Abstract

Title of Dissertation: Toward Optimizing Distributed Programs Directed By Configurations

Tae-Hyung Kim
Institute for Advanced Computer Studies
and Computer Science Department
University of Maryland
College Park, Maryland 20742 , Doctor of Philosophy, 1996

Dissertation directed by: Associate Professor James M. Purtilo
Department of Computer Science

Networks of workstations are now viable environments for running distributed and parallel applications. Recent advances in software interconnection technology enables programmers to prepare applications to run in dynamically changing environments because module interconnection activity is regarded as an essentially distinct and different intellectual activity so as isolated from that of implementing individual modules. But there remains the question of how to optimize the performance of those applications for a given execution environment: how can developers realize performance gains without paying a high programming cost to specialize their application for the target environment? Interconnection technology has allowed programmers to tailor and tune their applications on distributed environments, but the traditional approach to this process has ignored the performance issue over gracefully seemless integration of various software components.

Networks of workstations can be virtual parallel machines. For a distributed and parallel application on such environments, an ability to write performance-literate programs is as important as that to seemlessly integrate distributed modules. Our dissertation research is an effort to extend the plain interconnection technology to that with a variety of performance attributes. The RPC (remote procedure call) paradigm is used at the module programming level because it adopts a widely used and understood procedure call abstraction as the sole

mechanism of remote operations and thus helps to shape reusable components. Most of performance related decisions are pertinent to the interconnections among software components.

Our effort toward performance tuning consists of two main thrusts. One is an automatic adaptation from a performance configuration, which is analogous to the process of software interconnection for traditional structure-oriented configurations. We present how a performance configuration can be represented as an extension to traditional module interconnections. The other is an optimal transformation for RPC statements in an individual module using various program analysis techniques. Conventional stub generation based approach to implement RPC paradigm cannot serve for performance improvement because of its synchronous property. In concert with the two systematic approaches toward optimizing distributed programs, programmers can have high performance and conceptual simplicity in writing distributed programs.

This research has been supported by Office of Naval Research under contract No. N000149410320

Toward Optimizing Distributed Programs Directed By Configurations

by

Tae-Hyung Kim
Institute for Advanced Computer Studies
and Computer Science Department
University of Maryland
College Park, Maryland 20742

Dissertation submitted to the Faculty of the Graduate School
of The University of Maryland in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
1996

Advisory Committee:

Associate Professor James M. Purtilo , Chair/Advisor
Professor John Gannon
Professor Michael Brin
Associate Professor Richard Gerber
Assistant Professor Adam Porter

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

A new trend in parallel processing has emerged to use "clusters of workstations" over Local Area Networks (LANs) as cost-effective parallel computing platforms. This new trend is gaining popularity to establish a new parallel processing paradigm: "network-based computing." It has two noteworthy advantages. First, it provides an opportunity to have a parallel machine virtually with no extra costs, if software supports are properly provided. Workstations are ubiquitous and most of them have been under-utilized at most of times. Second, the virtual parallel machine can be constructed so as to take advantage of some special resources that are locally available on some of the network hosts, for example, graphic processors or vector processors. However, still there is an order of magnitude difference in the speed of latency and transmission rate from a real parallel machine whether it is a distributed memory machine like Intel Paragon [Hoc94] or shared memory machine like IBM RP3 [BMW85]. Many researches have engaged in providing novel hardware or software solutions to alleviate such an obstacle for workstation clusters to be a viable environments for running parallel applications. Hardware solutions like SHRIMP (Scalable High-performance Really Inexpensive Multiprocessor) [BLA$^+$93] and PAPERS (Purdue's Adaptor for Parallel Execution and Rapid Synchronization) [DMSM94] projects are to develop an add-on interface unit that is connected to each workstation so that the resulting cluster can run at faster communication speeds and even allow fine-grain parallel execution. Although the communication in the LAN-based networks of workstations is getting faster, no substantial results have been reported to assure whether the add-on interfaces or other communication hardwares can really accomplish a comparable communication performance with real parallel machines. As long as there is a noticeable difference between the two platforms for parallel execution, a special software support, that has an expertise in the 'new' parallel platforms, is called for.

Writing distributed and/or parallel programs is difficult for programmers, and even more difficult when high performance is required. Many mechanisms to achieve better performance in distributed programming have been proposed [BST89, Geh86, Geh90, Gen81, LS88]; however, in practice these mechanisms are hard to utilize, and do not take into account the burden placed on programmers who already encounter difficulty in writing functionally correct programs. The capability to write an efficient program is not enough, unless the writing complexity is controlled in a systematic way. Furthermore, most of these mechanisms are expressed by special programming language constructs for specifying the exact semantics on communication and synchronization [BST89]. Such languages are not good at accommodating the programming skills of those who are already accustomed to conventional programming languages like C.

A great deal of this difficulty in developing large distributed programs arises at the early stages of program development, when the relationship between modules' functionality, their interactions and overall performance is hard to discern. For a given module's functionality as dictated by some design, it is possible to implement many program units, each having some different calling conventions, servicing style and communication properties, yet all maintaining the same functionality. Previously this flexibility in how to implement the module resulted in a burden to the programmer, who was tasked with selecting one of the implementations based upon too little information, and who then would be faced with great programming burden should one of those decisions need to be changed later.

Since many mechanisms can be expressed in terms of the high-level configuration of application modules, we sought to derive a practical adaptation system for configuration level programming. This approach would allow programmers to express performance improvement techniques abstractly (in terms of the configuration, instead of the low-level implementation), and then prepare appropriate implementations automatically.

The objective of this dissertation has been to provide such an adaptation system, to allow practical employment of existing performance improvement techniques; to suggest new techniques; and to allow programmers the freedom to study the impact of various techniques – in concert with one another, as desired – upon the application. Under our adaptation framework, the programmer's original implementation of a module is transformed in two directions: (1) each RPC statement at client sites is translated into a set of fine grained message passing primitives (*optimization phase*), and (2) the *stubs* are generated at server sites to implement the particular techniques (e.g. load balancing and/or scheduling) specified at the configuration level (*adaptation phase*).

1.1 Motivation

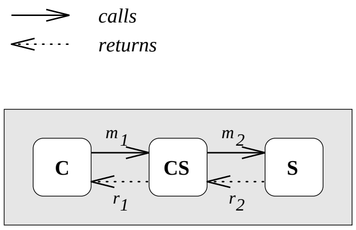
Our approach is motivated by the success of on-line system configuration technologies [Kra90] based on the perception that module interconnection activity is to be an essentially distinct and different intellectual activity from that of implementing individual modules; that is, "programming-in-the-large" is distinct from "programming-in-the-small" [DK76]. Moreover, workstation clusters are becoming viable environments for running parallel applications. One of the characteristics of such systems is the lack of solidity in a configuration of hardware platform. At the same time, performance is a key issue in writing distributed parallel programs. It would be an interesting step to adopt these configuration technologies to a performance improvement domain. We share a same philosophy of which module interconnection activity should be isolated from intra-module programming activity. Performance-related factors need to be delayed up to configuration level programming. Otherwise, programmers may encounter a large portion of their original programs for rewriting although their functionalities remain unchanged, because many performance improving techniques are tightly related to the structure of target workstation clusters. In many cases, such extra codes for better performance are orthogonal to the functional behavior among distributed software modules. By isolating performance factors from module programming level, module reusabilities can be enhanced. Not only it helps to enhance portability in workstation cluster environments, it can also provide an opportunity for programmers to experiment various performance-affecting configurations for performance tuning.

To carry out this approach, we consider the process of writing high-performance distributed parallel programs with two cooperative levels of programming: module- and configuration-level programming. Configuration programs will be expressed by a MIL (Module Interconnection Language) that has been used in the precursor of this dissertation research, *software packager* [CP91]. The MIL syntax is slightly extended to include performance-related factors for our purpose (Chapter 4). Module level programming is conducted by conventional programming languages like C with a remote procedure call abstraction for non-local interactions among software modules.

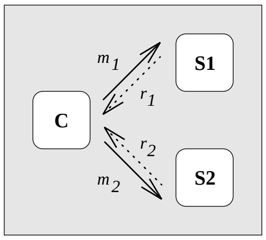
Conventional stub-generation based RPC implementations [BN84, Gib87, CP91] suffer low performance because parallelism is inhibited and communications may be redundant. That we want to circumvent those problems motivates us to have a *parallelization phase*. This is to develop source-level transformation of RPC-based distributed programs for higher performance. The RPC paradigm adopts the model of client-server computing; caller and callee correspond to client and server, respectively. Traditional researches on improving RPC program performance have focused on reducing latency and transmission time within this pairwise form [JZ93, BELL90, GG88]. When this simple topology extends to a *network* of client-server model computing, more advanced optimization other than just efficient pairwise hooking between client and server is called for. Figure 1.1 shows two basic topologies to form a general application. These two topologies are basic units where we can account for our optimal source-level transformation techniques. The optimization goals are "enhancing parallelism" (**Goal 1**) and "reducing communication length" (**Goal 2**).

In Figure 1.1 (a), the client **C** calls its server **CS** and it successively calls **S** to fulfill **C**'s request. The module **CS** plays both roles respectively to **C** and **S**. We name it *parallelism in depth* because the parallelism stretches in depth in the call graph. Normally, **C** and **CS** are blocked when **S** is working. Parallelism in depth (**Goal 1**) can be exploited if there are useful operations to continue during the call at a client. Towards **Goal 2**, the performance can be improved if we can establish a direct message passing path between an indirect client-server relationship like that of **C** and **S**. Direct path means that a recipient can get a data earlier, and consequently, it can start what it is supposed to do earlier. Moreover, it enables **C** to make a single multicast command instead of two separate request sending messages. A single multicast is faster than a series of point-to-point communication because the message passing system can keep same data in its internal buffer until delivered to all recipients. We need to assure that m_2 (r_2) is equal to m_1 (r_1) to make this possible (**Algorithm 5.2**).

In Figure 1.1 (b), the client **C** calls **S1** and then **S2**. We name it *parallelism in breadth* because the parallelism exists in the direction of breadth in the call graph. Normally, **S1**



(a) Parallelism in depth



(b) Parallelism in breadth

Figure 1.1: Basic topologies in client-server model from the perspective of optimization.

and **S2** cannot run in parallel due to a synchronous nature of RPC. However, the parallel execution of the two servers should be allowed, unless there is a data dependence between **S1** and **S2**. For the special case that the value of $r1$ is equal to the value of $m2$, we can establish a direct message passing path between **S1** and **S2**, which is for the **Goal 2**. This may be significant if **C** calls **S1** and **S2** in a loop.

Notice that it is intentional to treat every argument individually when its optimal message passing path is being sought. Each argument has its own data path. For example, if $S3(r_1, r_2)$ is called in **C** successively, it is desirable for r_1 and r_2 to be passed directly from **S1** and **S2** to **S3**.

Meanwhile, a structure of a distributed program is fairly diverse and it has an important meaning to the resulting performance. A structure is defined in a configuration that orchestrates a set of component modules in terms of their interactions. The *adaptation phase* is how to come up with compilable source codes according to the user's decisions in the configuration program. Figure 1.2 illustrates an example of a logical configuration of a distributed program that consists of three distinct modules. The arrows represent client-server relationships: $f1$ calls $f2$ and/or $f3$. Figure 1.3 enumerates a couple of possible physical configurations according to the various mappings from a logical module to a physical workstation in a cluster of workstations. Naming is one of the problems of message passing based distributed programming. The two communication parties (i.e. *sender* and *receiver*) should know the names of each other. In other words, if a physical configuration should be changed, then we have to change the codes in modules. In Figures 1.3 (a), (b) and (c), they have same topologies. Nonetheless, a change from one physical configuration to another one causes the corresponding source code changes due to the naming problem. For example, the change from Figure 1.3 (a) to Figure 1.3 (b) incurs source code changes in all modules. The change from Figure 1.3 (a) to Figure 1.3 (c) incurs changes only in $f1$ and $f3$. In Figure 1.3 (d), it is desirable to use interprocess communication primitives for the communication between $f1$

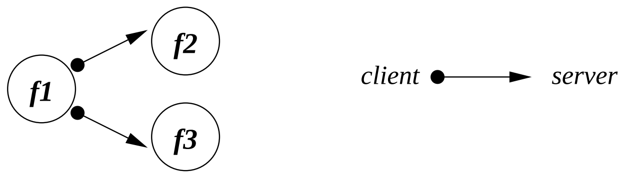


Figure 1.2: A Logical Configuration.

and $f2$, which is more efficient. These are examples of what programmers should encounter if they decides to use message passing paradigm for distributed applications. Obviously, it is true that message passing gives maximum flexibility in terms of writing the finest-tuned codes by programmers themselves. However, if the target platforms are workstation clusters, that flexibility accompanies formidable rewritings, even for simple structural changes as illustrated above.

Replacing a workstation with faster one can surely be a step to speed up. For example, in Figures 1.3 (a) and (b), if W4, W5 and W6 are faster than W1, W2 and W3, respectively, the same program in Figure 1.2 may run faster. However, such a change is not enough to improve significantly the overall performance of a program in general. Mostly, such a change is somewhat related to the porting and/or system reconfigurations rather than aimed at improving performances. Conventional interconnection methods like `polygen` [CP91] can be used to generate compilable source codes for this kind of various structural changes at configuration level.

More significant improvement can be achieved through a server replication. Server replication is a useful strategy in many cases, but also produces new problems that need to be carefully handled. Otherwise, even a parallelization anomaly, which is the longer elapse time for the bigger number of servers as explained in Chapter 6. Replication causes topological changes in a program structure. How to organize a pool of servers is a configuration-level decision too – for example, master/slave or pipeline style, and so on. Load imbalance is a serious impediment to achieving good performance if master/slave is a chosen style. Load balancing can be understood as an effort in a combination or one of two different kinds of module interactions. One is an interaction between a master and its slaves. A master process determines the workload for each slave. This is known as load distribution (or initialization). The other is an interaction among slaves. This is for load migration to balance loads among slaves at run-time. We will see various load balancing schemes can be parameterized where a configuration program contains a proper set of those parameters in Chapter 4 and Chapter 6.

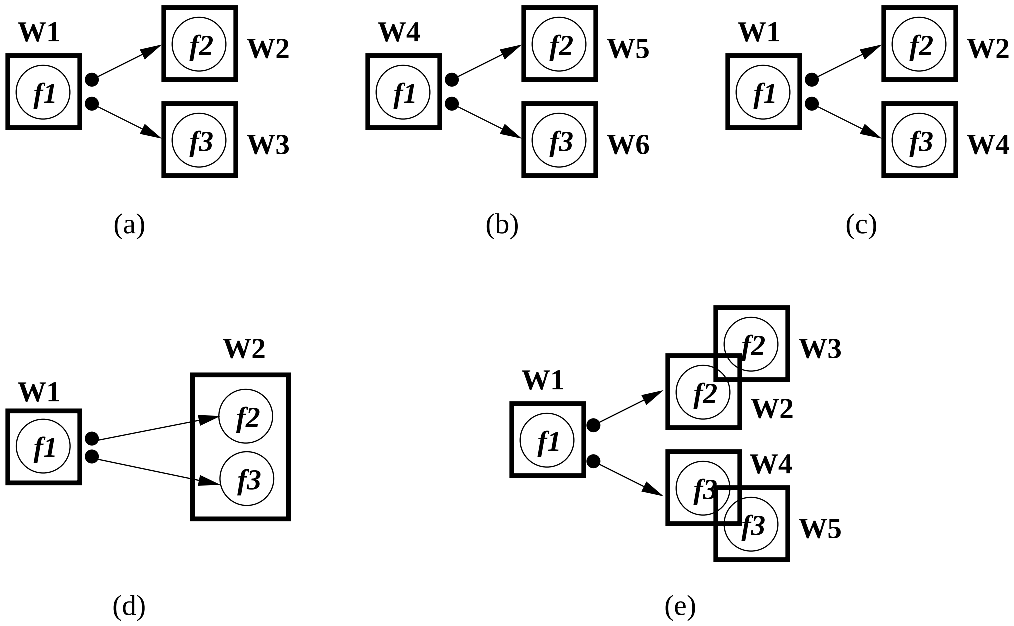


Figure 1.3: Various Physical Configurations.

1.2 Overview of Approach

Our approach towards optimizing RPC-based distributed programs consists of three primary thrusts. First, we study how to present various configurations that result in different performances after all. Such a configuration should be adequate to automatically generate compilable source codes from the basic module programs according to the directions in a configuration program. How to generate those compilable codes are the next two issues. As for a *parallelization phase*, we present a source-level transformation framework that is to transform RPC statements in a program into a proper set of message passing primitives. The transformed modules are expected to exploit parallelism and to optimize communication behaviors based on the given control and data dependence constraints. On the other hand, as a part of an *adaptation phase*, we focus on load balancing schemes, which are added to the original module interconnection activities. As a result, those important performance factors are able to be decoupled from module-level programming. The final high performance executables are automatically generated according to those user decided performance factors with a proper performance improving techniques like parallelizing and load balancing.

1.2.1 Configuration for High Performance

Module interconnection activity is understood to be an essentially distinct and different intellectual activity from that of implementing individual modules, that is "programming-in-the-large" is distinct from "programming-in-the-small" [DK76]. Analogously, this observation applies to performance programming as well. Decisions concerning how a configuration might be adapted in order to allow use of performance improvement mechanisms are inherently different from the task of tailoring individual program units and their interfaces to execute as dictated by the abstract decision. Thus, each module is written to satisfy its functional requirements while each configuration program is written to specify performance related as well as interconnection related information. Many existing performance oriented mechanisms can be achieved by using ordinary modules with proper configuration programs and source-to-source translation techniques. This frees programmers from making extensive amounts of manual adaptations for various performance configurations.

This builds upon the MIL (Module Interconnection Language) approach [CP91, Pur94] for distributed programming, where the original MIL specification is intended only for structural presentation of interfaces between interacting processes. We append performance related specifications onto each interface specification in a MIL. As the performance factors are isolated from the module programming level, changing that information in order to fine tune the performance requires not whole changes in source modules, but regeneration of new executables for the performance configuration.

1.2.2 Optimal Transformation

Two representative programming models for distributed memory machines are available for programmers, message passing (MP) and distributed share memory (DSM). Message passing primitives [Sun90, For93, BL93, Pur94, Fel79, Coo80] are expressive enough to program for efficiency; however, they are too low-level to write large distributed programs. Programmers are fully responsible for matching send/receive pairs, allocating buffers, and marshaling/unmarshaling data correctly. Programming under DSM systems [Car93] eases such difficulties, but the resulting programs suffer efficiency due to false sharing and coherence maintaining overhead especially in a distributed-memory machine environments. Our transformation based approach for implementing RPC paradigm is an effort to strike a compromise between these two models, using the RPC paradigm for writing distributed programs plus a source transformation framework for improving performance. Procedure call abstraction has been favored since early programming era because it contributes to construct a well structured modular program, which allows to reuse existing modules and helps write and maintain a large program by giving a clear view of its structure. The RPC paradigm adopts a widely used and understood procedure call abstraction as the sole mechanism of remote operations; thus it simplifies distributed programming by abstracting from details of communication and synchronization.

A distributed program is usually written with a number of different abstraction layers. It is natural to implement each layer of abstraction as a distinct module (or procedure). Following such a natural flow of concept would help write large distributed programs. However, this fact does not necessarily mean that the ideal flow of resulting program for high performance should be consistent with the conceptual flow of RPC paradigm. Two problems should be addressed to use RPC paradigm for high performance distributed programming. First, the parallelism is inhibited under the paradigm since the caller blocks during the call while it is desirable to make use of the time between sending requests and getting the responses back. Second, an unnecessary communication is likely to occur especially when a system is layered and implemented on the basis of modularization. For example, the communication between far distant layers might require a series of communications between a series of adjacent layers. The major problem of traditional stub generation based methods [BN84, Gib87] for implementing RPC paradigm is that it just adopts the natural flow of modularization as its actual flow of a program while the ideal flow of it does not conform to that way the program is written.

To cope with the discrepancy between the conceptual flow to write a program conveniently and the ideal flow to run a program efficiently, we present a source transformation framework for RPC-based distributed programs, in order to decide the proper locations of MP primitives for exploiting function level parallelism and reduction in communication lengths. It has several advantages over conventional stub generation: (1) it can be safely parallel – correctness is kept because it is transformed under preserving the given dependence constraints, (2) using fine-grained MP primitives to implement an RPC statement gives an opportunity for further code optimization through static program analysis techniques, and (3) modularization is not discouraged because the actual communication paths will be restructured optimally based on the given control and data dependences rather than the modular structure as written.

