the number of jobs in the system does not affect this difference. This result is counter-intuitive, since there are seemingly more degrees of freedom for the reduced algorithm to search for an optimization path.

6.7 Dynamic disjoint sets of processes

Dynamic disjoint sets of jobs reflect the concurrent execution of several independent applications in a dynamic environment, where parallel jobs arrive and leave the system throughout time. The goal here is to demonstrate the adaptation of the optimization algorithm to such a dynamic environment.

The simulation included a sequence of applications arriving at a constant rate to the system, that are assigned to the cluster using round-robin assignment. Each application includes either ring(s), torus(es) or CPU bound jobs(s). The arrival rate of the applications was slow enough to let the algorithm reach a stable state before the next job arrived. At that time we measured the *ASD* of the system and compared it to the *ASD* of the optimal assignment at that state. We also compared the *ASD* to that achieved by the reduced algorithm (i.e. with only the *single tone* operations).

The set of applications that was used is similar to that of Case *III*, as described in Table 6.11. From this set we created two sequences, differing only by their arrival order. The arrival order of the sequences is given in Table 6.13. The applications indices in the table refer to the numbering in Table 6.11. The applications were assigned to a cluster with 8 nodes. Each case was executed 50 times and the results were averaged.

Sequence number	**Application spawn order**						
	1st	2nd	3rd	4th	5th	6th	7th
1	App_6	App_4, App_5	App_7	App_2, App_3	App_8	App_9	App_1
2	App_6	App_4, App_5	App_9	App_7, App_8	App_2, App_3	App_1	

Table 6.13: *Arrival orders of the sequences for dynamic disjoint sets*

The results of the executions of the two sequences are given in Table 6.14 and Table 6.15 respectively. The tables present the *ASD* of the stable state reached after the arrival of each application in the sequence. The values are normalized by the optimal *ASD* at that stage. In the tables, rows 1-2 give the normalized *ASD* and the *optimality percentage* (OP) respectively at each stage, for the algorithm with the complete set of atomic operations. Rows 3-4 give the corresponding results for the reduced algorithm.

	Phase						
	1st	2nd	3rd	4th	5th	6th	7th
ASD_{dual}	1	1	1	1	1	1	1.006
OP_{dual}	100%	100%	100%	100%	100%	100%	80%
ASD_{single}	1	1	1.019	1.144	1.200	1.176	1.031
OP_{single}	100%	100%	70%	22%	30%	30%	9%

Table 6.14: *Normalized ASD and OP for execution sequence 1*

	Phase					
	1st	2nd	3rd	4th	5th	6th
ASD_{dual}	1	1	1	1.002	1.004	1.011
OP_{dual}	100%	100%	100%	96%	84%	50%
ASD_{single}	1	1	1.102	1.085	1.017	1.060
OP_{single}	100%	100%	0%	0%	86%	0%

Table 6.15: *Normalized ASD and OP for execution sequence 2*

From the tables it can be seen that the algorithm performs well on the two cases examined. It responds to changes in the state of the cluster and adaptively reassigned jobs to nodes to improve the ASD. As in previous cases, there is a clear performance differences between the algorithm that use both *single-* and *dual-tone* operations vs. the reduced algorithm that uses only *single-tone* operations. This is highlighted by the ASDs and the *optimality percentages*. We note that the results in the first two phases of the execution are identical because the initial (round-robin) assignment is optimal at these stages. Also note that the termination of jobs also implies a change in the system's state, therefore requiring the algorithm to respond. The observations above hold for these cases as well, i.e. the the performance differences between the two algorithms, and the near-optimality of the algorithm with both the *single-tone* and *dual-tone* operations remain the same.

Chapter 7

Conclusions and Future Work

This paper presented a distributed, on-line algorithm for the dynamic Inter Process Communication (IPC) optimization problem in a scalable computing cluster system. The algorithm uses partial knowledge and it attempts to reduce both the amount of the IPC and the computational load of the nodes, to improve the cluster wide resource utilization and the overall performance. It continuously reassigns tasks to nodes by means of process migration, in order to reduce the Average Slow Down (ASD) of the overall cluster.