1.2.3 Load Balancing

In a distributed parallel program, tasks are generated and distributed to multiple processors to be processed simultaneously. Load imbalance is a serious impediment to achieving good performance as it leaves some processors idle, when they could be working to make progress. While global load balancing should still be an issue in the whole operating system's concern, our focus is on balancing parallel tasks within an application. Since minimizing the execution time of an application is more important than average response time, each processor needs to keep making progress rather than merely to have a balanced load. Although the latter state may finally lead to the former, this is not a primary goal to shorten the finish time. From a program's viewpoint, loops are the largest source of task parallelism in a parallel application. A loop is called a *parallel loop* (DOALL-loop) if there are no data dependences among all iterations. The question of how to allocate an iteration to a particular processor for minimizing the total execution time is known as a loop scheduling problem [TY86, KW85, PK87, TN91, CLZ95].

Networks of workstations are somewhat new environments for loop scheduling problems: the communication delay is longer and the granularity of a sub-task is coarser. To our knowledge, the first work on parallel loop scheduling in a network of heterogeneous workstations was done by Cierniak *et al.* [CLZ95]. They considered three aspects of heterogeneity — loop, processor, and network — and developed algorithms for generating optimal and sub-optimal schedules of loops. Two major limitations are that it is static and that the loop heterogeneity model is linear. In this dissertation, we present a new dynamic load balancing method for parallel loops of more general patterns, since many non-scientific applications such as the DNA sequence search problem [CG89] or the Mandelbrot set computation [FvDF+93], which are good candidate applications for workstation clusters, often do not carry conventional regular loop patterns. The unpredictable patterns can even be detrimental to those improvements [KW85, PK87, TN91], although the pure self-scheduling scheme is orthogonal to the loop patterns. Our new load balancing scheme reassures the important role of configuration-level programming towards higher performance because a proper load balancing topology can be easily constructed under such a programming environment.

1.3 Summary of Contributions

The major contributions of this dissertation are itemized as follows:

- The configuration issues are popular in constructing large distributed softwares [DK76, CP91, Kra90]. We have extended the idea into the performance issue from the original interconnection related ones. One may perceive that any structural change in a distributed program configuration could result in a different performance. This dissertation elaborated on this perception in more details toward performance improvement. We showed what kind of performance factors affects the overall performance of a distributed program and how they can be represented in forms of MIL (Module Interconnection Language) based configuration programming. We studied how such an approach help to do a seamless process in developing high performance distributed programs [KP95].

- We have developed an automatic parallelization of RPC-based distributed programs [KP96b]. RPC is a convenient paradigm for the sake of writing programs at the sacrifice of performance, if traditional stub generation based approaches [BN84, Gib87, CP91] are used. The automatic parallelization technique compensates programmers for the performance problem.

- A new decentralized load balancing scheme has been developed for workstation cluster environments [KP96a]. This development intensifies the importance of MIL style programming toward performance tuning because the scheme showed that the load balancing power is dependent on the topology of load migration network. Configurable load migration networks have had no application areas before the emergence of workstation clusters. The optimal topology is hard to decide in advance and subjected to be changed for performance tuning. A topological change is what a MIL program is for. A programmer does not have to rewrite a module unless its functionality remains unchanged.

- Moreover, other well-known load balancing schemes (e.g. water-marking, sender-initiated, receiver-initiated, or their combinations) can be parameterized in a configuration program. A new scheme may also be adopted later in order to generate the necessary code for the new load balancing scheme. Programmers are free to experiment various schemes that perform differently depending on applications without having to change many of written codes.

1.4 Outline of the Dissertation

This dissertation is organized as follows. In Chapter 2 we survey background and related work. Chapter 3 discusses the basic forms of parallel programming structures in order to illustrate that an efficient form of module interactions in a distributed program varies from application to application. Chapter 4 discusses a configuration-level programming that is topped over the conventional module programming level for various high performance oriented configurations. Some performance related parameters and how they can be represented in a configuration are illustrated. In Chapter 5 we provide a source-level transformation framework that transforms one and each of RPC statements into a proper combination of message passing primitives in the light of enhancing parallelism and reducing communication lengths. Chapter 6 presents a new load balancing technique that is suitable to workstation cluster platforms. Finally, in Chapter 7 we conclude this dissertation with future research directions.

Chapter 2

Related Work

In this chapter we survey related work in distributed programming. First, we review some of RPC implementations and its variations and other parallel programming paradigms like message passing and Linda. Second, we briefly introduce static program analysis techniques that have been used to establish our source-level transformation framework as a way of implementing RPC statements, and similar researches that focused on function-level parallelism as ours. Finally, we study various load balancing methods and compare them to our dynamic and decentralized load balancing method.

2.1 Remote Procedure Call

Remote procedure call [BN84, Cor91] is a popular paradigm for distributed programming since it simplifies program construction by abstracting away from details of communication and synchronization. However, these early RPC implementations are synchronous in nature, and hence fail to exploit the inherent parallelism in distributed applications. Optimizing RPC performance has been limited to how to efficiently hook in a pairwise sense between client and server communications as shown in the original work by Birrel and Nelson [BN84] and Peregrine high performance RPC system [JZ93].

Since the synchronous property of RPC results in hindrance to parallel executions that can increase the total elapse time, various asynchronous RPC mechanisms have been devised to implement RPC in a non-blocking way [LS88, ATK91, WFN90, GG88]. Call streaming [LS88] is a pioneering work in an asynchronous RPC implementation. A new data type called a *promise* – which is created at the time of a call so that the caller can continue its execution – was designed to support asynchronous calls known as call streaming. It is inspired by Multilisp [Hal85] that is for parallel execution of Lisp programs by means of *future* data type at run-time. However, a static alternative is more attractive because we do not need to rely on new language constructs for parallel execution. Thus far, it has not been sought as a way to improving RPC programs. Remote pipe [GG88] is used to efficiently handle communication patterns of incremental results passing and bulk rate data transfer which are major problem areas in the synchronous RPC communication model. However, the remote pipe can work only for the remote operation that does not expect the return value. In fact, the call streaming approach includes return value streaming as well, thus it should be superior in this sense. Sun RPC system [Cor91] supports three different asynchronous RPC (non-blocking, asynchronous broadcast, and callback RPC) in addition to synchronous one. For those asynchronous RPC mechanisms that support return values, the disadvantage is that the programmer is responsible for claiming the delayed return value by specifying the right location in the program. Unfortunately, if users should choose those 'right' routines for a proper communication style, the RPC paradigm loses its superiority over message passing style programming with explicit send and receive primitives.

Another approach to cure the synchronous nature of RPC is using light-weight threads for RPC calls [BELL90, BELL89, ABLL92, SB90]. When an RPC is invoked, a new thread is created to take a waiting burden for the return value, and the calling process continues its execution. Anderson et al. [ABLL92] reported that user-level light-weight process control is more efficient than kernel-level control [ABLL92].

Special mechanisms need be provided to make RPC possible if servers are replicated, which is an another form of variation. Replicated Distributed System [Coo85], PARPC [MBR87], Marionette [SA89] and MultiRPC [SS86] present mechanisms to call multiple instances of same remote operation in parallel on multiple servers. The caller then blocks until one or all of the requests have been completed. MultiRPC is primary intended for fault tolerance like in invoking replicated file servers, rather than for high performance through parallelism. Unlike general asynchronous RPC systems, these parallel RPC systems can not allow for a client to invoke different kinds of RPCs in parallel because they are simply extended to support server replications.

2.2 Message Passing Systems

Message passing is a capability to explicitly communicate information among simultaneously executing components of a distributed application. Unix sockets are the simplest mechanisms available and they provide basic mechanism underlying systems like PVM [Sun90]. They are the lowest-level primitives provided directly by operating systems so aggressive programmers can achieve significant improvements in the efficiency of the message passing by minimizing software overhead at the expense of additional effort on the part of the programmer. It can be analogous to the trade-off between assembler vs. compiler. Higher level message passing systems include PVM [Sun90], MPI [For93], p4 [BL93] and POLYLITH [Pur94].

PVM PVM [Sun90] (Parallel Virtual Machine) is a user-level code and uses `rsh` commands to initiate daemons on remote machines. The user writes applications as a collection of cooperative *tasks*. Tasks access PVM resources through a library of standard interface routines. These routines allow the initiation and termination of tasks across the network as well as communication and synchronization between tasks. The PVM primitives are intended for heterogeneous operations, therefore include buffering and data encoding and decoding routines. Communication structures include those for sending and receiving data as well as high-level primitives such as broadcast, barrier synchronization and reduction operations. Task (or process) management operations like spawning, killing, initializing are provided as well.

p4 Message passing in p4 [BL93] system is achieved through similar traditional explicit send and receive primitives as other message passing systems. However, the user is responsible for buffer allocation and management. Broadcast, barrier synchronization, global operations are provided. The p4 system supports both the shared-memory model through monitors and the distributed memory model through conventional message passings.

MPI MPI [For93] (Message Passing Interface) is an effort by a group of vendors to consolidate the experienced gleaned from the use of various message passing packages into a standardized system. It is intended to be useful to a wide range of users and efficiently implementable on a wide range of parallel machines. By clearly defining the base set of standardized communication interfaces, many parallel machine vendors can optimally implement those primitive functions for distributed computing. It allows other high-level oriented software packages to use those underlying primitives in order to be portable on various systems as well as efficient.

Polylith POLYLITH [Pur94] system integrated a collection of machine and operating system dependent ingredients for communication into a single entity called a *bus*. In hardware platforms, a bus system simplifies to establish a communication network among many different hardware components like main memories, disks, or I/O devices. Any components that wish to communicate with others simply need to be plugged in to a bus system without having to know the details of other components' details. Similarly, software bus provides an environment where programmers simply need to communicate with message passing *interfaces* and the underlying bus system does the rest of job to accomplish the message passing, which includes data abstraction, communication and synchronization. The POLYLITH system is an implementation of such a software bus that hides compatibility problems from software developers.

2.3 The Linda

Linda [CG89, CG90] is a unique programming system that is based on a special memory model called *tuple space*. It consists of a few simple operations and is orthogonal to the base languages in which it is embedded. Linda memory consists of a collection of logical tuples. Tuples are either active (process tuples) or passive (data tuples). As explained in Chapter 3, Linda model perceives a parallel programming as three basic paradigms of coordinations: agenda, result and specialist. Linda is a programming model that coordinates those three paradigms. One of its realization, C–Linda [CG90] has four basic tuple-space operations as follows.

out(t) causes tuple t to be added to tuple space; the executing process continues immediately. A tuple is a series of typed values, for example, `("a string", 1, 0.17, y)`. **in**(s) causes some tuple t that matches s to be withdrawn from tuple space. For example, **in**`("a string", ?a, ?b, y)` matches the above tuple and the values in the actuals in t are assigned to the corresponding formals in s: i.e. `1` is assigned to `a` and `0.17` is to `b`. If no matching t is available when **in**(s) executes, the executing process suspends until one is, then proceeds as before. If many matching t's are available, one is chosen arbitrarily. **rd**(s) is same as **in**(s), with actuals assigned to formals as before, except that the matched tuple remains in tuple space. **eval**(t) is the same as **out**(t), except that t is evaluated after rather than before it enters tuple space; **eval** implicitly creates one new process to evaluate each field of t.

Linda adopts a fairly high level of abstraction for distributed and parallel programming. How to enable associative memory access in a tuple space is purely an implementation issue. Programmers do not need to pay any attention to distributed data or processes or their interactions. As a result, performance is generally sacrificed for easy programming.

2.4 Program Analysis Techniques for Optimization

Program analysis techniques for program optimization have been successful in various kinds of parallel machines, although most of previous works have done for SPMD (Single Program Multiple Data) style parallel computations. This section briefly summarizes the basic techniques that have been used in our transformation framework.

2.4.1 Basic Definitions

Control Flow Graph (CFG) A *control flow graph* is a directed graph, $CFG = (V_{CFG}, E_{CFG})$, with unique nodes $Entry, Exit \in V_{CFG}$ such that there exists a path from $Entry$ to every node in V_{CFG} and a path from every node to $Exit$; $Entry$ has no incoming edges, and $Exit$ has no outgoing edges. An edge in E_{CFG} is annotated by a control predicate that determines whether or not to take the edge [FOW87].

Use and Def sets Each vertex in CFG has a Def and a Use set associated with it. The $Use(v)$ consists of all variables that are accessed during the computation associated with the vertex v. The $Def(v)$ consists of the variable that is defined at the vertex, if any. A **du-chain** (def-use chain) is the set of uses of a variable associated with each definition of a variable. That is, a **du-chain** allows us to find all tuples that might use the value assigned a variable at a particular vertex in CFG. A **ud-chain** (use-def chain) is the set of reaching definitions associated with each use of a variable [ASU86].

Dominance and Post-dominance Node v *dominates* node w, denoted by $v \Delta w$, if v appears on every path from $Entry$ to w. A node always dominates itself. Node v *immediately* *dominates* node w iff v *dominates* w and there is no node x such that $v \Delta x \Delta w$. In a dominator tree (DT) of a CFG, the children of a node v are all immediately dominated by v. Node v *postdominates* node w, denoted by $v \Delta_p w$, if v appears on every path from w to $Exit$. If v *postdominates* w but $v \neq w$, then v *strictly* postdominates w [ARZ92]. In a postdominator tree (PDT), the children of a node v are all immediately postdominated by v.

Control Dependence A CFG node w is *control dependent* on a CFG node v if both of the following hold [FOW87]:

1. There is a non-null path p: $v \stackrel{+}{\Rightarrow} w$ such that w *postdominates* every node after v on p.
2. The node w does not strictly postdominate the node v.

In other words, w is *control dependent* on v if v can directly affect whether or not w is executed.

Control Dependence Graph (CDG) The *control dependence graph* CDG is a directed graph, $CDG = (V_{CDG}, E_{CDG})$, where the vertices are the same as in CFG and (v, w), for v and w in V_{CDG}, is in the edges E_{CDG} when w is *control dependent* on v.

Data Dependence Vertex v conflicts with vertex w if either v or w share access to a common variable, at least one of which is a “write” operation. Conflicts induce a *data dependence* relation among vertices. If v and w conflict with each other, and if v is reachable from w or w is reachable from v in CFG, we say that v is *data dependent* on w or w is *data dependent* on v.

2.4.2 Previous Works

Many parallelization compilers [KLS$^+$94, HKT92, CMZ92] have a main target of data parallelism at the loop level. Researches on function-level parallelism are relatively rare partly because many scientific problems contain data parallelism at the loop level as shown in a systematic work by Fox et el. [FJL$^+$88]. However, workstation clusters environments provide more necessity to exploitation of function-level or control parallelism because fine grain parallelism is not appropriate due to relatively high communication costs.

Girkar and Polychronopoulos [GP92] uses interprocedural dependency analysis techniques to exploit function-level (task-level) parallelism from ordinary programs written in a serial program model. Task-level parallelism exists across loop and procedure boundaries. Using Hierarchical Task Graph (HTG) as an intermediate parallel program representation, they try to exploit and extract task-level parallelism. To this end, they present how to construct HTG at a given hierarchy level of task and how to derive execution conditions of tasks which maximize task-level parallelism. Supplementally, an optimization algorithm which reduces synchronization overhead with preserving control and data dependence constraints. Finally, parallel source code is automatically generated using `cobegin/coend` parallel constructs and `wait`, `post` and `clear` synchronization primitives: `wait(`a`)` waits on the event a, `post(`a`)` tosses an occurrence of the event a for some other process who is on `wait(`a`)`, and `clear(`a`)` clears all prior posts on the event a.

The PARADIGM compiler project [SLR$^+$95, RB93] also deals with extracting function-level parallelism using MDGs (Macro Dataflow Graphs) representation, which is similar to *HTG*, from the perspective of processor allocation and scheduling problems. When we know the available processor resources in a distributed memory multicomputers in advance, the problem boils down to how to optimally allocate each task node in MDGs to a processor and how to schedule those limited number of available processors.

2.5 Load Balancing Schemes

Load balancing concept has been widely studied from an operating system's concerns [Son94, ELZ86b, KS94, CS93, ELZ86a]. When there are multiple applications that are working simultaneously on distributed environments, some processors can be too heavily loaded while others are not. Dynamic (or adaptive) load balancing, which achieves load balance by migrating excessive tasks from overloaded processor to underloaded one according to the load information of each processor in the middle of computing, is regarded as an effective way in spite of its accompanying overhead, providing that many factors are fairly unpredictable or too complicated to make a best scheduling statically in advance.

Load balancing for multiple sub-tasks generated from a single application has been known as "parallel loop scheduling problems" [TY86, KW85, PK87, TN91, CLZ95], which have been researched as a way of loop parallelization in a shared-memory programming model. If there are I uniform-sized iterations, and P identical processors, load can be balanced simply by assigning I/P iterations to each processor. Since both factors may not be known in advance or may vary substantially, such a static method is often difficult or inefficient. Self-scheduling (SS) [TY86] is the simplest dynamic solution. It assigns a new iteration to a processor only when the processor becomes available. However, this method requires tremendous synchronization overhead; to be practical, hardware support to fast barrier synchronization primitives is desirable. Uniform-sized chunking (CSS) reduces such synchronization overhead by sending K iterations instead of one [KW85]. In this method, the overhead is amortized to $1/K$, but the possibility of load imbalance increases when K is increased. In guided self-scheduling (GSS), the fixed chunk function (K) is replaced with a non-linearly decreasing chunk function in order to reduce the overhead at the beginning of a loop by allocating larger chunks, and also to reduce the chance of load imbalancing at the end of the loop by allocating smaller chunks [PK87]. Trapezoid self-scheduling (TSS) uses a linearly decreasing chunk function, which helps to reduce scheduling overhead while still maintaining a reasonable balance [TN91].

Grimshaw et al. [GWWECL94] presented a static load balancing method for parallel executions in heterogeneous distributed computing systems. The basic idea is to allocate sub-tasks proportionally to the known throughputs of participating workstations. If a size of sub-task as a unit of computation is variable, and even worse, if it cannot be known in advance or the throughputs are variable too, this proportionally allocating approach cannot be effective. Cierniak et al. [CLZ95] deals with the very parallel loop scheduling problem in a version of a network of workstations. They present an optimal algorithm that allocates sub-tasks to all involving processors so that the elapse time can be minimized, although their heterogeneity model is limited to linear one. In other words, the processor speeds, the communication overheads, and the size of sub-tasks (an iteration in a loop), are assumed to be only linearly changeable. Their approach is static as well.

Chapter 3

Distributed and Parallel Programming Structures

Sequential programs have a unique paradigm in the light of module interactions, which is a form of *call* and *return*, within a single process boundary. Distributed programs are inherently parallel. Each module is compiled to run as an independent process. Multiple processes run in parallel for an application. As each module plays its own functional role, the parallelism arising in distributed applications is referred to as *functional parallelism*. An efficient form of interactions among components in a distributed application via remote communications can be constructed differently from application to application. In other words, the nature of an application determines such an 'efficient' form of interactions. Those interactions are summarized by four categories as follows, which have been presented in the previous researches on various parallel execution paradigms [YBS86, BDZ88, Geh84, Geh86, And91a, CG89, LHG86, SA89, Gen81, Geh90].

In the rest of this dissertation, we study how those fundamental parallel program structures can be properly handled during the process of automatic adaptation with optimization.

3.1 Client/Server structure

Figure 3.1 depicts an interaction of the client and server structure where two clients share a server. This is how conventional synchronous RPC [Gib87, BN84] works. As a typical application in this category, let's consider a disk server that repeatedly handles read and write requests from client processes. A scheduling like SST (Shortest Seek Time first), SCAN, and C-SCAN is often used to optimize the moving distance of disk head. Thus, a disk server may include such a scheduling feature. Figure 3.2 illustrates three possible designs. In the first design, a scheduler is separated from the disk server: (1) clients call the scheduler to request an access, (2) the scheduler returns an acknowledgement, (3) clients call the disk server to access data, (4) the server returns the data, and (5) clients finally call the scheduler to release their requests. In other words, three calls are necessary for a disk access, which is tantamount to five message passings — the final call does not need a return value. When the scheduler is an intermediary, we only need four message passings as shown in the

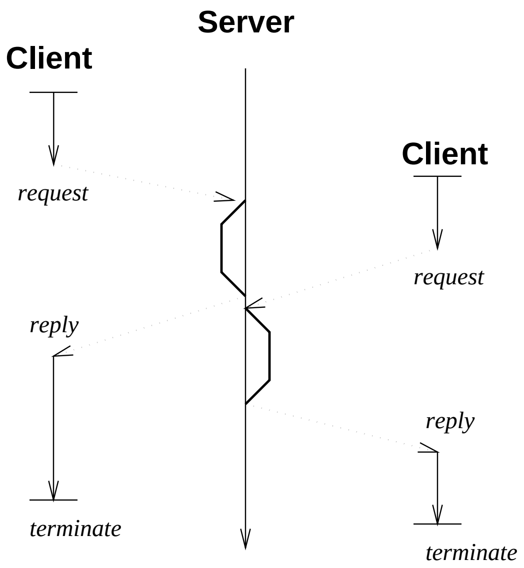


Figure 3.1: Client and Server structure with single threaded server.

figure. Moreover, if we coalesce the scheduler into the server, which is a self-scheduling disk server, the number of needed message passings is reduced to two.

Two observations can be made through this example. First, the two-way communication in client-server computing can simplify the process of writing distributed programs. A client may have other independent computations to the disk access. It is nice if the client can continue to execute those computations during accessing the remote disk. The synchronization restriction imposed by two-way commnunication, which is too conservative for such an application, needs to be safely relieved for higher performance. An automatic program transformation technique is called for. We can see the second design in Figure 3.2 can be automatically transformed from the first one (Chapter 5). Second, while the third design in Figure 3.2 is better than the previous two structures, it is the least tolerant to the change of the scheduling policy. A scheduling policy was initially not a part of a server function, but coalesced for saving unnecessary communications. When a server is likely to be heavily loaded, a certain kind of scheduling may be a performance optimization decision. Moreover, it could be clueless at the time of writing modules that which scheduling policy would be the best. Isolating functionality-independent but performance-affecting factors from the module programming level allows programmers to change those factors freely without having to worry about re-writing burdens.

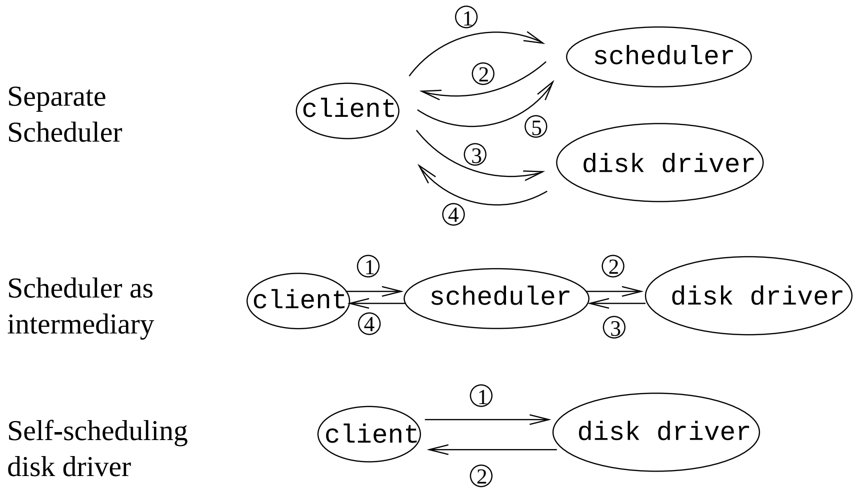


Figure 3.2: Disk scheduling structures.

3.2 Master/Slave structure

This is a variation of client/server computation, through server replication, in order to relieve heavy load at a server side. Replicated servers, called slave processes, are supposed to cooperate to finish a heavy task under the control of a client process, which is called a *master* process. Master process generates many divided sub-tasks and allocates them to multiple slave processes and finally collect the results to get an ultimate result as shown in Figure 3.3.

Many applications can be suitable to this form of computations. It is also known as administrator and workers structure [Gen81]. This category encompasses a well-known *parallel loop* problem in parallel processing [TY86, KW85, PK87, TN91, CLZ95].

Conventional RPC represents client/server computations. Some modifications are required to utilize the replicated servers. PARPC [MBR87] and MultiRPC [SS86] changed the basic RPC mechanism to be able to deal with multiple servers. They distinguish a remote procedure call to a replicated server from a normal RPC, by providing special procedures that programmers need to invoke to process multiple tasks at replicated server sites. As a result, the source code is not transparent for RPCs to replicated servers, which makes modules difficult to be easily configurable.

On the other hand, load balancing is crucial to good performance in parallel processing. The master/worker structure is a strong basis for building parallel applications with good load-balance characteristics. We discuss the general issues on how to write load-balancing intended configuration and what general techniques (e.g. water-marking techniques, scheduling) are available in Chapter 4. Specifically, we present a new decentralized load balancing scheme that is suitable to the workstation cluster environments in Chapter 6.

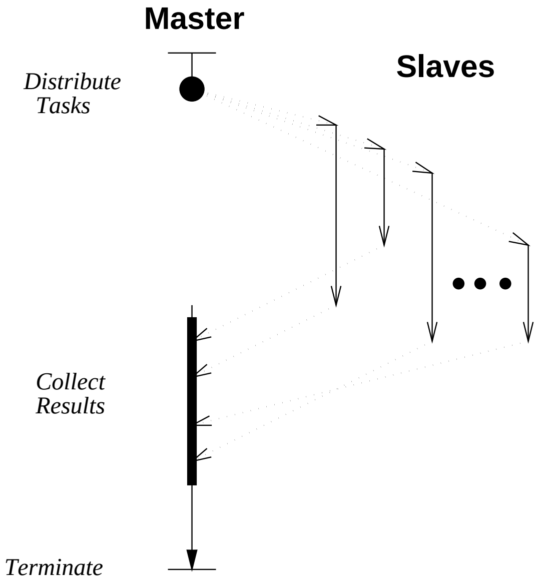


Figure 3.3: Master and Slave structure.

3.3 Pipeline structure

Figure 3.4 shows a pipeline structure in distributed computations. One long computation stage is divided into n sub-stages so that a pipelined parallelism can be exploited. Data pipelining is primarily a network of filters that transforms an input [And91a]. It is often used in data reduction or image processing systems. Since this structure represents a network of filter processes, it is naturally suitable to a network of workstations environment.

Figure 3.5 exemplifies a 3-stage sorting network where each stage can work in parallel. Each filter process accepts two inputs and emits a stream of sorted data. Figure 3.5 (a) shows a data flow in a sorting network of a pipeline structure. For RPC to be used to express the pipeline style computation, the return part of RPC needs to be directed to the next stage of the computation instead of its client side. This can be done by analyzing return path optimization in Chapter 5.

To be precise, as a side note, the merge-sort network in the figure is a hybrid form of pipeline structure and master/server structure. The first stage in the pipeline has four replicated merge modules, and the second has two. Indeed, the merge-sort network is in nature hybrid for parallel version. The composition of various computation structures are natural in writing large distributed programs.

3.4 Data parallel structure

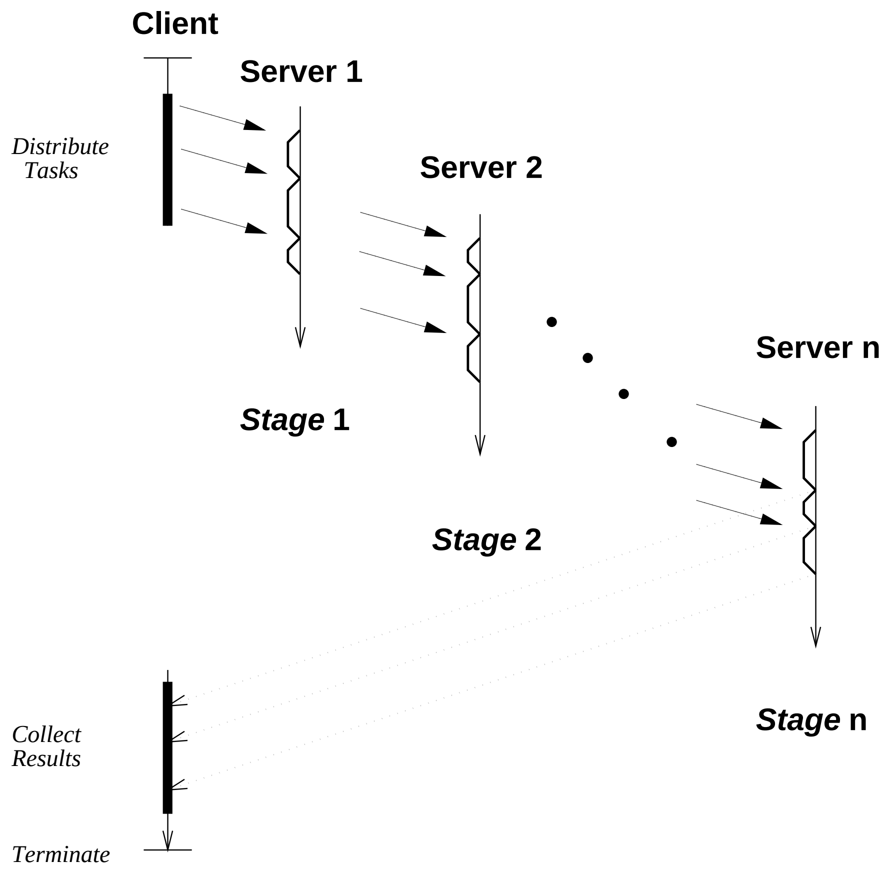
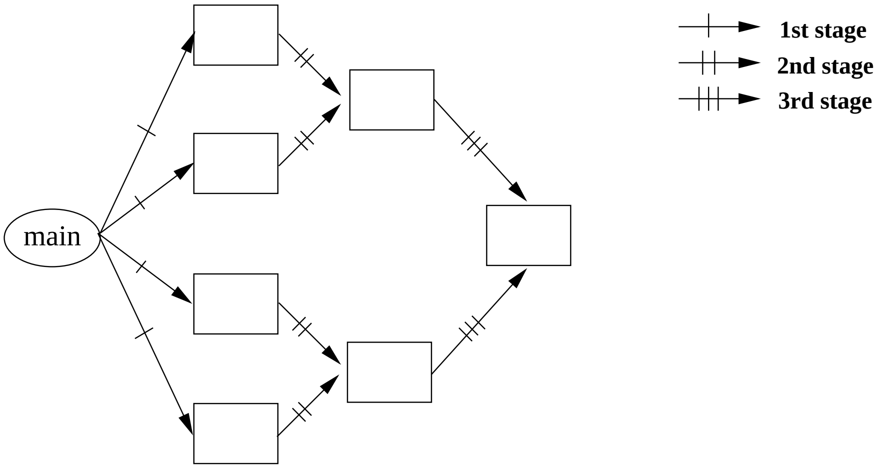


Figure 3.4: Pipeline structure.

Figure 3.6 illustrates a data parallel computation. Unlike previous structures each of the processes is executing the same program, where the individual processes work on a different set of data.[1] It is also called SPMD (Single Program Multiple Data) programming. When a process needs to access non-local data that are stored and maintained by other processes, they communicate with each other.

The communication patterns among processes are expected to be highly structured and often predictable so that the entire data can be decomposed before starting computations for the good performance. Initial data layout and communication optimization [vHK94, HQ91, AL93] are hot issues in this kind of computations and they are mutually related. Many practical scientific problems except fairly irregular ones are known to be efficiently computed under this structure [FJL+88]. As scientific problems are main targets for this style, parallel variants of Fortran programming language — like Fortran D [HKT92], Vienna

[1] Although the previous merge-sort example uses an identical merge module in its parallel computation, this is not because of the enforcement of the structure but because of user's liberty to do so.



Pipelined structure of 3-stage merge sort network

```
merge_main(DATA)
char *DATA
{
    Split DATA stream into 4 data streams;

    Sort each data streams;

    /* construct merge network */
    /* first stage: */
    out1 = merge1(data1, data2);
    out2 = merge2(data3, data4);
    out3 = merge3(data5, data6);
    out4 = merge4(data7, data8);

    /* second stage: */
    out12 = merge5(out1, out2);
    out34 = merge6(out2, out3);

    /* third stage: */
    out1234 = merge7(out12, out34);
}
```

```
char *
merge(strm1, strm2)
char *strm1, *strm2;
{   int i = 1, j = 1, k = 1;
    char outstrm[MAX];

    while ( (i <= strm1[0]) || (j <= strm2[0])) {
        if (i <= strm1[0] && j <= strm2[0]) {
            if (strm1[i] <= strm2[j])
               outstrm[k++] = strm1[i++];
            else outstm[k++] = strm2[j++];
        }
        else if (i <= strm1[0])
            outstrm[k++] = strm1[i++];
        else
            outstrm[k++] = strm2[j++];
    }
    outstrm[0] = strm1[0] + strm2[0];
    return(outstrm);
}
```

Figure 3.5: Merge sort network.

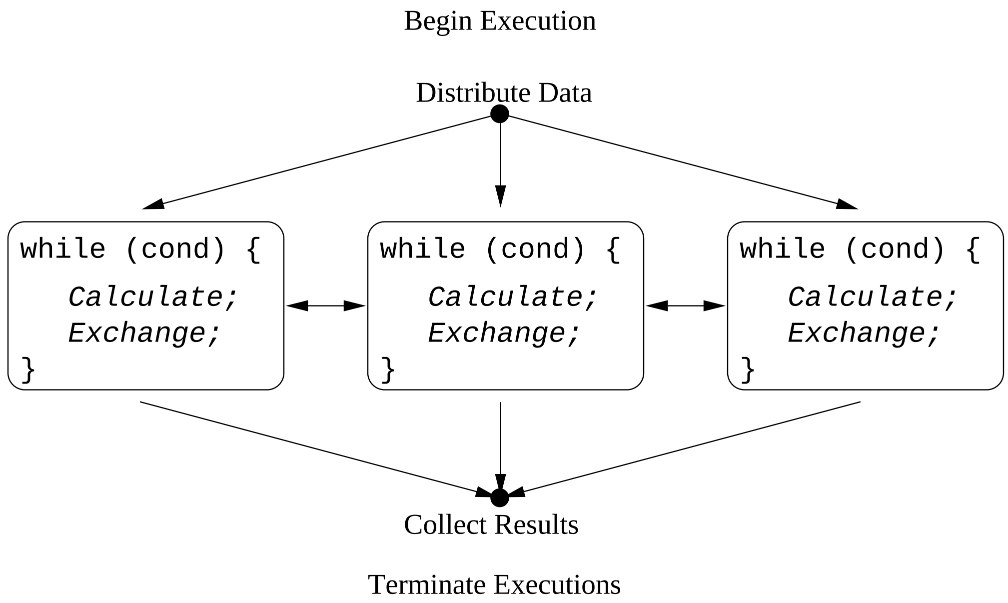


Figure 3.6: SPMD (Single Program Multiple Data) structure.

Fortran [CMZ92] and HPF (High Performance Fortran) [KLS+94] are popular. The primary works are concentrated on how to optimize such a Fortran program using program analysis techniques.

Many research works have been done for this structure of parallel computations on distributed memory parallel machine environments. Same techniques can be applied for workstation cluster environments. Workstation clusters are by nature distributed memory machines with slightly different communication parameters. Thus, the dissertation research does not focus on exploiting data parallelism, while those program optimization techniques can be adopted as one of infra-structures in our optimization framework that is aimed at multi-paradigm distributed and parallel applications on workstation clusters.

3.5 Remarks

Andrew [And91a, And91b] classified distributed and parallel program structures based on the behavioral type of a process component in a program rather than the type of interaction among its components as given in the chapter. The four basic types are ‘filter’, ‘client’, ‘server’ and ‘peer’ processes. A filter is a data transformer. It receives streams of data values from its input channels, performs some computation on those values, and sends streams of results to its output channels. Because of those attributes, we can connect filter processes into networks that perform larger computations. A client is a process that triggers a service request and a server is a process that reacts for the service request. A client thus initiates an activity; it then delays until its service has been serviced and returned. A server waits for requests to be made, do the services, and finishes by replying the results. A server is often a non-terminating process and often provides service to more than one client. A peer is one of a collection of identical processes that interact to provide a service or to solve a large problem (SPMD) or several peers might interact to solve a parallel problem, with each solving a piece of the problem (master-slave). For example, the types of processes in client and server structure are of client and server. Pipeline is of filter. Master/slave and SPMD are of peer.

Another classification on paradigms of parallel computations is given by the Linda [CG90] project: result, specialist, and agenda parallelism. Result parallelism focuses on the shape of result in order for process interactions. Specialist parallelism focuses on the makeup of the work crew that are specialists for particular jobs. How to divide a job and to design a set of specialists who will take care of those distinct small pieces of the job are the issues in this paradigm. Agenda parallelism focuses on the list of tasks to be performed. Result parallelism is somewhat unique because it no longer presumes that each process has its own local data and non-local data access through message passings. It is a shared data object. Processes' efforts to read or write those data are controlled in regards with the critical section problem. The remaining two paradigms are easily constructed through the basic four processes — specialist by filter, client and server and agenda by peer processes.

Andrews [And91a, And91b] presented typical distributed and parallel applications that can be optimally expressed as one or combination of the four process types. A versatile programming environment for all of the forms of parallelism or process can be provided by message passing paradigms like PVM [Sun90] and MPI [For93]. Not only because of the resulting complexity in writing distributed programs with such a low-level abstraction, but also because of its difficulty to reconfigure, a higher level of abstraction for module interaction is called for.

Procedure call abstraction has been favored since early programming era because it contributes to construct a well structured modular program, which allows to reuse existing modules and helps write and maintain a large program by giving a clear view of its structure. The RPC paradigm adopts a widely used and understood procedure call abstraction as the sole mechanism of remote operations; thus it simplifies distributed programming by abstracting from details of communication and synchronization.

Since those conventional implementations of RPC paradigm [BN84, Gib87] only allow client and server types of processes, the RPC paradigm has a limited coverage of applications especially for high performance. Indeed, any single paradigm is not enough for high performance. That is why Linda has three paradigms for writing a program based on the data sharing in the form of tuple space. And that is why Andrews [And91a] suggests to use different types of processes for different types of applications. To strike a compromise between the pros and cons in the RPC paradigm, and to allow various structures in a distributed application using reusable modules, we need an automatic adaptation and optimization process for distributed programs based on RPC paradigm.

Chapter 4

Configuration-Level Performance Programming

In this chapter we address writing a MIL-style configuration program that orchestrates modules in an optimal program structures for a specific application. We first illustrate a concrete example that motivates the optimization of RPC-based distributed programs for high performance. The example we will discuss involves DNA sequences, an conceptually straightforward problem whose solutions, though very intricate in implementation, are simple and may admit several types of parallelism.

This is essentially a data structure problem: when a new DNA sequence is discovered, geneticists want to find out how and which previously known sequences the new one resembles. Suppose we have tens or hundreds of newly discovered sequences that are to be compared to a large database of existing sequences. Suppose the length of each sequence is variant, and so is the comparison time. Figure 4.1 (a) is a client (or master) module that initiates the required number of comparison tasks.

Two basic parallelizing approaches to the DNA example illustrate the problems that we are dealing with: one approach performs many sequential comparisons simultaneously as shown in Figure 4.1 (b), which is a master/slave model (target database is replicated to each server), and the other constructs a pipeline of a series of sub-comparison modules that do comparisons to a part of the entire database (database is divided into smaller ones) in Figure 4.1 (c).

The problems for this example, which make direct use of conventional RPC inappropriate to high performance distributed computing, may be summarized as follows:

1. **Load balancing:** Server replication is a basic way to improve throughput. However, the performance of a replicated server can be degenerated to that of the bottleneck process or processor unless a proper load balancing scheme is used. In Figure 4.1 (b), no slaves should be idle while others are busy. So far, RPC in itself does not make any association with load balancing. Previous RPC systems for multiple servers like PARPC [MBR87] and MultiRPC [SS86] do not consider load balancing.