The IPC optimization problem is a critical issue that affects the efficiency of execution of parallel programs in distributed memory machines. The main issue is to find a golden path between two conflicting objectives: keeping processes separated, to increase the degree of the parallelism vs. process clustering, to reduce the IPC overhead. Since this is an NP-hard problem, our algorithm conducts a local search for a better assignment of processes to nodes, using a set of heuristic rules.

The idea used for developing these heuristics was the observation that simple operations, such as reassignment of a single task, are generally too weak, leading to assignments that constitute local optima. On the other hand, we required that the rules are kept simple to reduce complexity, and that they can be applied in a decentralized manner. We introduced the idea of *dual-tone* operations, that involve reassignment of two tasks at a time, and thus significantly reduce the probability to reaching a local optimum. We identified six such operations, which, together with the *single tone* operation, called *atomic operations*, constitute the basic building blocks of the algorithm. The algorithm repeatedly improve the system's performance by applying these operations to migrate tasks among the nodes. Our approach leverages from the advantages of a local search and therefore it is scalable.

The algorithm that we developed does not guarantee to find optimal assignments. It has been shown to give near-optimal results for common communication topologies, both in static and dynamic test cases. We tested the algorithm using a simulation program built for this purpose. The performance results obtained show that the algorithm provides an assignment that is optimal or near-optimal in all the tested cases. It outperforms a load balancing assignment that does not take into account communications requirement. The results also demonstrate both the usefulness and viability of the *dual tone* operations as important building blocks of the algorithm.

The algorithm that was developed could be implemented in any environment that supports dynamic process migration. It could be part of a comprehensive, dynamic resource management policy, that manages other resources, such as memory and I/O. Together, these algorithms are geared to create an efficient execution environment, with improved performance and a high degree of overall resource utilization and sharing in scalable systems. We believe that such a policy along with a process migration mechanism could ease the design, development and use of scalable, distributed memory multicomputer systems.

This work can be extended in several directions. One important extension would be to relax some of the assumptions regarding the communication model. In particular, the model presented in Chapter 3 assumes asynchronous communications. It would be interesting to generalize it to synchronous communication between the processes. The implications, however, would have an enormous effect on the the complexity of the problem. Firstly, it would introduce synchronization (*pause*) time of communicating tasks, e.g. when a receiver waits for a message therefore becoming idle during this period. This would change the way the load is calculated. Secondly, the effect of a task migration can no longer be considered to be local; a task migration affects the *pause* time of both the tasks in the participating nodes and tasks that communicate with these nodes. This in turn affects other tasks which communicate with the latter and so on. This creates a chain effect which results in non-local consequences. Another way to extend the communication model is to allow overlap of computation and communications (e.g., by using DMA data transfer). This would imply similar complexity to synchronous communication. Finally, we can relax the assumption of a fully connected network with uniform links between the node, generalizing it to any network topology.

Another possible extension of this work can be by looking for ways to improve the local search strategy. This can be achieved, for example, by infrequently applying techniques such as simulated annealing, or the use of probabilistic decisions. Another approach is to use higher order *atomic operations* as described in Section 4.3. Finally we note that a natural extension of this work would be to implement it in a scalable computing cluster that supports dynamic process migration, such as MOSIX.

Acknowledgments

We are indebted to Amnon Shiloh for his valuable contributions.
This work was supported by grants from the Ministry of Defence and the Ministry of Science.

Bibliography

[1] N. Alon, A. Barak, and U. Manber. On Disseminating Information Reliably Without Broadcasting. In *Proc. 7th Intl. Conf. on Distributed Computing Systems*, pages 74–80, Berlin, September 1987.

[2] Y. Amir, B. Averbuch, A. Barak, R.S. Borgstrom, and A. Keren. An Opportunity Cost Approach for Job Assignment and Reassignment in a Scalable Computing Cluster. In *Proc. PDCS '98*, October 1998, to appear.

[3] T.E. Anderson, D.E. Culler, and D.A. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[4] A. Barak and A. Braverman. Memory Ushering in a Scalable Computing Cluster. In *Proc. IEEE Third Int. Conf. on Algorithms and Architecture for Parallel Processing*, Melbourne, December 1997.

[5] A. Barak, S. Guday, and R.G. Wheeler. The MOSIX Distributed Operating System, Load Balancing for UNIX. In *Lecture Notes in Computer Science, Vol. 672*. Springer-Verlag, 1993.