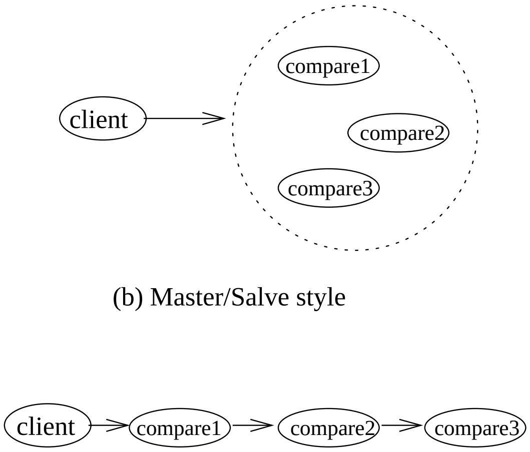
2. **Scheduling:** In our example, the length of each DNA sequence varies, so does comparison time. In this situation, if the longest sequence is assigned to an unfortunate

```
client()
{
  /* get next sequence to compare */
  for (i = 0; i < NUM_NEW_SEQUENCE; i++)
       seq[i] = get_next_seq();

  /* compare a sequence with each sequences */
  /* in a database */
  for (i = 0; i < NUM_NEW_SEQUENCE; i++)
       result[i] = compare(seq[i]);

  /* update result */
  for (i = 0; i < NUM_NEW_SEQUENCE; i++)
       if (real_max < result[i].max) {
             real_max = result[i].max;
             real_max_id = result[i].db_id;
       }
}
```

(a)



(b) Master/Salve style

(c) Pineline style

Figure 4.1: Simple DNA sequence search.

process at a late time near the end of all computations, only that process will be busy while others sit idle. This problem can be solved if the longest sequence is serviced first. To do this, the RPC server must be constructed to service tasks with respect to their given priorities.

3. **Parallelism:** RPC is synchronous in nature. A client must wait to get a response for its call before calling another server. Preparing multiple servers or multi-stage pipelines may not be of much use if a synchronous RPC is used for remote interaction as then only one server may be activated by a client. Parallelism can be sought if the gap between send and receive primitives is widened to allow more useful computations during the wait for a result.

4. **Length of communication paths:** RPC can lengthen communication paths unnecessarily if involved modules form a computation network (like the *trellis* model in Chapter 8 of [CG90]) because of its two–way communication protocol. For instance, in Figure 4.1 (c), an intermediate result in each stage of the *compare* module must go back to the client first before being delivered to the next stage. An optimization step that eliminates such unnecessary communication paths is called for.

This example illustrates the several dimensions open to programmers, and serves to help us state simply our objective: since each of the above types of improvement admits several strategies for success, and also each can be characterized in terms of the application's configuration level description, we seek a development environment where developers may implement modules in terms of RPC interfaces (which are comparatively simple constructs),

```
module client {
   source = "C" "local"::
   entrypoint = "main"::
   use interface compare
      : pattern = {string}
      : accepts = {integer}
      : interface = "stdio"::
}
```

```
module server {
   source = "C" "remote"::
   define interface compare
      : pattern = {string}
      : returns = {integer}
      : interface = "null"::
}
```

```
module DNA_seq_search {
   instance client::
   instance server: standalone::
   bind client compare server compare::
   interface = "stdio"::
}
```

Figure 4.2: Basic configuration for DNA sequence search example.

yet separately be able to express performance improvement strategies in terms of the configuration description. Figure 4.2 shows the basic configuration program for the example of Figure 4.1; it represents (in the notation of our system to be described) the conceptual starting point for configuration programmers who wish to experiment with different optimization techniques. After programmers express directions in terms of this configuration, the system should tailor all executables to be consistent with both specifications.

4.1 Requirements For Configuration Optimization

We exposed some limitations of using RPC for high performance distributed programs, and in doing so suggested some dimensions by which improvement can be achieved. This also makes it clear that we can separate what programmers should be able to do and what tools can do as follows:

1. High-level decisions regarding performance factors that affect overall performance should be specified in the programming-in-the-large level so that module reusability can be enhanced, especially in the process of performance tuning. Programmers should be able to specify those decisions independently.

2. High-level decisions regarding performance factors should be automatically realized and optimized with low-level message passing primitives.

The purpose of this section is to discuss in greater detail the various strategies by which performance can be improved by configuration level annotation. This will identify which features will be used for optimal realization of RPC (Section 4.1.1) and expression of the load balancing scheme (Section 4.1.2).

4.1.1 Performance Factors

Performance benefits are realized as latency and throughput improvements. A distributed program is composed of clients, servers and their interactions. We distinguish the task of performance improvement of a distributed program from the perspectives of its three components. Namely, clients should be able to make multiple requests (parallelism), load must be balanced among servers (load balancing), and interprocess communication and its overhead must be minimized (communication optimization). We will elaborate on factors that affect performance and what we can do to improve performance in the followings. All of these factors are related in module interactions rather than functionality; thus they will be represented at the interconnection programming level. Throughout this section, the expressions enclosed by an oval box denote our extension of the original MIL specification in `polygen` [CP91] for a performance configuration.

Calling Style

A synchronous call is a call whereby the client blocks the call until the server completes it [BN84]. An asynchronous call does not block the client, and replies can be received as they are needed. To date, the decision on calling style is not the programmer's (for example, calls may be synchronous only [BN84] or they may be asynchronous only [ATK91, LS88, WFN90]), or the decision has to be made at module programming level by use of different library routines [Cor91]. If we let this decision be separate from RPC statement, the modules will remain reusable for different calling styles. Therefore, in devising requirements for a configuration level optimization system, an asynchronous RPC should be implemented by separating the *Send_Request* primitive and the *Receive_Result* primitive to allow other useful operations in the midst of remote service. Synchronous calls would be implemented by their placement in sequence in a client module. Thus, a server module does not distinguish whether a server is called synchronously or asynchronously. It implies that the same server can be called asynchronously for one client and synchronously for another client in the same application. The calling style should be easily prescribed by programmers in terms of a **use** clause in the module specification. Consider the module *client* in Figure 4.2, which calls the remote procedure *compare*. To specify an asynchronous call, programmers may simply state so in the MIL specification as follows:

```
use interface compare
:    callstyle = "async"
:    pattern = { string }
:    accepts = { integer } ::
```

Servicing Style

When the length of a service queue is long, throughput can be improved by the choice of a good servicing style. Servicing style can be characterized by two factors: scheduling policy and server replication. Scheduling policy determines the desirable order of requests to be serviced. Usually the order of service is fixed by arrival time. Scheduling generalizes the order – i.e. other parameters besides arrival time are considered to determine the order of service. For example, the length of a DNA sequence to be compared may be a parameter that determines such an order as mentioned before. Server replication improves throughput as well because the load is distributed among replicated servers, although load balance is crucial to good performance.

As with calling style, the module specification for expressing scheduling and replication features should be simple for programmers to assign. Illustrating one way this might appear for the introductory example, is

```
module server {
    source = "C" "remote" ::
    define interface compare
    :  priority = "strlen(x)"
    :  replication = "harvey.cs.umd.edu,..."
    :  pattern = { string }
    :  returns = { integer }
    :  interface = "null" ::
}
```

Here the **priority** attribute is an expression, which would use valid syntax within the module *compare* in order to evaluate a priority. Since we hoped to assign a higher priority to the longer sequence, evaluating `strlen(x)` produces the right order of priorities. The *compare* module is replicated in its simplest form here, while load balancing will be considered in Section 4.1.2.

The **priority** expression is directly used to evaluate a priority for the corresponding service request when a server stub is generated. So, it should have a legal expression in a module language. The variables used in the expression should also be defined in the server module. For more examples, when `head`, `loc` and `dir` are variables used in a disk server for the current head location, the location of the requested data and the current moving direction of the head, the **priority** expressions for SSTF (Shortest Seek Time First), SCAN and C-SCAN are `"abs(loc - head)"`, `"(head - loc) * dir"` and `"loc * dir"`, respectively.

The **replication** attribute contains a list of machine names on which the server is replicated.

Communication Style

A communication pattern in distributed programs occurs in three different forms: intermittent, incremental and bulk rate data transfer. A conventional RPC protocol covers only the case of intermittent data transfer, i.e. when the number of messages between client and server is not too big or too frequent. An incremental pattern of communication occurs when we try to exploit pipeline concurrency for a chain of clients and servers as in Figure 4.1 (c) and Figure 4.3. This pattern, which forces a single computation to be decomposed into a series of distinct RPCs, reduces the server's performance since it is inactive between calls unless the synchronous behavior of RPC has been changed. Also, if we want to send bulk data by a series of RPCs, the communication performance is severely limited since it is not possible to aggregate data of successive procedure calls from a single client. Even worse, contemporary RPC systems are optimized to transmit limited amounts of data (usually less than 10^3 bytes) per call. To support the incremental and bulk rate data transfer, wherein conventional RPC systems performance suffers severely, a new communication model called *remote pipe* [GG88] has been devised. In the framework we are motivating, these patterns may be efficiently handled with automatic communication optimization if programmers specify which communication pattern will appear.

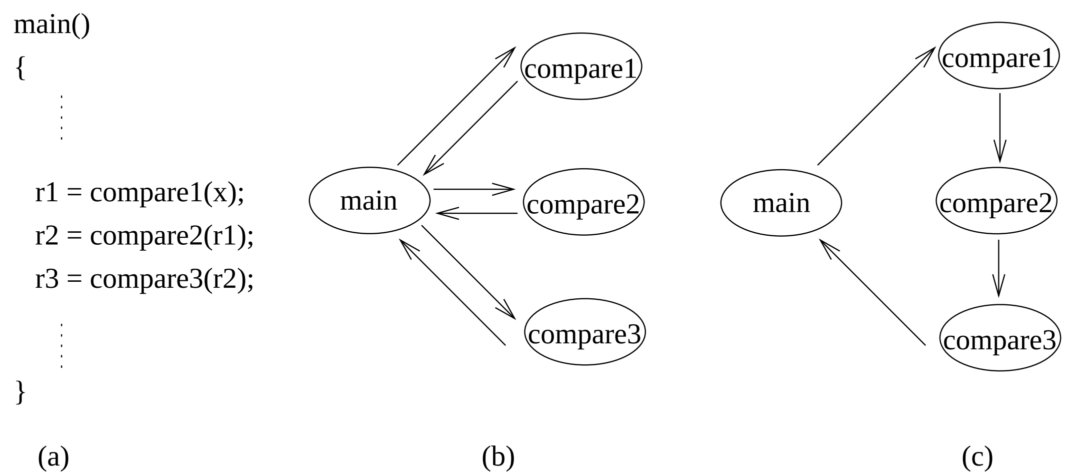


Figure 4.3: Communication optimization for Figure 4.1 (c).

Once that information has been provided, there would be three ways to improve communication performance: (1) choice of proper transport, (2) reduction of kernel overhead by data aggregation and (3) elimination of unnecessary communication. The best transport protocol depends on the amount of data to be transferred. In other words, the connectionless transport protocol (UDP: User Datagram Protocol) works best for the intermittent data transfer pattern, and the connection-oriented transport protocol (TCP: Transmission Control Protocol) for the incremental and bulk rate data transfer pattern. Data aggregation allows us to amortize the overhead of kernel calls. If the size of aggregated data is increased, the throughput is increased, and if it is decreased, then the latency is reduced. Programmers can control high throughput vs. low latency by assigning the size of aggregated data to a particular server.

Unnecessary communication is unavoidable in conventional RPC implementation as illustrated in Figure 4.3. Figure 4.3 (b) is optimized to (c) by elimination of the unnecessary communication paths. The communication optimization like unnecessary communication paths elimination can be accomplished by flow analysis techniques. Chapter 5 presents a source-level transformation algorithms with communication optimization.

4.1.2 Load Balancing

Since load balance is crucial to good performance among replicated servers, we need to provide a systematic way to customize proper load balancing schemes for an RPC to replicated servers. Load balancing schemes can be classified as follows. In this section, we present how to express various kinds of load balancing schemes with various load balancing factors.

Type 1: Static load distribution

Static load distribution is a simple approach to load balancing. The tasks generated by master process are distributed to the pool of slave processes according to the statically defined task distribution ratio, which is decided by programmers based on the average performance of participating workstations. The task distribution ratio is the only parameter in this scheme. The approach by Grimshaw et al. [GWWECL94] belongs to this type. Since load distribution is a client side concern, an attribute **loadratio** is needed in the **use** clause. The ratio description is matched with the **replication** attribute in the corresponding **define** clause as follows:

use interface `compare`	**define interface** `compare`
: **callstyle** = `"async"`	: **replication** = `"harvey.cs.umd.edu,`
: **loadratio** = `"1:2:3"`	`bugs.cs.umd.edu,`
⋮	`thumper.cs.umd.edu"`
	⋮

Type 2: Demand-driven load distribution

Simple dynamic load balancing can be achieved through demand-driven load distribution, which does not need to migrate tasks among slaves. When a master process receives a result from a slave, it sends another task to the slave as the load situation of the slave has decreased due to the recent finish, i.e. receiving a result is regarded as a demand for another task. This scheme contains two problems. First, the master process can generate a bottleneck [GBSS89]. For example, if there are 100 slaves and a master needs 10^{-2} second to prepare and send a task, the master would create a bottleneck unless the average time for each slave to finish a task is greater than a second. Furthermore, if all slaves took the same amount of time to finish their own tasks, the finish replies would come in burst, and this would cause a bottleneck, too. Second, the scheme does not allow overlap between communication and computation because the next task can not be issued unless the current one has been finished.

To alleviate these problems, water-marking can be used [CG90]. The idea is to maintain the number of tasks between an upper and a lower limit. The upper water-mark limits the maximum number of queuing tasks to a particular slave, thus prevents possible overload. The lower water-mark is used to maintain the minimum number of tasks to cope with network latency, which makes a slave sit idle between a finish of a task and a wait for another one. This requires a change in calling style, represented by "`async-demand(`L`:`U`)`", where L and U are a lower and an upper water-mark, respectively; '*' denotes an unspecified water-mark. Both of L and U cannot remain unspecified. A master sends a task to a server only if the number of queued tasks to the server is less than U. If the number is less than L, it needs to send a number of tasks to make it L. Available servers are the servers of which queue lengths are less than U. The demand-driven load distribution can be represented with proper water-mark ranges as follows:

module `client1` {
 ...
 use interface `compare`
 : **callstyle** = `"async"`
 : **loadratio** = `"1:1:1"`
 ...
 }
 module `client2` {
 ...
 use interface `compare`
 :**callstyle** = `"async-demand(*:5)"`
 ...
 }
	module `server` {
	 source = `"C" "remote" ::`
	 define interface `compare`
	 : **priority** = `"strlen(x)"`
	 : **replication** =
	 `"harvey.cs.umd.edu,`
	 `bugs.cs.umd.edu,`
	 `thumper.cs.umd.edu"`
	 : **pattern** = { `string` }
	 : **interface** = `"null" ::`
	 }

Type 3: Dynamic load balancing

When load balance cannot be reached through the above load distribution methods, tasks should migrate, which is known as dynamic or adaptive load balancing. Many dynamic load balancing algorithms [CS93, ELZ86b, KS94, LK87, Son94] have been devised for good load balance with less migration overhead; they are characterized by the following parameters which distinguish them. Load balancing algorithms can be fine tuned when programmers can change those factors conveniently.

- **Topology:** Topology determines the shape of task migration paths. A fully connected topology provides a way to gain load balance in any case, but with some system overhead due to periodic load state exchange. The overhead can be cut through simplified topology. A compromise must be sought between reduced overhead and load balancing gains. In Chapter 6, we present such a topology.

- **Transfer policy:** Transfer policy determines whether load has to migrate at a particular load state. The decision can be made based on local or global load information.

- **Location policy:** Location policy determines which process initiates the migration and which process should be the source or the destination in this migration: sender-initiated, receiver-initiated or symmetrically-initiated.[1]

- **Selection policy:** Selection policy determines how many work load units are to migrate.

The following generic form of a **replication** expression can express the above information, where M_1 and M_2 are IP addresses of host machines, o_i and u_i are logical expressions that denote the conditions to be overloaded and underloaded for M_i, and γ_{12} and γ_{21} are the fraction of current load to be migrated from M_1 to M_2 and M_2 to M_1, respectively. The symbol # denotes a migration linkage. The load balancing information must be given to all load migration paths in an application:

:replication = "$M_1(u_1, o_1)[\gamma_{12}]\#M_2(u_2, o_2)[\gamma_{21}]$"

Table 4.1 illustrates various kinds of load balancing expressions and their meanings. Types of $M_1()\#M_2(u_2, o_2)$, $M_1(u_1)\#M_2(u_2, o_2)$, $M_1(o_1)\#M_2(u_2, o_2)$, $M_1(u_1, o_1)\#M_2(o_2)$, $M_1()\#M_2(o_2)$, $M_1()\#M_2(u_2)$, $M_1(u_1)\#M_2(o_2)$, are omitted for brevity, because each of them is dual to one of the cases in the table. To denote a symmetrically-initiated case, both of the source and the destination of the load migration are explicitly specified in the *Comments* column; e.g. M_1 carrys load out to M_2. For a receiver-initiated case, only the destination is explicitly specified; e.g. M_1 carrys load in. For a sender-initiated case, only the source is explicitly specified; e.g. M_1 carrys load out. Followings are the summary of those types:

- Type 1 is the case where only M_1 initiates migration as a sender.

- Type 2 is the case where only M_2 initiates migration as a receiver.

- Type 3 is a mixed case where M_1 initiates as a sender and M_2 does as a receiver.

- Type 4 is a purely receiver-initiated case.

[1]With a sender-initiated policy, an overloaded process will look for a destination to export load to. With a receiver-initiated policy, an underloaded process will look for a destination to import load from. Symmetrically-initiated process can play both roles depending on whether it is underloaded or overloaded.

- Type 5 is a purely sender-initiated case.
- Type 6 is the case where only M_1 initiates migration as both a sender and a receiver.
- Type 7 is an asymmetric case where M_1 can be either a sender or a receiver, but M_2 only can be a receiver.
- Type 8 is a fully symmetrically-initiated case where both M_1 and M_2 can initiate migration as either a sender or a receiver.
- Types 1 through 3 have uni-directional migration paths.
- Type 4 through 8 have bi-directional paths.

	M_1	M_2	*Direction*	*Comments*
1	o_1	–	$M_1 \xrightarrow{o_1} M_2$	M_1 carrys load out if o_1 holds.
2	u_1	–	$M_1 \xleftarrow{u_1} M_2$	M_1 carrys load in if u_1 holds.
3	o_1	u_2	$M_1 \xrightarrow{o_1 \vee u_2} M_2$	M_1 carrys load out if o_1 holds or M_2 carrys load in if u_2 holds.
4	u_1	u_2	$M_1 \xleftarrow{u_1 \wedge \neg u_2} M_2$ $M_1 \xrightarrow{\neg u_1 \wedge u_2} M_2$	M_1 carrys load in if $u_1 \wedge \neg u_2$ holds. M_2 carrys load in if $\neg u_1 \wedge u_2$ holds.
5	o_1	o_2	$M_1 \xrightarrow{o_1 \wedge \neg o_2} M_2$ $M_1 \xleftarrow{\neg o_1 \wedge o_2} M_2$	M_1 carrys load out if $o_1 \wedge \neg o_2$ holds. M_2 carrys load out if $\neg o_1 \wedge o_2$ holds.
6	u_1, o_1	–	$M_1 \xrightarrow{o_1} M_2$ $M_1 \xleftarrow{u_1} M_2$	M_1 carrys load out if o_1 holds. M_1 carrys load in if u_1 holds.
7	u_1, o_1	u_2	$M_1 \xrightarrow{o_1 \wedge u_2} M_2$ $M_1 \xleftarrow{u_1 \wedge \neg u_2} M_2$ $M_1 \xrightarrow{\neg u_1 \wedge u_2} M_2$	M_1 carrys load out to M_2 if $o_1 \wedge u_2$ holds. M_1 carrys load in if $u_1 \wedge \neg u_2$ holds. M_2 carrys load in if $\neg u_1 \wedge u_2$ holds.
8	u_1, o_1	u_2, o_2	$M_1 \xrightarrow{o_1 \wedge u_2} M_2$ $M_1 \xrightarrow{\neg u_1 \wedge u_2} M_2$ $M_1 \xleftarrow{u_1 \wedge o_2} M_2$ $M_1 \xleftarrow{u_1 \wedge \neg u_2} M_2$	M_1 carrys load out if $o_1 \wedge u_2$ holds M_1 carrys load out if $\neg u_1 \wedge u_2$ holds. M_1 carrys load in if $u_1 \wedge o_2$ holds M_1 carrys load in if $u_1 \wedge \neg u_2$ holds.

Table 4.1: Various load balancing expressions and their meanings.

As an example, a server module specification is given below, which forms a dynamic load balancing scheme that has a circular topology and sender-initiated task migration policy.

```
                     Makefile client.cl server.cl all.cl
a.cfg                x.client.c x.client.h
client.c     ⟹       x0.server.c x0.server.h x1.server.c
server.c             x1.server.h
                     x2.server.c x2.server.h x3.server.c
                     x3.server.h
```

Figure 4.4: Generated files from user provided modules using CORD.

```
module server {
   source = "C" "remote" ::
   define interface compare
   :  priority = "strlen(x)"
   :  replication =
        "harvey.cs.umd.edu(o=(L>=10))[1/2]#bugs.cs.umd.edu,
         bugs.cs.umd.edu(o=(L>=20))[1/3]#thumper.cs.umd.edu,
         thumper.cs.umd.edu(o=(L>=10))[1/4]#harvey.cs.umd.edu"
   :  pattern = { string }
   :  interface = "null" ::
}
```

4.2 Developing Applications In CORD

The previous section has characterized the various forms of optimization which are possible to discuss in terms of an application configuration. We have developed a support environment called CORD (Configuration-level Optimization for RPC-based Distributed programs) to allow us to experiment with introduction of such adaptations at low cost.

The configuration language chosen for expressing modules and their compositions is derived from the POLYLITH module interconnection language (MIL), and the distributed run time environment chosen is likewise the software bus behind POLYLITH. Basic tools for preparing applications to run in this environment are already available within the `polygen` system [CP91], although they are to be tailored to attain our source translation (rather than stub generation) principle. Therefore the principle thrust of our effort has been to add a source translator (**gen_trans**) to the suite of `polygen` tools. The source translator operates differently depends on whether a module is a client or a server from given RPC's viewpoint: for a client module, it performs data flow analyses to place message passing primitives optimally, and for a server module, it generates proper codes to implement particular servicing styles described in configuration programs. The next chapter presents the heart of algorithms for this purpose.

The development of an application in CORD consists of a number of steps. At some point, each module used in the application must be given an implementation, each dealing with interfaces in generic RPC terms, of course. Since performance decisions that occur in module interactions are decoupled from the module programming level, module functionality is the only concern in this step.

The second step is to define an application using the module interconnection and performance configuration. In the next step, CORD generates all necessary files for an executable automatically with respect to the configuration program. Figure 4.4 shows the automatically generated files from the user provided files, which are source programs `clnt.c, srv.c` and the configuration program `a.cfg`, using CORD. (In the figure, it is assumed that the `srv.c` is replicated to four distinct machines.) This step follows the similar packaging process in `polygen`, which deals with automatic adaptations for divergent structural and geometric configurations. The interaction between modules in distinct sites, which is an RPC, is resolved by generating client and server stubs automatically by `polygen`. CORD does not generate stubs but translates source codes in which every RPC is replaced with a set of message passing primitives interspersed for the purpose of optimization. The script of the entire process, which includes both user commands and the execution of the configuration program, is shown in Figure 4.5. The tools that are involved in this process, are summarized as follows:

- **config** generates prolog assertions (`a.pl`), which encode facts about the modules and bindings in the configuration, from user provided configuration (`a.cfg`).

- **prolog**: After reading the assertions (`a.pl`), the CORD uses **prolog** inferencing mechanism to search for satisfying the goal, which asks the possibility to create an application for the configuration described in `a.cfg` by means of the available tools in the environment. This inference results in a package information (`a.pkg`) if successful.

- **gen_imake**: Using the package information (`a.pkg`), **gen_imake** generates an `Imakefile` to create a `Makefile` for an application. A UNIX `imake` is used to generate a `Makefile` from a provided template in CORD.

- **gen_module** generates a MIL program (`.cl`) for the module descriptions[2].

- **gen_cluster** generates a MIL program (`.cl`) for the application description.

- **gen_header** generates a header file for each module if necessary.

- **gen_trans** generates translated source code to realize RPCs using message passing primitives, and proper codes for scheduling and/or load balancing.

[2]The components of the MIL program are the module descriptions and the application description. See [Pur94].

`% config < a.cfg > a.pl`	*Initially a user has source codes ("client.c","server.c") and a configuration file ("a.cfg").*
`% prolog < a.pl > a.pkg`	*Creates prolog assertions for the configuration. Given inference engine ("package.pl"), this generates the packaging information ("a.pkg") to satisfy the packaging goal according to the generated assertions.*
`% gen_imake < a.pkg > Imakefile`	*Creates Imakefile file from the packaging information.*
`% imake -T "Imake.tmpl"`	*Creates Makefile file using a prepared Imake template ("Imake.tmpl").*
`% make`	*Creates executables according to the interface generation, source transformation, and compilation information in the Makefile. The following output is from commands in Makefile.*
`gen_header client<a.pkg>client.h`	*generates a header file for the client.c.*
`gen_header server<a.pkg>server.h`	*generates a header file for the server.c.*
`gen_trans client.c<a.pkg>x.client.c`	*translates from the original "client.c".*
`gen_trans server.c<a.pkg>x.server.c`	*translates from the original "server.c".*
`gen_module client<a.pkg>client.cl`	*generates the client specification.*
`gen_module server<a.pkg>server.cl`	*generates the server specification.*
`gen_cluster < a.pkg > all.cl`	*generates the application specification.*
`csc client.cl`	*compiles the client spec. into client.co.*
`csc server.cl`	*compile the server spec. into server.co.*
`csc all.cl`	*compiles the application spec. into all.co.*
`csl -o all client.co server.co all.co`	*creates a root executable that executes client.*
`cc -o client x.client.c -lith`	*compiles and creates a client executable.*
`cc -o server x.server.c -lith`	*compiles and creates a server executable.*

Figure 4.5: Script for the design (user commands prefixed by a % prompt).

The final step is to execute the application, identify performance bottlenecks using a performance measurement tool, and repeat the process from the second step until the resulting performance is satisfactory.

It is possible to suggest the potential for CORD in helping programmers to discover desirable optimization opportunities at low cost. We do illustrate this using the Mandelbrot example, using a generically coded C implementation built in the Polylith system. In this implementation, a sub-task is to compute the set for one row in 200×200 pixel window, therefore 200 RPCs will be made to complete the whole computation. This formulation of the problem increases traffic beyond that of alternative implementations, but makes the effect of any optimization strategies more easily measured for illustration.

Table 4.2 shows timing results when we execute this Mandelbrot program for various performance improvement alternatives, where the programmer may select each mechanism by making only a simple attribute change in the module specification as in Figure 4.6. Table 4.2 (a) compares the performance between synchronous and asynchronous RPC where the computation is run on each of several different servers in turn. (To be concrete, '`harvey`' is SparcStation IPC, '`rimfire`' is SparcStation IPX, '`thumper`' is Sparc-

	harvey	rimfire	thumper	highpower
Sync	216	103	86	57
Async	125	59	52	30

(a) Single server case

	Type 1	Type 2	Type 3
ALL	34	26	17

(b) Multiple server case

Table 4.2: Measured time (in seconds) to compute Mandelbrot set on [0.5,-1.8] to [1.2,-1.2] with 200×200 pixel window used.

Station 2, and '`highpower`' is SparcStation 10: the broad spectrum of computing power in these machines is intentional to cause load imbalance in the later load balancing test.) Asynchronous RPC is better because it allows to overlap server computation with communication. Table 4.2 (b) shows timing results when all four machines are cooperating for the computation. Each row in the Table 4.2 (b) indicates the type of load balancing among four servers. Type 1 is when tasks are distributed equally in spite of divergence in computing power – the performance is degenerated to that of harvey, the slowest machine (see "$34 \approx 125/4$"). Type 2 is when the task migration paths are linearly connected, i.e. "client $\rightarrow$ harvey $\rightarrow$ rimfire $\rightarrow$ thumper $\rightarrow$ highpower." Type 3 is when the paths are circular and the client distributes the equal number of tasks to all servers initially. The CORD system allows us to track down these configurations towards better performance without having to worry about extensive amount of manual adaptations. Each of the execution scenarios shows performance that is comparable to a manually coded counterparts, yet these were achieved without extensive manual intervention on the part of programmers.

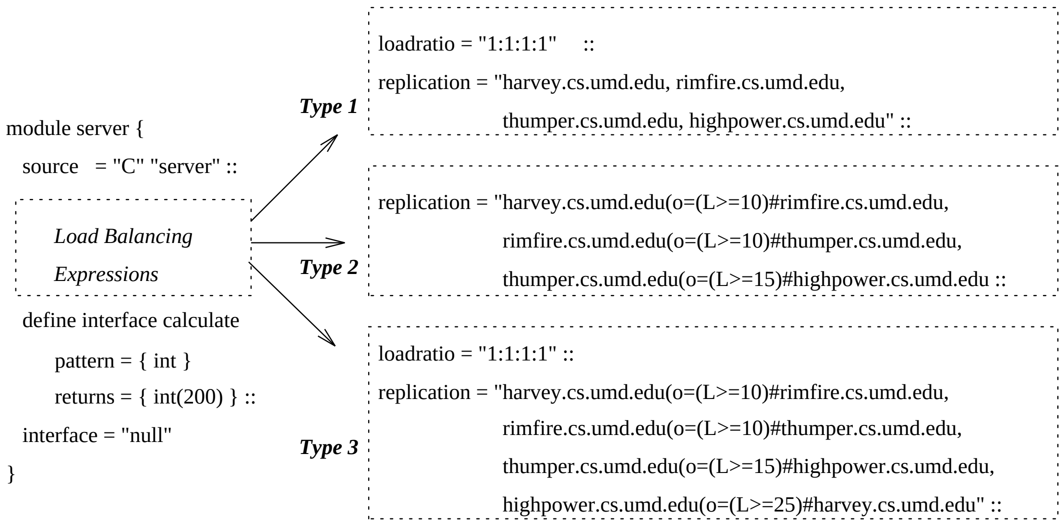


Figure 4.6: Module specification for various load balancing schemes.

Chapter 5

Source-to-Source Transformation

When an RPC is implemented through traditional stub generation based methods [BN84, CP91, Gib87], a stub takes in charge of the three functions: (1) **communication** – RPC arguments are transmitted to the remote callee, and the result is back to the caller, (2) **synchronization** – the caller is suspended until the result is back, and (3) **data conversion** – machines may have distinct data representation formats from others.[1]

Basically there is no technical difference between stub generation and in-line transformation for an RPC statement. Both of them rely on communication primitives provided by underlying MP systems [For93, Sun90] or operating systems. But the transformation based method allows us to apply various program analysis techniques for program transformation towards high performance.

As we assemble those MP primitives to implement an RPC statement, which is regarded as a 'big' statement, we have freedom to place each low-level primitive appropriately interspersed in a module in order to achieve our aforementioned goals of enhanced parallelism and minimized communication. The transformation merely affects client parts. The transformation is performed at client side to implement its remote procedure call. A server is synthesized to start with *prologue* part that receives various requests from all eligible clients, and to end with *epilogue* part that sends the result to the actual destination(s) rather than its caller (Chapter 5.3). The prologue and epilogue parts will be synthesized to process receive requests and send results along with optimized data paths according to the control dependences.

This chapter presents the constraints that must be preserved through in-line RPC transformation process, the compiler techniques to achieve our optimization goals, and the method to finally produce appropriate client and server modules based on optimizing transformation.

[1] In general, a data type is converted into a standardized type like XDT [Cor91] (*encoding*) before converted into a specific type (*decoding*). Without having external data conversion, if L different languages and M different machines are intermixed in a distributed application, then potentially $(L \times M)^2$ cases of data conversion must be used [Gib87].

(a) Original code	(b) After renaming	(c) After transformed
		S'_1: **Send**`(f(), x);`
		S'_2: **Send**`(g(), z);`
		S'_3: **Send**`(h(), v, w);`
		S'_4: `x =` **Receive**`(f());`
S_1: `x = f(x);`	S_1: `x = f(x);`	S'_5: **Send**`(g(), x);`
S_2: `y = g(x, z);`	S_2: `y = g(x, z);`	S'_6: `y =` **Receive**`(g());`
S_3: `z = h(v);`	S_3: `zz = h(v);`	S'_7: `z =` **Receive**`(h());`
S_4: `x = h(w);`	S_4: `xx = h(w);`	S'_8: `x =` **Receive**`(h());`

Figure 5.1: Eliminating spurious data dependences for parallelization.

5.1 Constraints On Source Transformation For RPC

Exploitation of parallelism is limited by data and control dependences in a program. Dependence constraints are directly related to the semantics of a program. Executing dependent statements simultaneously or in different order may change the original semantics of a program. Program transformation to improving the performance must be guided by given program dependences.

In Figure 5.1 (a), the execution order between S_1 and S_2 must be preserved because S_2 uses the value of `x` which is defined by S_1. This is a flow (true) data dependence denoted by $S_1 \ \delta \ S_2$. The order between S_2 and S_3 must be also preserved, otherwise the value of `z` at S_2 may be changed by S_3. This is an anti dependence denoted by $S_2 \ \delta^{-1} \ S_3$. The order between S_1 and S_4 must be preserved as well because they have same variable `x` to store the results. This is an output dependence denoted by $S_1 \ \delta^o \ S_4$. Anti and output dependences are "spurious" ones because they can be eliminated if we rename the associated variables properly; for example, `z` in S_3 to `zz` and `x` in S_4 to `xx` like in Figure 5.1 (b). Such a renaming releases the imposed execution order constraints by spurious dependences, as a result, executing them in parallel is possible.

Suppose there is a sequential (client) program, where two arbitrary statements S_1 and S_2 are totally ordered with respect to $\prec$: i.e. $S_1 \prec S_2$ denotes S_1 is executed before S_2. The parallelization process is to convert the total ordering $\prec$ into a partial ordering $\prec_P$ under the following semantic-preserving constraints. The relation $\prec_P$ is an irreflexive partial ordering[2] defined as follows: (1) if S_1 is executed before S_2, then $S_1 \prec_P S_2$ and (2) if $S_1 \prec_P S_2$ and

[2]It resembles the *happened–before* relation on a set of distributed events [Lam78]. While the associated events in that relation are distributed, the relation $\prec_P$ is an ordering between statements in a single program. Since a statement cannot be executed before itself, it is irreflexive.

$S_2 \prec_P S_3$, then $S_1 \prec_P S_3$. If two statements, S_1 and S_2, are not related by the $\prec_P$ relation, then we say these two statements can be executed in parallel. If, however, $S_1 \prec_P S_2$, then it is possible for statement S_1 to causally affect statement S_2.

When we transform RPC statements into a set of MP primitives, we have following ordering relation per RPC due to the *law of causality* that is a reply can be received only after the proper request has been sent out. Let R be a set of RPC statements in a client:

$$\forall S \in R : S \stackrel{\text{transform}}{\Longrightarrow} S^{snd} \prec_P S^{recv} \qquad \textit{(Eq5.1)}$$

The statement S^{snd} only *uses* variables whereas the S^{recv} only *defines* variables. This behavioral difference between S^{snd} and S^{recv} can be utilized to widen the gap by placing S^{snd} as early as possible and placing S^{recv} as late as possible. It practically implies that other useful statements can be executed during a remote call S.

We assume that the execution time of a statement in a single program is negligible compared to the time for S^{recv}, which is for server processing time plus communication time to get back to the client. We even ignore the time for a program to finish S^{snd} assuming that the underlying MP system immediately takes the control after executing S^{snd} to complete the send. For example, a relation like $S_1^{snd} \prec_P S_2^{snd}$ is of no significance; they are parallel. Consequently, the outstanding number of S^{snd} implies the potential degree of parallelism in an RPC-based distributed program. This implies that we do not need any special constructs like **parbegin** and **parend** to express parallelized form after transformation. S^{recv} is a blocking statement and an order-preserving one that is used to preserve program semantics when there are data dependences as follows:

$$\forall S_1, S_2 \in R : \quad S_1 \; \delta \; S_2 \;\wedge\; S_1 \prec S_2 \Rightarrow S_1^{recv} \prec_P S_2^{snd} \qquad \textit{(Eq5.2)}$$

$$\forall S_1, S_2 \in R : \quad S_1 \; \delta^o \; S_2 \;\wedge\; S_1 \prec S_2 \Rightarrow S_1^{recv} \prec_P S_2^{recv} \qquad \textit{(Eq5.3)}$$

$$\forall S_1, S_2 \in R : \quad S_1 \; \delta^{-1} \; S_2 \;\wedge\; S_1 \prec S_2 \Rightarrow S_1^{snd} \prec_P S_2^{recv} \qquad \textit{(Eq5.4)}$$

Since the constraint (5.2) combined with (5.1) yields $S_1^{snd} \prec_P S_1^{recv} \prec_P S_2^{snd} \prec_P S_2^{recv}$, the two RPCs involved must be serialized. However, the constraints (5.3) and (5.4) apparently do not inhibit parallelism because S_1^{snd} and S_2^{snd} that trigger the server computations are still independent and can be executed simultaneously.

Figure 5.1 (c) shows the transformed code under the above constraints. In summary, the constraint (5.1) produces relations of $S_1' \prec_P S_4'$, $S_2' \prec_P S_6'$, $S_2' \prec_P S_7'$, and $S_5' \prec_P S_8'$; (5.2) imposes $S_4' \prec_P S_5'$ because of $S_1 \; \delta \; S_2$; (5.3) imposes $S_4' \prec_P S_8'$ because of $S_1 \; \delta^o \; S_4$; (5.4) imposes $S_5' \prec_P S_7'$ because of $S_2 \; \delta^{-1} \; S_3$. As a result, the three RPCs in statements S_2, S_3 and S_4 can run in parallel while all RPCs in the statements S_1–S_4 are executed sequentially in the original code.

5.2 Transformation Framework

In this section we present the heart of our transformation algorithm, which is to hasten RPC argument passing as early as possible (even over a procedure boundary), and to delay receiving the return value as late as possible, according to the result of **def** and **use** analysis to the variables involved. We do this in a three-step process. First, all RPCs in an application are enumerated to be *positionally different*,[3] and represented by a *call tree* (Section 5.2.2). Next, **use-def** chains for RPC arguments and **def-use** chains for a return value are evaluated by def-use analysis (Section 5.2.3). Finally, global optimization is performed over a procedure boundary (Section 5.2.4).

5.2.1 Definitions

Suppose there is a distributed program P that is composed of k different executable modules, $M_1, M_2, \ldots, M_k$ running at distinct sites. RPC_{M_1}, RPC_{M_2}, …, RPC_{M_k} are sets of *positionally different* occurrences of RPCs that are imported in M_1, M_2, …, M_k, respectively. If M_i calls M_j $(i \neq j)$ via an RPC $r \in RPC_{M_i}$, M_i is a client module and M_j is a server module. Notice that "client" and "server" are relative terms; i.e. a client to one module can be a server to another module and vice versa. A positionally different RPC r has two attributes: its client ($r.Client$) and server ($r.Server$).

$DUC_m(l)$ (Def-Use-Chain) is a set of reachable uses of a definition to a variable l in a module m. $UDC_m(r)$ (Use-Def-Chain) is a set of reaching definitions of a variable associated with use of a variable r in a module m.

$Receive\_Request(r)$ denotes a set that contains every source of arguments, which form a *request* for a remote call r. Let $|r|$ be the number of arguments for r. Then, $Receive\_Request(r)$ can be written by $\{t_i \mid t_i = (s_i, v_i), 1 \leq i \leq |r|\}$, where s_i is the module that defines the value of the i-th actual argument of the call, and v_i is the variable that contains the value in the module; i.e. $v_i \in UDC_{s_i}(a_i)$ where a_i is the i-th actual argument. The initial (i.e. unoptimized) state of $Receive\_Request(r)$ is a set of tuples of the client module of r ($r.Client$) and its argument variables. For example, if "`l = f(`$v_1, \ldots, v_n$`)`" is an RPC statement r in m, the initial contents of $Receive\_Request(r)$ will be given by $\{m : v_1, \ldots, m : v_n\}$. Our optimization algorithm tries to find an ultimate source of each argument among k modules.

$Send\_Result(r)$ denotes a set that contains every recipient of the r's return value. Conventionally, this is a singleton as the caller is the only recipient of the return value. It can be written by $\{t \mid t = (d, v)\}$, where d denotes a destination module in $\{M_1, \ldots, M_k\}$ that receives the return value, and v denotes a variable that needs it. As we are seeking direct message passing paths, this set may have multiple elements — for example, in the case of "`b = f(a); c = g(b); d = h(b);`", `g()`, `h()` and maybe the client of `f()` are the recipients of the call `f()`. In that case, a single multicast can replace a series of point-to-point communications for efficiency.