[6] A. Barak and A. Keren. On-line Scheduling of Communicating Processes in a Scalable Computing Cluster, 1998.

[7] A. Barak and O. La'adan. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. *Journal of Future Generation Computer Systems*, 13(4-5):361–372, March 1998.

[8] D.R. Cheriton and K. Harty. A Market Approach to Operating System Memory Allocation. URL page: http://www-dsg.stanford.edu/Publications.html, Stanford Univ.

[9] F. Douglis and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software – Practice & Experience*, 21(8):757–785, August 1991.

[10] K. Efe. Heuristic Models of Task Assignment Scheduling in Distributed Systems. *Computer*, pages 50–56, June 1982.

[11] G. Fox et al. *Solving Problems on Concurrent Processors*. Prentice-Hall, 1988.

[12] D.G. Feitelson and L. Rudolph. Toward Convergence in Job Schedulers for Parallel Supercomputing. In D.G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1162. Springer-Verlag, 1996.

[13] D.F. Ferguson, C. Nikolaou, J. Sairamesh, and Y. Yemini. Economic Models for Allocating Resources in Computer Systems. In *Ed. Scott Clearwater*. World Scientific Press, 1996.

[14] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completenes*. Freeman, 1979.

[15] A. Gerasoulis and T. Yang. On the Granularity and Clustering of Directed Acyclic Task Graphs. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):686–701, June 1993.

[16] M. Harchol-Balter and A.B. Downey. Exploiting Process Lifetime Distributions for Dynamic Load Balancing. In *Proc. ACM SIGMETRICS '96 Conference on Measurement and Modeling of Computer Systems*, pages 13–24, Philadelphia, May 1996.

[17] D.S. Johnson, C.H. Papadimitriou, and M. Yannakakis. How Easy Is Local Search ? *J. Comput. System Sci.*, 37:79–100, 1988.

[18] B.W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *AT&T Bell Labs. Tech. J.*, (49):291–307, February 1970.

[19] S. Kirkpatrick, C.C. Gelatt, and M.P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, May 1983.

[20] C. McCreary and H. Gill. Automatic Determination of Grain Size for Efficient Parallel Processing. *Comm. ACM*, pages 446–451, 1994.

[21] M.A. Palis, J.C. Lious, and D.S.L. Wei. Task Clustering and Scheduling for Distributed Memory Parallel Architectures. *IEEE Trans. on Parallel and Distributed Systems*, 7(1):46–55, January 1996.

[22] C.H. Papadimitriou and M. Yannakakis. Towards an Architecture-Independent Analysis of Parallel Algorithms. *SIAM Journal of Computing*, 19(2):322–328, April 1990.

[23] S. Plotkin. Competitive Routing of Virtual Circuits in ATM Networks. *IEEE Journal on Selected Areas in Communications*, August 1995.

[24] H.E. Rewini and T.G. Lewis. Scheduling Parallel Program Task onto Arbitrary Target Machines. *J. Parallel and Distributed Computing*, 9:138–153, 1990.

[25] E. Rosti, E. Smirni, G. Serazzi, and L.W. Dowdy. Analysis of Non-Work-Conserving Processor Partitioning Policies. In D.G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 949. Springer-Verlag, 1995.

[26] B.A. Shirazi, A.R. Hurson, and K.M. Kavi. *Scheduling and Load Balancing in Parallel and Distributed Systems.* IEEE Computer Society Press, Los Alamitos, CA, 1995.

[27] I. Stoica, H. Abdel-Wahab, and A. Pothen. A Microeconomic Scheduler for Parallel Computers. In D.G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 949. Springer-Verlag, 1995.

[28] H.S. Stones. *High Performance Computer Architectures.* Addison-Wesley, Reading, MA, third edition, 1993.

[29] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1984.

[30] M. Stumm. The Design and Implementation of a Decentralized Scheduling Facility for a Workstation Cluster. In *Proc. Second Conf. Computer Workstations*, pages 12–22, Los Alamitos, CA, 1988. IEEE Press. Order No. 810.

[31] E.G. Talbi and P. Bessière. A Parallel Genetic Algorithm for the Graph Partitioning Problem. In *Intl. Conf. on Supercomputing*, 1991.