[3]Even if there is a single imported RPC in a client, the remote procedure can be called several times at different places in the client. All of these occurrences are for the the same RPC, but they are considered *different* because they may have different data flow in terms of argument passing and result returning.

5.2.2 Call Tree Construction

All occurrences of RPC in a program should be distinguished so as to construct their own optimized message passing paths. For example, in "`a = f(x); b = f(y);`", the first call to `f()` has different data flows on `x` and `a` from those on `y` and `b` in the next call. We construct a Call Tree (CT) in order to represent all *positionally different* calls with control predicates for them. The call tree (CT) is defined as follows.

Definition 5.1 Let P denote an RPC-based distributed program. The call tree of P is an unordered tree $CT = (V_{CT}, E_{CT})$, where

- The vertices V_{CT} represent a set of positionally different RPCs in P. In addition, there is a distinguished vertex *main*, which represents the root of the tree; *main* is a main procedure that is called by an operating system. The remaining vertices are partitioned into $n \geq 0$ disjoint sets $T_1, \ldots, T_n$, and each of these sets is a call tree.

- The edges E_{CT} represent calling sequences. That is, an edge $(v, w) \in E$ means that w is an RPC statement in the module of $w.Client$ (or $v.Server$, equivalently) that may be executed when v is being executed. Each edge carries a control predicate. (How to determine such a predicate is discussed later.) The predicate determines whether or not w is executed. An edge without a control predicate means T (true).

Since the call tree encompasses all possible call sequences in the program, the control predicates on edges are *flow-sensitive* information [Bar78, Cal88, Hal90]. We also define "$v \stackrel{+}{\Rightarrow} w$" to mean that $v.Server$ may indirectly call $w.Server$ by a series of RPC statements from v to w. □

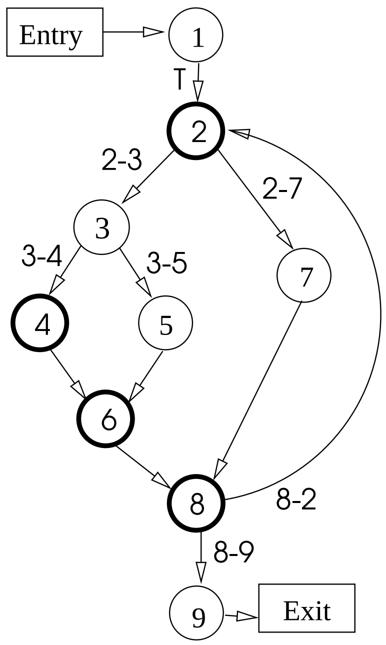
A *control flow graph* is a directed graph, $CFG = (V_{CFG}, E_{CFG})$, with unique nodes $Entry, Exit \in V_{CFG}$ such that there exists a path from $Entry$ to every node in V_{CFG} and a path from every node to $Exit$; $Entry$ has no incoming edges, and $Exit$ has no outgoing edges [FOW87]. An edge in E_{CFG} is annotated by a control predicate that determines whether or not to take the edge. We assume T (*true*) on single outgoing edge (no branch), that means the edge is always taken after executing the predecessor. Otherwise, the $(v - w)$ denotes the control predicate on an edge (v, w) among all outgoing edges from v. If P is a path from v_1 to v_n, which is $<v_1, \ldots, v_n>$, the control predicate for the path $Cpred(P)$ is $(v_1 - v_2) \wedge \ldots \wedge (v_{n-1} - v_n)$. If there are n different paths $P_1, \ldots, P_n$ that are all reachable to v_n from v_1, the control predicate for v_n from v_1 is $Cpred(P_1) \ \vee \ \cdots \ \vee \ Cpred(P_n)$.

The control predicates will be used to construct optimized server module with low-level MP primitives in the following section. Consider an example program shown in Figure 5.2 (a).[4] From the CFG in Figure 5.2 (b), we can evaluate a control predicate as follows.

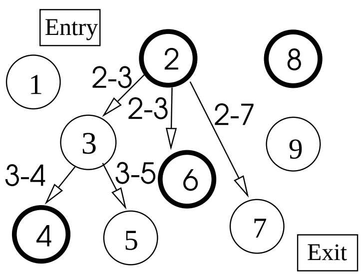
[4]In the SSA (Static Single Assignment) [CFR+91] representation of the program, a join node for the loop construct is omitted for brevity.

```
main()
{
[1]; /* defs on x1, y1, z1 */
loop {
    [2]: y2 = f1(x1);
    if ([2] − [3]) {
        [3];
        if ([3] − [4])
            [4]: z2 = f2(y2);
        else
            [5];
        [6]: w2 = h(x1);
    }
    else if ([2] − [7])
        [7];
    [8]: v2 = g(φz(z1, z2));
}
printf(y, z, w, v);
}
```

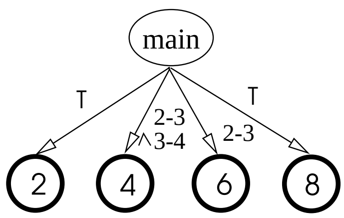
(a) An example



(b) Control Flow Graph



(c) Control Dependence Graph



(d) Call Tree

Figure 5.2: An example: CFG and CDG to construct Call Tree.

Example 5.1

$$\begin{aligned}
Cpred(Entry - \boxed{4}) &= Cpred(<E,1,2,3,4>) \ \vee \\
&\quad\ Cpred(<E,1,2,3,5,6,8,2,3,4>) \ \vee \\
&\quad\ Cpred(<E,1,2,7,8,2,3,4>) \\
&= [(2-3) \wedge (3-4)] \ \vee \\
&\quad\ [(2-3) \wedge (3-5) \wedge (8-2) \wedge (3-4)] \ \vee \\
&\quad\ [(2-7) \wedge (8-2) \wedge (2-3) \wedge (3-4)] \\
&= [(2-3) \wedge (3-4)] \ \vee \\
&\quad\ [(2-7) \wedge (8-2) \wedge (2-3) \wedge (3-4)] \\
&= (2-3) \wedge (3-4) \\
&\quad\ [\text{OR}-\text{simplification on } (2-3) \wedge (3-4)]
\end{aligned}$$

The above method requires finding all reachable paths and simplifying boolean expressions; it is computationally expensive. Control dependence [FOW87] captures the essential control flow relationships in a program. Informally, for nodes v and w in CFG, w is *control dependent on* v if v can directly affect whether w is executed or not. The *control dependence graph* is a directed graph, $CDG = (V_{CDG}, E_{CDG})$, where the vertices V_{CDG} are the same as V_{CFG} and (v, w), for v and w in V_{CDG}, is in E_{CDG} if w is *control dependent* on v. The control predicate for $(v, w) \in E_{CT}$ in Figure 5.2 (d) is computed as follows. Let all reachable paths to w in CDG of $w.Client$ be $P_1, \cdots, P_n$. Then, the predicate is $Cpred(P_1) \vee \cdots \vee Cpred(P_n)$, where $Cpred(P_i)$ is $(v_1 - v_2) \wedge \cdots \wedge (v_{n-1} - v_n) \wedge (v_n - w)$, if P_i is $<v_1, \ldots, v_n, w>$. For the edge (main, 4) in Figure 5.2 (d), $(2-3) \wedge (3-4)$ is directly obtained as its predicate, since there is the only path $<2,3,4>$ as shown in Figure 5.2 (c). No boolean simplification is needed if we use CDG as in **Example 5.1**.

A call tree construction algorithm is given below.

Algorithm 5.1 (*Call Tree Construction*)

Input:

1. All involved modules $M_1, \ldots, M_k$, M_1 is an imported set by a main procedure *main*.
2. All $r \in RPC_{M_1} \cup \cdots \cup RPC_{M_k}$ where $r.Server, r.Client \in \{M_1, \ldots, M_n\}$.

Output: $CT = (V_{CT}, E_{CT})$ as defined in **Definition 5.1**.

```
Begin
    Root ← Create_Node(main);
    Suppose RPC_1 is {v_1, ..., v_m}
    For i = 1 To m {
      t ← Create_Node(v_i);
      Add_Child(Root, t);
      Evaluate_Control_Predicate(Root, t);
      EXPAND(t);
    }
End

procedure EXPAND(T : TreeNode) {
    r ← T -> r;
    Suppose RPC_{r.Server} is {w_1, ..., w_n}
    For i = 1 To n {
      t ← Create_Node(w_i);
      Add_Child(T, t);
      Evaluate_Control_Predicate(T, t);
      EXPAND(t);
    }
}
```

5.2.3 Initialization

Suppose R is a node in CT (i.e. $R \in V_{CT}$). Our goal is to translate R into *send* (R^{snd}) and *receive* (R^{recv}) primitives for improving performance. First, we initialize *Receive_Request*(R) and *Send_Result*(R) within a module boundary, and then optimize globally. Let $t_{rr} = (s_{rr}, v_{rr})$ be an element of *Receive_Request*(R). Recall that a tuple t_{rr} corresponds to an argument in R. Since the argument values are provided by a caller itself according to a conventional procedure call/return paradigm, s_{rr} is initialized with *R.Client* and unchanged until the global optimization in the next section is performed. On the other hand, the latest location among the *definitions* that reach R initializes v_{rr}. Likewise, let $t_{sr} = (d_{sr}, v_{sr})$ be an element of *Send_Result*(R). Then, d_{sr} is initialized by *R.Client*, and v_{sr} is initialized by the earliest *use* among the uses that are reached by the return value of R. This initialization widens the gap between R^{snd} and R^{recv}. The more the gap is widened, the more statements (including another remote call) can be executed during executing R.

Let us discuss the initialization in more details. Node v *dominates* node w, denoted by $v\Delta w$, if v appears on every path from *Entry* to w [ASU86]. Node v *immediately dominates* node w iff $v\Delta w$ and there is no node x such that $v\Delta x$ and $x\Delta w$. In a dominator tree (DT) of a CFG, the children of a node v are all immediately dominated by v. When v is a closer descendent to x than y in the DT, the dominator x is called *closer* to v than y. Node v *post-dominates* node w, denoted by $v\Delta_p w$, if v appears on every path from w to *Exit* [FOW87]. Node v *immediately post-dominates* node w iff $v\Delta_p w$ and there is no node x such that $v\Delta_p x$ and $x\Delta_p w$. In a post-dominator tree (PDT), the children of a node v are all immediately post-dominated by v. When v is a closer descendent to x than y in the PDT, the post-dominator x is called *closer* to v than y. Then, the effect of initial transformation is described concisely as follows:

Property 1 *R^{snd} is the closest common post-dominator to $\{d_1, \ldots, d_m\}$ that is a UDC set for an argument variable in R.*

Property 2 *R^{recv} is the closest common dominator to $\{u_1, \ldots, u_m\}$ that is a DUC set for the return value of R.*

It might be an error that UDC is empty; that means accessing an undefined variable. When DUC is empty, the return value is never used in the caller. In this case, R^{recv} is of no use, thus eliminated by the global optimization in Section 5.2.4. For these exceptional cases, we can safely put R^{snd} (R^{recv}) to the first (last) line of the program. As the properties describe, an algorithm to find such R^{snd} and R^{recv} is straightforward: (1) compute UDC and DUC accordingly, (2) find the least common ancestors for those elements of the sets in the PDT and DT, respectively, and (3) repeat the process for all $R \in V_{CT}$. Then the transformation with **Property 1** and **Property 2** preserves the constraints (1)-(4) in Section 5.1.

Theorem 5.1 *Property 1 and 2 satisfy the Constraint (5.1) in Section 5.1.*

Proof: Obvious. □

Theorem 5.2 *Property 1 and 2 satisfy the Constraint (5.2) in Section 5.1.*

Proof: Suppose S_1 is data dependent on S_2 w.r.t a variable x. By **Property 1**, S_2^{snd} follows any reaching definitions on x, obviously including the definition by S_1^{recv}. If S_2^{snd} has to precede S_1^{recv}, S_2 must not be a reachable use from S_1, by **Property 1**, or equivalently, S_1 must not be a reaching definition to S_2, by **Property 2**, both of which contradict the data dependence between S_1 and S_2. □

Theorem 5.3 *Property 2 satisfies the Constraint (5.3) in Section 5.1.*

Proof: Suppose S_1 is output dependent on S_2 w.r.t a variable x. Let x_1, x_2 be the l-values of the definitions by S_1, S_2, respectively. Suppose there exists a $u \in DUC(x_1)$ such that it is preceded by one of $DUC(x_2)$ in the CFG. Then it means that the use u is preceded the definition of x_2, i.e. the definition of x_1 is killed by x_2 at this point. This is impossible because v must be in $DUC(x_2)$ then. Thus, all members of $DUC(x_1)$ precede those of $DUC(x_2)$. That is, the maximum depth of $DUC(x_1)$ is shallower than the minimum depth of $DUC(x_2)$ in the DT. Therefore, the least common ancestor node of $DUC(x_1)$, which is S_1^{recv}, precedes the least common ancestor node of $DUC(x_2)$, which is S_2^{recv}, in other words, $S_1^{recv} \prec_p S_2^{recv}$. □

Theorem 5.4 *Property 1 and 2 satisfy the Constraint (5.4) in Section 5.1.*

Proof: Suppose S_1 is anti dependent on S_2 w.r.t a variable x. Let x_{old} be the used variable in S_1^{snd}. Let x_{new} be the l-value of the new definition by S_2^{recv}. Suppose that $S_1^{snd} \prec_p S_2^{recv}$ cannot be satisfied by the **Property 1**; i.e., $S_2^{recv} \preceq_p S_2^{snd}$ is possible after the transformation. To make it possible, some uses in $DUC(x_{new})$ must precede (for '$\prec$') or be equal to (for '$=$') some definitions in $UDC(x_{old})$. This is impossible, by the definitions of UDC and DUC sets. □

5.2.4 Global Optimization

This phase is to seek a direct message passing path that is originally a series of message passings since it is not optimized at the interprocedural level. Sending a message m at a module y to a module z, if that is sent by a module x, is an unnecessary communication because it can be replaced with a direct communication between y and z: i.e. replacing $x \rightarrow y \rightarrow z$ with $x \rightarrow z$. To make this optimization possible, we should know that the message m is not killed at y before sending to z and not used for the rest of the program at y either. Even if m is used at y, seeking a direct path between x and z is still useful because (1) z can receive it earlier than being sent via y and (2) a single multicast operation is faster than a series of point-to-point communications. From the viewpoint of each *procedure*, the interprocedural data flow equations to this end can be expressed as following recursive forms where the $\mathsf{Called}(P)$ is the set of remote procedures called directly from P [Bar78]:

$$Use(P) = LocalUse(P) \bigcup_{Q \in \mathsf{Called}(P)} Use(Q) \qquad (Eq5.5)$$

$$Def(P) = LocalDef(P) \bigcup_{Q \in \mathsf{Called}(P)} Def(Q) \qquad (Eq5.6)$$

We can rewrite the above equations as the following concrete forms, because (1) call/return is the sole mechanism of interactions between remote processes [BN84], and (2) call-by-value semantics is useful enough in general distributed programs [HL82].

$$Use(P) = LocalUse(P) \bigcup RetUse(P) \bigcup_{Q \in \mathsf{Called}(P)} Call(Q) \qquad (Eq5.7)$$

$$Def(P) = LocalDef(P) \bigcup ArgDef(P) \bigcup_{Q \in \mathsf{Called}(P)} Return(Q) \qquad (Eq5.8)$$

A local use ($LocalUse(P)$) is a *use* that is not used for remote interactions like in arguments to issue an RPC or in a return statement in P to its remote client. Non-local uses are two kinds: $RetUse(P)$ is a *use* in a return statement (it is expected to be used at a remote site ($P.Client$) that calls P and waits for the return) and $Call(Q)$ is a set of variables that are used in a statement of calling another remote procedure Q. A local definition ($LocalDef(P)$) is a *definition* that is not defined by P's client (the caller provides the initial values of the formal parameters in P) or P's servers (a variable in P is assigned by the return value of P's server procedure). Non-local definitions are two kinds: $ArgDef(P)$ is a *definition* that defines a formal parameter of P, and $Return(Q)$ is a *definition* that defines a variable in P as an l-value of the RPC to Q.

On the other hand, from the viewpoint of each *positionally different RPC statement*, where we are interested in seeking *true definitions* and *true uses* associated with the call, the $Receive\_Request(r)$ and $Send\_Result(r)$ sets can be defined as follows:

$$Receive\_Request(r) = RR_{r.Server}(M_1) \cup \cdots \cup RR_{r.Server}(M_k) \qquad (Eq5.9)$$

$$Send\_Result(r) = SR_{r.Server}(M_1) \cup \cdots \cup SR_{r.Server}(M_k) \qquad (Eq5.10)$$

$RR_{r.Server}(M_i)$ is a set of variables that are defined at M_i in order to be used at $r.Server$. $SR_{r.Server}(M_i)$ is a set of uses of the return value of $r.Server$ at M_i. Recalling the only way to interact between distinct modules is via an argument and return value passing, these two sets can be defined as follows:

$$RR_s(c) = \begin{cases} Use(s) \cap Def(c) \equiv Call(s) \cap Def(c) & \text{if } c \text{ calls } s \text{ directly} \\ RR_s(t_n) \cap \cdots \cap RR_{t_2}(t_1) \cap RR_{t_1}(c) & \text{if } c \overset{+}{\Rightarrow} s \\ \phi & \text{otherwise} \end{cases} \qquad (Eq5.11)$$

$$SR_s(c) = \begin{cases} Def(s) \cap Use(c) \equiv Return(s) \cap Use(c) & \text{if } c \text{ calls } s \text{ directly} \\ SR_s(t_n) \cap \cdots \cap SR_{t_2}(t_1) \cap SR_{t_1}(c) & \text{if } c \overset{+}{\Rightarrow} s \\ \phi & \text{otherwise} \end{cases} \qquad (Eq5.12)$$

$Use(s)$ in Eq. (5.11) can be replaced with $Call(s)$ because the passed arguments are the only variables that are used in the server module s, since there are no aliasing and reference variables. Likewise, $Def(s)$ in Eq. (5.12) can be replaced with $Return(s)$, because the return value is the only definition that can be defined by a remote procedure s. Notice that Eq. (5.11) and Eq. (5.12) are dual only if $Def(c)$ in Eq. (5.11) is $Return(s')$ and $Use(c)$ in Eq. (5.12) is $Call(s')$ (i.e. all other terms are null in Eqs. (5.7), (5.8)), which implies that a return value of an RPC s' is used to call s when c calls s.

Consequently, if we compute each *definition* and *use* set as shown in Eqs. (5.7) and (5.8), we can compute $Receive\_Request(r)$ and $Send\_Result(r)$ sets that contain direct message paths. As the intraprocedural **def-use chains** and **use-def chains** have already been computed in an initialization phase, we are ready to solve Eqs. (5.9)–(5.12).

Interprocedural data path may be analyzed when there is a chain of procedure calls, i.e. $c \stackrel{n}{\Rightarrow} s$ when $n > 1$. Solving these equations directly is not realistic, however, we can obtain the solutions indirectly using the implications of the equations. To see if a solution exists in Eq. (5.11), we need to check if an argument is passed without being changed from c to s along the call path. For example, as shown in Figure 5.3 (a), if the client $\mathtt{C}_n$ sends a value a to the server `S` and the value a is an input argument provided by its caller $\mathtt{C}_{n-1}$, then the server `S` can receive the argument value directly from $\mathtt{C}_{n-1}$, and ultimately up from $\mathtt{C}_1$. Similarly, to see the same thing in Eq. (5.12), we need to check if a return value from s is returned again in c. As shown in Figure 5.3 (b), if the client `C` receives a result from the server $\mathtt{S}_1$ and the value is the return value from its server $\mathtt{S}_2$, then the client `C` can receive the value directly from $\mathtt{S}_2$, and ultimately from $\mathtt{S}_n$. Finally, if the client `C` sends a request of values `x, y` for an RPC "`z = S3(x, y)`" and the values are actually defined by another RPCs "`x = S1(`$\cdots$`)`" and "`y = S2(`$\cdots$`)`", respectively, then they can be directly sent from the module `S1()` and `S2()` to the module `S3()` as shown in Figure 5.3 (c). As this is a mixed case, it is checked by solving $SR_{s1}(c) \cap RR_{s2}(c)$. Notice that the *parallelism in breadth* is exploited between `S1()` and `S2()`. Interestingly, the *parallelism in breadth* is also exploited between `S1(), S2()` and `S3()` because of the data dependence on `x` and `y`.

Other than seeking a direct path, a message passing path can be eliminated if a return value is not used in a caller module except being used as an argument for another RPC; i.e. the corresponding *send_result* and *receive_result* pair collapses. Figure 5.4 summarizes the algorithm for the global optimization we have discussed in this section.

5.2.5 Loop Transformation

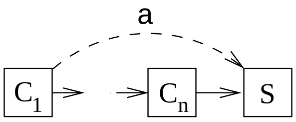
Many research works have been focused on loop transformations in various parallel compilers, for loops are hot spots in a program [Pol88]. We are interested in transforming a loop as well, especially when RPC statements are surrounded by a loop. Executing an RPC is involved in rather longer delay. Aggregating remote messages can drastically reduce an inter-networking overhead by sharing the overhead by multiple messages. If a loop contains RPCs, the chance to reduce the overhead through aggregation is higher [LS88], thus careful loop transformation provides good opportunity for aggregation.

```
C1()
{
···
x=C2(a);
···
}
```

```
···  Cn-1(Type a)
{
···
x=Cn(a);
···
}
```

```
Cn(Type a)
{
···
x=S(a);
···
}
```

```
S(Type a)
{
/* do S() */
return(···);
}
```



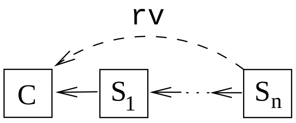
(a) Call path optimization

```
C()
{
···
x=S1(···);
y=x+2;
···
}
```

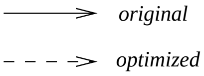
```
S1()
{
/* do S1()
*/
rv=S2(···);
return(rv);
}
```

```
···  Sn-1()
{
/*do Sn-1()*/
rv=Sn(···);
return(rv);
}
```

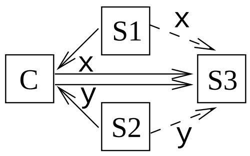
```
Sn()
{
/*do Sn()*/
return(rv);
}
```



(b) Return path optimization




```
C()
{
      x = S1(...);
      y = S2(...);
      z = S3(x, y);
}
```



(c) Mixed path optimization

Figure 5.3: Example code shapes for global optimization.

Loop distribution breaks a single loop into multiple loops with the same iteration space but each enclosing a subset of the statements in the original loop [PW86]. It is used to improve instruction and data locality by shortening loop bodies and to allow parallelism that is hindered by loop-carried dependences in the original loop. The latter effect is important in applying the technique to a loop that contains RPC statements. An original loop shown in Figure 5.5 (a) can be distributed as in Figure 5.5 (c). It surely eliminates the flow dependence between two statements in the loop, however, the `F()` over the iteration space cannot run in parallel in Figure 5.5 (c) even if we assume there are replicated servers for the procedure, because RPC in each iteration is synchronous. If we transform RPC statements into statements of message passing primitives according to our transformation based approach, the original loop in Figure 5.5 (a) would be transformed into Figure 5.5 (b), and then Figure 5.5 (d) after loop distribution.

Recalling our assumption that S^{snd} takes negligible amount of time, N different calls can be placed when there are N servers. Even if only a server is available, the calls can

Algorithm 5.2 (*Optimization*)

Input:
1. $CT = (V_{CT}, E_{CT})$ by **Algorithm 5.1**.
2. All $r \in V_{CT}$ where $r.Server, r.Client \in \{M_1, \ldots, M_n\}$.
3. Initialized $Receive\_Request(r)$ and $Send\_Result(r)$.

Output: Optimized $Receive\_Request(r)$ and $Send\_Result(r)$.

Begin
  for each $t_{rr} \in Receive\_Request(r)$ **do**
    while $(val(t_{rr}) \in ArgDef(r, source(t_{rr})))$ **do**
      $r \leftarrow previous(r)$; /* the predecessor of r in CT */
      $source(t_{rr}) \leftarrow r.Client$; $val(t_{rr}) \leftarrow UDC_{r.Client}(ACTUAL(r, t_{rr}))$;
    endwhile
  endfor /* Call path optimization */
  for each $t_{sr} \in Send\_Result(r)$ **do**
    while $(val(t_{sr}) \in RetUse(r, dest(t_{sr})))$ **do**
      $r \leftarrow previous(r)$; /* the predecessor of r in CT */
      $dest(t_{sr}) \leftarrow r.Client$; $val(t_{rr}) \leftarrow DUC_{r.Client}(LVALUE(r))$;
    endwhile /* Return path optimization */
    for each sibling edge r_{sib} of a node $dest(t_{sr})$ in CT **do**
      if $(val(t_{sr}) \in Call(r_{sib}, r_{sib}.Server))$
        $new\_t_{sr} \leftarrow CreateTupleSR(r)$;
        $dest(new\_t_{sr}) \leftarrow r_{sib}.Server$; $val(new\_t_{sr}) \leftarrow FORMAL(r_{sib}, t_{sr})$;
        $Send\_Result(r) \leftarrow Send\_Result(r) \cup \{new\_t_{sr}\}$;
        for each $t_{rr} \in Receive\_Request(r_{sib})$ **do**
          if $(val(t_{rr}) \in Return(r, r.Client))$
            $source(t_{rr}) \leftarrow r.Server$; $val(t_{rr}) \leftarrow RETVAL(r)$;
        endfor
    endfor /* Mixed path optimization */
    if $(t_{sr} \notin LocalUse(r, r.Client))$
      $Send\_Result(r) \leftarrow Send\_Result(r) - \{t_{sr}\}$;
  endfor
End

Figure 5.4: Global optimization algorithm.

```
for (i = 0; i < N; i++) {
    a[i] = a[i] + c;
    x[i+1] = F(a[i], x[i]);
}
```

(a) Original loop

⇒

```
for (i = 0; i < N; i++)
    a[i] = a[i] + c;
for (i = 0; i < N; i++)
    x[i+1] = F(a[i], x[i]);
```

(c) After loop distribution

⇓

```
for (i = 0; i < N; i++) {
    a[i] = a[i] + c;
    send_req(F(), a[i], x[i]);
    x[i+1] = recv_res(F());
}
```

(b) Transformed RPC into MP primitives

⇒

```
for (i = 0; i < N; i++)
    a[i] = a[i] + c;
for (i = 0; i < N; i++)
    send_req(F(), a[i], x[i]);
for (i = 0; i < N; i++)
    x[i+1] = recv_res(F());
```

(d) After loop distribution

Figure 5.5: Loop distribution and call streaming

be streamed, so it reduces the cost of transmitting the call and reply messages because the streamed calls and replies can be buffered and sent to allow us to amortize the overhead of kernel calls and the transmission delays over several calls. It is called *call-streaming*, which was proposed to effectively support asynchronous calls with an aid of a special data type called "*promises*" [LS88]. Our method presents a static solution for call-streaming without relying on special programming language constructs. Moreover, an output of one remote procedure can be directly connected to an input of another one as presented in the previous section. This is not allowed in call-streaming because the results must be returned to the original caller before streams are composed.

Data aggregation to amortize kernel overhead and the transmission delays over several calls can be achieved transparently by an aid of underlying MP systems or statically by an aid of compiler that properly generates finer grained MP primitives. For instance, `send_res()` can be composed of finer primitives of `msg_decode(); msg_send()`.

5.3 Module Synthesis

The synthesis phase involves implementing source transformations based on the information from optimizing transformations. This must account for correct program behaviors in spite of drastically changed communication paths between callers and callees. The information to synthesize communication paths is summarized in $Receive\_Request(r)$ and $Send\_Result(r)$

	Receive_Request$_{INI}$	Send_Result$_{INI}$	Receive_Request$_{OPT}$	Send_Result$_{OPT}$
$\boxed{2}$	(main, x_1)	(main, y_2)	(main, x_1)	(main,y_2) & (f_2,Arg1)
$\boxed{4}$	(main, y_2)	(main, z_2)	(f_1, RetVal)	(main,z_2) & (g, Arg1)
$\boxed{6}$	(main, x_1)	(main, w_2)	(main, x_1)	(main,w_2)
$\boxed{8}$	(main,z_1) | (main,z_2)	(main,v_2)	(main,z_1) | (f_2,RetVal)	(main,v_2)

Table 5.1: Receive_Request and Send_Result sets after initialization and global optimization.

for all nodes r in V_{CT}. The information for control paths is contained in CT. A server can be associated with multiple positionally different RPCs. So, the server should be aware of all peculiar message paths for each RPC statement in CT and its run-time condition.

Table 5.1 shows the contents of *Receive_Request* and *Send_Result* sets for each RPC in the example of Figure 5.2, after initialization (Section 5.2.3) and global optimization by Algorithm 5.2. All are single-argument functions in the example; in other words, *Receive_Request*(r) has a single element. As shown in *Receive_Request*$_{INI}$($\boxed{8}$), '|' (that denotes an 'or') implies that there are multiple reaching definitions. As in *Send_Result*$_{OPT}$($\boxed{2}$) and *Send_Result*$_{OPT}$($\boxed{4}$), '&' (that denotes an 'and') implies that the result should be sent to the both destinations.

We will use three pairs of message passing primitives: `send_req`, `recv_req`, `send_res`, `recv_res`, `send_ctrl`, `recv_ctrl`. The suffixes "`_req`", "`_res`", and "`_ctrl`" (abbreviating "request", "result", and "control", respectively) are merely used to distinguish their usages. Basically, `send` and `receive` primitives suffice to implement. A pair of `send_req` and `recv_req` forms a call part in an RPC. A pair of `send_res` and `recv_res` forms a return part. A control message is used when an execution should wait for a certain run-time decision. For example, a server can continue to do a service when all input arguments are received, without knowing whether that control flow is eventually taken or not. But it should wait at the time of finishing that service and see if the control message is decided. If it is validated, then the result can be sent, otherwise, it must be discarded. We have explored all control predicates for an RPC statement and its optimal communication paths except conditions that must be resolved at run-time.

In a client module of a call r in CT, those RPC statements are transformed to a pair of `send_req()` and `receive_res()` according to the contents of *Receive_Request*(r) and *Send_Result*(r). Figure 5.6 (a) and (b) shows transformed codes, in which the original positions of RPC statements are commented by "null". In a server module S, an original code is surrounded by a pair of synthesized *prologue* and an *epilogue* codes for each $r \in CT$ such that *r.Server* is S. For example, there are two calls for the same remote procedure `f()` at $\boxed{2}$ and $\boxed{4}$, thus two pairs of prologue and epilogue are synthesized as shown in Figure 5.6 (c) because the two calls have different control and data flows.

Data availability is the only firing condition to perform *that* particular call. Therefore, when there are multiple data sets that are ready to be serviced, a selection is done non-deterministically. This can be implemented by special message passing primitives that allows a *non-blocking* receipt. For example, in POLYLITH system [Pur94], `mh_readselect()` allows us to read the next message to arrive on *any* interface (it will be blocked if no message arrives), then `mh_readback()` completes the receipt. In PVM [Sun90], `pvm_nrecv( int msgtag )` checks to see whether a message with label `msgtag` has arrived. If not arrived, it immediately returns so that other message can be checked out. *Non-blocking* receive primitives are commonly supported by MP systems.

Finally, let's consider what has been improved in Figure 5.6 (b) from (a). There is no difference regarding the degree of parallelism, that is constrained by inherent data dependences. However, if '$\boxed{2} \rightarrow \boxed{4} \rightarrow \boxed{6} \rightarrow \boxed{8}$' is a call sequence to be taken, the message passing path of '$\mathtt{main} \rightarrow \mathtt{f}_1 \rightarrow \mathtt{main} \rightarrow \mathtt{f}_2$' is simplified by '$\mathtt{main} \rightarrow \mathtt{f}_1$' and '$\mathtt{main} \rightarrow \mathtt{f}_2$', and '$\mathtt{f}_2 \rightarrow \mathtt{main} \rightarrow \mathtt{g}$' is simplified by '$\mathtt{f}_2 \rightarrow \mathtt{g}$'. Moreover, the execution of `g()` is hastened by hoisting the corresponding `send_req` primitive up to the point before $\boxed{5}$ (if $\boxed{3}-\boxed{4}$ branch is taken), or by receiving the necessary argument earlier directly from `f()` (if $\boxed{3}-\boxed{5}$ branch is taken).

```
main() {
  [1]; /* defs on x1, y1, z1 */
  loop {
    send_req(f(), x1);
    [2]; /* null */
    if ([2] − [3]) {
      send_req(h(), x1);
      [3];
      if ([3] − [4]) {
        [4]; /* null */
        y2 = recv_res(f());
        send_req(f(), y2);
      }
      else [5];
      [6]; /* null */
    }
    else if ([2] − [7]) [7];
    [8]; /* null */
    if ([2] − [3] ∧ [3] − [4])
      z2 = recv_res(f());
    send_req(g(), φ(z1, z2));
    if ([2] − [3])
      w2 = recv_res(h());
    v2 = recv_res(g());
  }
  printf(y, z, w, v);
}
```

(a) After initialization only

```
main() {
  [1]; /* defs on x1, y1, z1 */
  loop {
    send_req(f(), x1);
    [2]; /* null */
    if ([2] − [3]) {
      send_req(h(), x1);
      [3];
      send_ctrl(f(), [2] − [3] ∧ [3] − [4]);
      if ([3] − [4])
        [4]; /* null */
      else {
        send_req(g(), z1);
        [5]; }
      [6]; /* null */
    }
    else if ([2] − [7]) [7];
    [8]; /* null */
    y2 = recv_res(f());
    if ([2] − [3] ∧ [3] − [4])
      z2 = recv_res(f());
    if ([2] − [3])
      w2 = recv_res(h());
    v2 = recv_res(g());
  }
  printf(y, z, w, v);
}
```

(b) After global optimization

```
f(/* int a */) {
f1_p: a = recv_req(main(), x1); end_p
f2_p: a = RetVal of f(); end_p
      /* do f(): original source */
f1_e: send_res(main(), y2);
   c1 = recv_ctrl(main());
   if (c1) { /* [2] − [3] ∧ [3] − [4] */
      goto f2_p;
   }
end_e
f2_e: send_res(main(), z2);
   send_res(g(), Arg1);
end_e
}
```

```
g(/* int a */) {
   a=recv_req(f()) || recv_req(main());
     /* do g(): original source */
   send_res(main(), v2);
}
```

```
h(/* int a */) {
   a = recv_req(main());
     /* do h(): original source */
   send_res(main(), w2);
}
```

(c) Servers after global optimization

Figure 5.6: Transformed client and server modules for Figure 5.2.

Chapter 6

Load Balancing

Under a heterogeneous network of workstations, a simple policy like equally distributing workloads to multiple processors may lead to a parallelization anomaly. That is, the execution time of the given workload may take longer even if the number of workstations is increased. Suppose there are n processors $\{P_1, \ldots, P_n\}$, and T identical tasks. Let τ_i be the number of tasks per unit time that the processor i can process. In equal distribution, each processor has T/n numbers of tasks. The execution time of the program is determined by the critical processor that has the smallest τ_i value; let's say it is τ_{min}. Then the execution time is $\frac{T/n}{\tau_{min}} = \frac{T}{n\tau_{min}}$. Now, let's add a new processor of τ_{new} to the cluster for the application. Each processor will have $T/(n+1)$. Therefore, if $\tau_{new} < \frac{n}{n+1}\tau_{min}$, the execution time of $(n+1)$-processors cluster is $\frac{T}{(n+1)\tau_{new}}$, which is longer than that of n processors!

One may want to get around this problem by allocating tasks according to the known computing power of each processor [GWWECL94, CS93]. However, their methods were static, thus of limited usefulness. Dynamic loop scheduling methods can deal with more general cases, but the centralized nature of the methods — the central processor that generates sub-tasks has to manage all other processors — may cause a bottleneck in a network of many workstations. For example, if there are 100 servers, and if a master needs 10^{-2} second to prepare and send a task, the master would create a bottleneck unless the average time for each server to finish a task is greater than one second. In our experimentation with the Mandelbrot set computation on $[0.5, -1.8]$ to $[1.2, -1.2]$ using a 400×400 pixel window, the program reached its saturation point at 25 workstations under the *self-scheduling* scheme. To avoid such a situation, sub-tasks should be sufficiently large grained compared to communication overheads, but it is not likely considering relatively high communication costs in workstation clusters. Since there are many "embarrassingly parallel" applications, a decentralized load balancing scheme is called for. We present such a method that can reduce the overheads by means of establishing proper migration topology based on the known computing powers of the processors involved.

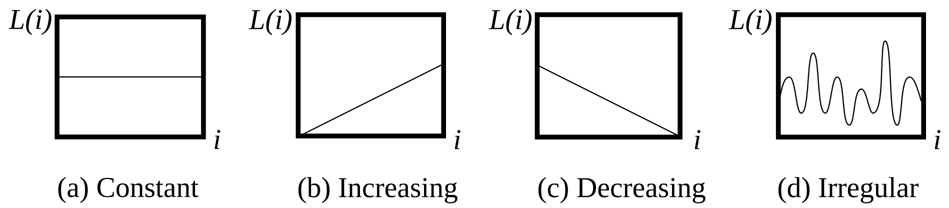


Figure 6.1: Four typical parallel loops.

6.1 Loop And Workstation Cluster Models

In this section, we classify four typical parallel loop patterns that affect performance of load balancing schemes based on workload distribution in an iteration space. Next, we discuss our workstation cluster model to deal with those diverse patterns, especially if the workstations involved are heterogeneous.

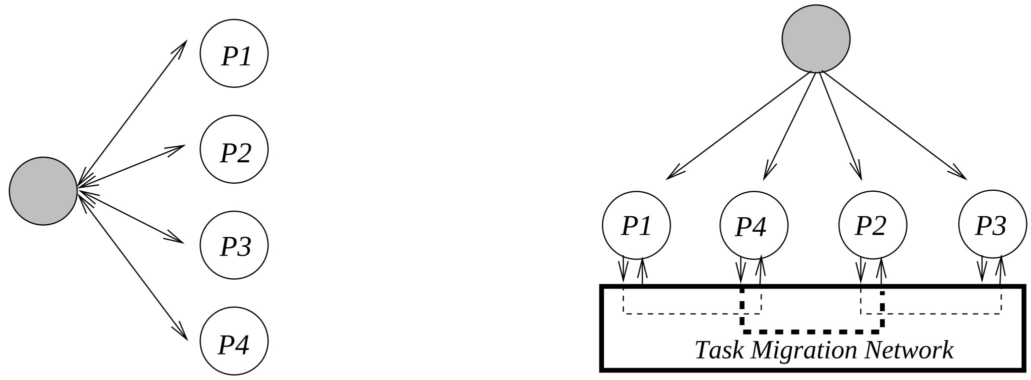
6.1.1 Loop Model

Figure 6.1 shows four typical parallel loops where $L(i)$ represents the execution time of the i-th iteration. The workload may be uniformly distributed over an iteration space as shown in Figure 6.1 (a). It may also be non-uniform but *linearly* distributed as in Figs. 6.1 (b) and (c); this kind of distribution is often contained in scientific programs. Finally, as in Figure 6.1 (d), the workload may be quite irregular. Many non-scientific applications carry parallel loops of this type. The first three cases have been specially considered by conventional loop scheduling methods [PK87, TN91, CLZ95] in order to improve on the basic self-scheduling method.

Particularly for irregular loops, we can distinguish between the two cases: predictable vs. unpredictable. For example, the parallel tasks in the DNA sequence search problem [CG89] and the Mandelbrot set computation are all irregular, but the tasks in the first problem are predictable while the tasks in the second one are not. Of course, the above three loops are all predictable.

6.1.2 Workstation Cluster Model for Load Balancing

Figure 6.2 shows two representative topologies in the workstation cluster model for parallel loops. Figure 6.2 (a) represents the topology of traditional loop scheduling methods [TY86, KW85, PK87, TN91], in which load migration is not performed. Instead, the main processor (shaded circle) prepares a set of tasks and allocates them to each server whenever the server demands them. Since the scheduling process is dedicated to the main processor (shaded circle), its chance of creating a bottleneck rises as the number of servers present on the network increases. Figure 6.2 (b) illustrates the topology of our workstation



(a) Loop Scheduling (b) Our Approach

Figure 6.2: Topologies in workstation cluster model for load balancing.

cluster model. The main processor distributes workloads to all servers initially. Load balancing is attempted by task migration via pre-determined paths, deeming load state polling or exchange overhead unnecessary, unlike in global dynamic load balancing schemes. The migration is performed in a decentralized fashion between only the two processors involved. The workstation cluster model for load balancing is characterized by the following parameters:

- N: the number of workstations, $\{W_1, \ldots, W_N\}$.
- τ_i: the throughput of W_i, which is defined by the number of unit tasks per unit time.
- γ_{ij}: the amount of load to migrate from i to j.

6.2 Load Balancing Method

Two important components of dynamic load balancing schemes are *transfer policy* and *location policy* [ELZ86b, KS94]. The transfer policy determines whether a task should be processed locally or remotely by transferring it at a particular load state. The location policy determines which process initiates the migration and its source or destination. These are for global load balancing from the OS's viewpoints. Multi-dimensional load vectors determine the load state of a processor. In our system, we aim to balance parallel loops in an application. A simple 'demand' message is enough to initiate load migration rather than load state exchange [KS94] or random polling of candidate processors [ELZ86b] because the only load vector is the number of sub-tasks in a processor. The transfer policy then becomes simple: if a processor receives a request message for transfer from a processor that is running out of sub-tasks to work on, it migrates some of its sub-tasks to that processor.

```
/* P_i:  sender */
for (i = 0; i < taskcnt; i++) {
  if (pvm_nrecv(P_j,MoreTaskReq)) {
    /* a request arrived */
    n = (taskcnt-i+1) * Ratio_ij;
    /* Migrate to P_j */
    if (n) {
      pvm_initsend(PvmDataDefault);
      pvm_pkint(&n,1,1);
      pvm_pkint(&TaskQ[i],n,1);
      pvm_send(P_j,TaskMigrating);
      i += n;
      continue;
    }
  }
  /* loop body on TaskQ[i] */
}
```

```
/* P_j:  receiver */
LOOP:
  for (i = 0; i < taskcnt; i++) {
    /* loop body on TaskQ[i] */
  }
  /* Check the partner processor P_i */
  pvm_initsend(PvmDataDefault);
  pvm_pkint(&more,1,1);
  pvm_send(P_i,MoreTaskReq);
  /* Wait until killed by parent */
  while(1)
    if (pvm_nrecv(P_i,TaskMigrating)) {
      /* migrated tasks arrived */
      pvm_upkint(&taskcnt,1,1);
      pvm_upkint(TaskQ,taskcnt,1);
      goto LOOP;
    }
```

Figure 6.3: Programs generated for a migration path in Figure 6.2 (b).

Likewise, the location policy is now modified by the problem of establishing proper task migration paths. Workstation clusters have virtually no restrictions on topology for migration. It may be assumed that any two point-to-point communication overheads are equal, but identifying the optimal sender and receiver pair is essential. Considering all possible candidates for sender (or receiver) to migrate the excess load causes high overhead, but it is avoidable. The key is how to identify the busy and the idle processors in the middle of computations. Since the relative processing speeds of workstations in a cluster are known in advance, the possible senders and receivers of migrations are not unknown — momentary overload by other activities is the reason for uncertainty.

In this section, we present how to construct such a task migration network as shown in Figure 6.2 (b). Once the network is constructed, load balancing is pursued through task migration on it. For example, each pair connected in a dotted line in Figure 6.2 (b) ($P_i \rightarrow P_j$) is a basic unit of migration; whenever the faster processor (P_j) depletes its workload, it demands that its pre-determined partner P_i share some of P_i's workload, and P_i migrates γ_{ij} of its current workload to P_j. Figure 6.3 shows the generated source codes for such a connection. First, we will formally define the cluster model in Section 6.1.2. Then, we will describe how to construct such a cluster and its corresponding migration network based on the model.

A cluster is a bipartite form of (w_s, w_f), in which w_s is slower than w_f: i.e. $\tau_s < \tau_f$. Throughout the paper, we use the notation (τ_s, τ_f) interchangeably with the notation (w_s, w_f) when we focus on throughputs. An entire workstation cluster is defined as follows:

Definition 6.1 The *cluster tree* (CT) of N workstations $\{W_1, \ldots, W_N\}$ is a binary tree $CT = (V, E_{left} \cup E_{right})$, where

- The vertices V represent *clusters*. A distinguished vertex 'root' represents an entire cluster, and the right sub-cluster is faster than (or equal to) the left sub-cluster.

- E_{left} is a set of edges to the left sub-trees. E_{right} is a set of edges to the right sub-trees.

- If $(c, v) \in E_{left}$ and $(c, w) \in E_{right}$, a load migration path exists from v to w. When v and w are not terminal nodes, the path is established from the fastest node in cluster v, which is the rightmost terminal in the subtree of v, to the slowest node in cluster w, which is the leftmost terminal in the subtree of w.

Terminal vertices are individual workstations. Each terminal v is associated with its throughput τ_v. Throughput of non-terminal node $C = (v, w)$ is defined by $(\tau_v + \tau_w)$, which is explained by Theorem 6.3. □

Definition 6.2 In a cluster $C_1 = (\tau_1, \tau_2)$, the *balance ratio* B_{C_1} is defined by $\frac{(\tau_2 - \tau_1)}{(\tau_2 + \tau_1)}$. A cluster $C_1 = (\tau_1, \tau_2)$ is said to be *more balanced* than another cluster $C_2 = (\tau_3, \tau_4)$, if the *balance ratio* of C_1 is less than that of C_2, i.e. $\frac{(\tau_2 - \tau_1)}{(\tau_2 + \tau_1)} < \frac{(\tau_4 - \tau_3)}{(\tau_4 + \tau_3)}$. □

Definition 6.3 A cluster $C_1 = (\tau_1, \tau_2)$ is *faster* than another cluster $C_2 = (\tau_3, \tau_4)$ if τ_{C_1} is greater than τ_{C_2}, or if τ_{C_1} is equal to τ_{C_2} and C_1 is *more balanced* than C_2. □

In the extreme case that τ_1 is equal to τ_2, the *balance ratio* is zero; thus load is perfectly balanced. Likewise, in the other extreme in which τ_2 is much greater than τ_1, the ratio is asymptotically 1. The *balance ratio* in a cluster can be related to the amount of load migration. When the components in a cluster are equally loaded initially, if the cluster is perfectly balanced, then no intra-cluster migration is necessary. In other words, the more balanced a cluster is, the less migration is needed.

The process of constructing a cluster tree from a set of workstations is done in recursive "bitonic" fashion. First, workstations in the set $\{w_1, \ldots, w_n\}$ become terminal nodes in the tree. They are sorted in ascending order by their throughputs. Let the sorted set be $\{w'_1, \ldots, w'_n\}$. The fastest one (w'_n) is coupled with the slowest one (w'_1), the second fastest one (w'_{n-1}) is coupled with the second slowest one (w'_2), and so forth. The couples come to have parents in the tree, i.e. $\{c_1 = (w'_1, w'_n), \ldots, c_{n/2} = (w'_{n/2}, w'_{n/2+1})\}$, which are likewise sorted by their throughputs. Again, they are coupled in bitonic fashion. This process continues until it reaches a single cluster. Notice that the cluster of the two identical components still needs an intra-cluster migration because an equal distribution is not always possible. Once such a tree is constructed, the task migration topology is determined as follows:

Algorithm 6.1 (*Task migration network from CT)*

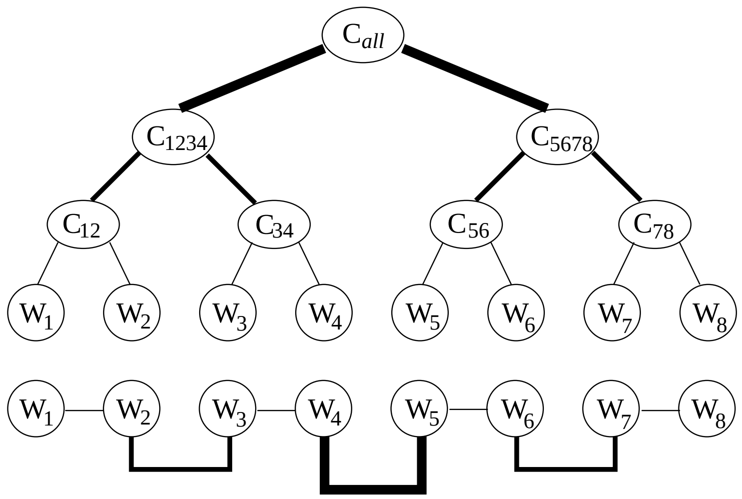


Figure 6.4: A cluster tree and its corresponding task migration paths.

Begin
  For all clusters (non-terminal nodes) c in CT
    For two children v and w such that $(c, v) \in E_{left}$ and $(c, w) \in E_{right}$
      if (v, w are terminals) **then** $CONNECT$ v TO w
      else $CONNECT$ RightmostTerminal(v) TO LeftmostTerminal(w)
End

Figure 6.4 shows the relationship between the cluster tree and the migration topology. For example, the rightmost terminal of C_{1234} is W_4, and the leftmost terminal of C_{5678} is W_5, so the link for the root cluster C_{all} is constructed between W_4 and W_5. The thicker links denote higher level links; they will be used only if the load cannot be balanced through the lower links.

6.3 Analysis Of Migration Behaviors

There are two important concerns in devising a load balancing scheme [ELZ86b]. First, the overhead should not negate the benefits of an improved load distribution. Next, the potential migration instability[1], in which processors spend too much time transferring tasks, should be avoided. Our method is orthogonal to the stability issue because a demand is issued only when the processor is idle. In this section, we present an analytic result on the overheads incurred by our method. We start with an example case to explain our method qualitatively.

[1]For example, in a two-processor system where both are overloaded, they may continuously migrate each part of loads to the other processor, which does not improve the situation at all.

Example 6.1 Suppose there are four processors $P1, P2, P3$ and $P4$ that have N identical tasks initially and we know their relative throughputs, which are $\tau, 2\tau, 3\tau$ and 4τ. When a load state of a potential sender $P1$ is probed by other processors, migration to $P2$ or $P3$ would be wasteful because its resulting resolution of $P1$'s overloaded state may be merely temporal. Since $P4$ is the fastest, the then-migrated load may have to be migrated again to $P4$, while a single migration directly to $P4$ would have been more efficient. Thus we can say the $P1$ has the greatest affinity to $P4$ among all possible receiver candidates. $\square$

The above example suggests that the slowest processor should be connected to the fastest processor, and the second slowest one is to the second fastest one, and so on, in bitonic fashion. The resulting pairs would tend to be *more balanced* in terms of the combined throughputs. We will elaborate on the effects of this kind of bitonic pairing in Section 6.3.1. This method calls for load migration to be done in as much bulk as possible. One ten-byte sized load migration is cheaper than ten one-byte sized load migrations. This is particularly important in workstation clusters where the communication overheads are still high.

Example 6.2 Let us consider the topology of $P1 \rightarrow P4$ and $P2 \rightarrow P3$ as shown in Figure 6.2 (b). Throughputs are the same as in Example 6.1. In this case the combined throughputs of the two sub-clusters turn out to be equal. That is, no further load migration is necessary through the link between the two clusters $(P1, P4)$ and $(P2, P3)$! $\square$

However, now that cluster $(P2, P3)$ is *more balanced* than cluster $(P1, P4)$, the resulting decrease in the intra-cluster migration makes cluster $(P2, P3)$ process more tasks. That is why this cluster is defined as the faster one in Def. 6.3. In general, such an ideal case may not be common in real situations; throughputs may fluctuate in the middle of computing and initial distributions are not always equal. For the case that the load is not balanced in the first cluster for some reason, we continue to balance the load through inter-cluster migrations. In the following analysis, we use $\gamma_{ij} = 1/2$, for all i, j, which guarantees uni-directional migration is enough for load balancing (notice P_j is faster), although more aggressive choice like $\gamma_{ij} = \tau_i/\tau_j$ may reduce overheads.

6.3.1 Preliminaries

To examine migration overhead, we need a communication time model. The conventional approach to modeling communication time for transferring a message of m bytes is a simple linear function, i.e. $T_{comm} = \alpha + \beta m$, where α is startup time and β is transfer time per byte [BR89]. The empirical values for α and β under the PVM system [Sun90] at LAN-based clustered workstations are 4.527 *msec*, 0.0024 *msec* and 1.661 *msec*, 0.00157 *msec* for datagram and stream transmission cases, respectively, which imply $\alpha \gg \beta$ [SS94].

In Theorems 6.1 and 6.2, we compute the total number of migrated tasks (β's multiplier) and the frequencies of migrations (α's multiplier) in a cluster. Furthermore, we also illustrate an important characteristics of our method, which is that balance ratio gets improved as clustering happens at higher levels.

Theorem 6.1 In a cluster $C = (v, w)$ where v and w are terminal nodes in CT, and they have initially loaded N identical tasks respectively, the total number of tasks to be migrated from v to w to meet the finish times at both processors is $\frac{\tau_w - \tau_v}{\tau_w + \tau_v} N$, i.e. the balance ratio of C times N.

Proof: Let us determine the general terms of the number of tasks to be migrated from v to w at the time w becomes idle. Since w is faster than v, w's first incidence of task depletion occurs after $\frac{N}{\tau_w}$; thus the number of tasks in the first migration is half of what remains in v at that time, which is $\frac{1}{2}(N - \frac{N}{\tau_w} \cdot \tau_v) = \frac{N}{2}(1 - \frac{\tau_v}{\tau_w})$. Notice that τ_v/τ_w is less than 1. T_w, the total number of tasks that are eventually processed by w, is a summation of the following series:

$$\begin{aligned} T_w &= N + \frac{N}{2}(1 - \frac{\tau_v}{\tau_w}) + \frac{N}{4}(1 - \frac{\tau_v}{\tau_w})^2 + \cdots = N \sum_{i=0}^{\infty} \frac{1}{2^i}(1 - \frac{\tau_v}{\tau_w})^i \\ &= N \lim_{k \to \infty} \frac{1 - (\frac{1}{2}(1 - \frac{\tau_v}{\tau_w}))^{k+1}}{1 - \frac{1}{2}(1 - \frac{\tau_v}{\tau_w})} = \frac{2N\tau_w}{\tau_v + \tau_w} \end{aligned}$$

Therefore, $\text{Migrated}_{v \to w} = T_w - N$, which yields $\frac{\tau_w - \tau_v}{\tau_w + \tau_v} N$. □

Theorem 6.2 In a cluster $C = (v, w)$ where v and w are terminal nodes in CT, and they have initially loaded N identical tasks respectively, the frequency of migration from v to w to meet the finish times at both processors is $\log_{\frac{1}{2}(1 - \frac{\tau_v}{\tau_w})} \frac{1}{N}$.

Proof: The general term in the series is $\frac{N}{2^k}(1 - \frac{\tau_v}{\tau_w})^k$. Thus, $k = \log_{\frac{1}{2}(1 - \frac{\tau_v}{\tau_w})} \frac{1}{N}$. □

Theorem 6.3 In a cluster $C = (v, w)$ where v, w are arbitrary nodes in CT, and they have initially loaded N identical tasks, the combined throughput of a cluster $C = (v, w)$ is $\tau_v + \tau_w$, assuming no migration overhead. **Proof:** Suppose v and w are terminal nodes in CT. In Theorem 6.1, the total number of tasks processed by v and w is given by $\frac{2N\tau_v}{\tau_v + \tau_w}$ and $\frac{2N\tau_w}{\tau_v + \tau_w}$, respectively, and the finish time is $\frac{N}{(\tau_v + \tau_w)/2}$ at either processor. As cluster C have loaded $2N$ tasks in total, this may be interpreted to mean that the *de facto* throughputs of the cluster is $\tau_v + \tau_w$. Now let us assume this holds for two clusters $C_1 = (\tau_1, \tau_2)$ and $C_2 = (\tau_3, \tau_4)$; i.e. τ_{C_1} and τ_{C_2} are $\tau_1 + \tau_2$ and $\tau_3 + \tau_4$, respectively. For a cluster $C = (C_1, C_2)$ (we can assume C_1 is slower without loss of generality), we can calculate the number of tasks processed by C_2 as follows:

$$T_{C_2} = N \sum_{i=0}^{\infty} \frac{1}{2^i}(1 - \frac{\tau_{C_1}}{\tau_{C_2}})^i = N \cdot \frac{2\tau_{C_2}}{\tau_{C_1} + \tau_{C_2}} = N \cdot \frac{\tau_{C_2}}{(\tau_{C_1} + \tau_{C_2})/2}$$

By induction, this completes our proof. □

Theorem 6.3 implies that the sum of the two throughputs in a cluster may represent the combined throughput of the cluster so that we can cluster recursively in bitonic fashion. The real combined throughput can be yielded by subtracting the throughput loss incurred by migration overheads (see Section 6.3.2) from that amount.

Theorem 6.4 If there are two clusters $C_1 = (\tau_1, \tau_4)$ and $C_2 = (\tau_2, \tau_3)$, and C_1 is slower than C_2 (i.e. τ_{C_1} is less than τ_{C_2}), then another cluster $C = (C_1, C_2)$ is always *more balanced* than the *less balanced* cluster between C_1 and C_2.

Proof: Consider the case when B_{C_1} is greater than B_{C_2} (i.e. C_1 is *less balanced* than C_2). Due to the property of bitonic coupling, $\tau_1 \leq \tau_2 \leq \tau_3 \leq \tau_4$ must hold. Let us write $\tau_2 = a\tau_1$, $\tau_3 = ab\tau_1$ and $\tau_4 = abc\tau_1$, where $a, b, c \geq 1$. By Theorem 6.3, B_C is yielded by $\frac{\tau_2+\tau_3-\tau_1-\tau_4}{\tau_1+\tau_2+\tau_3+\tau_4}$. That is, $B_{C_1} = \frac{abc-1}{abc+1}$ and $B_C = \frac{a+ab-(abc+1)}{abc+ab+a+1}$. Since $(abc + ab + a + 1) \cdot (abc - 1) - (a + ab - (abc + 1)) \cdot (abc + 1) = 2abc(abc + 1) - 2a(b + 1) \geq 0$, B_C is less than or equal to B_{C_1}. But if $2abc(abc + 1) - 2a(b + 1) = 0$, all a, b, c must be 1, which implies $\tau_1 = \tau_2 = \tau_3 = \tau_4$ that contradicts the given assumption ($\tau_{C_1} < \tau_{C_2}$ or $B_{C_1} > B_{C_2}$). Hence B_C is strictly less than B_{C_1}. Likewise, when B_{C_1} is less than B_{C_2} (i.e. C_2 is *less balanced* than C_1), we also can show that B_C is less than B_{C_2} — now $\tau_2 \leq \tau_1 \leq \tau_4 \leq \tau_3$ holds. Finally, consider the case when B_{C_1} is equal to B_{C_2}. Again, due to the property of bitonic coupling, this condition implies $\tau_1 = \tau_2 = \tau_3 = \tau_4$, which is a contradiction. This completes the proof. $\square$

Theorem 6.4 contains an important subtlety. It implies the amount of inter-cluster migration is always less than that of intra-cluster migration in a critical sub-cluster. Since migrations through a higher-level link may need multi-hop communications, they result in higher overheads. Theorem 6.4 assures that the amount of migrations of such higher overheads get smaller. Consequently, the complexity of migration overheads is bounded.

6.3.2 Complexities of Task Migration Overhead

Consider the topologies in Figure 6.2 (a) and (b) extended to p processors and the total number of tasks are pN. Self-scheduling requires $pN(\alpha + \beta)$, where N is the total number of tasks between a master and its servers. Putting aside the fact that the master can easily create a bottleneck in that topology, we investigate the complexity of our method and compare it with that of self-scheduling.

The worst case happens when the fastest processor (the rightmost one in a cluster tree) is far faster than the remaining ones: i.e. $\tau_3 \gg \tau_1, \tau_4, \tau_2$ in Fig 6.2 (b). Let us calculate the overhead for a one-hop migration in this scenario. For example, in a link between $P2$ and $P3$, the total number of tasks to migrate is, by Theorem 6.1, $\frac{\tau_3-\tau_2}{\tau_3+\tau_2}N$. As $\tau_3 \gg \tau_2$, the number becomes N. In other words, all of the task in a slower processor must be migrated to the infinitely faster one. Likewise, by Theorem 6.2, the frequency of migrations is given by $\log_{\frac{1}{2}} \frac{1}{N} = \log_2 N$. Thus, the one-hop overhead (OH_1) is $\alpha \log_2 N + \beta N$. Since the farthermost tasks need $p-1$ hops, we obtain the worst case complexity of migration overhead as follows:

$$OH_{worst} = \sum_{k=1}^{p-1} k \cdot OH_1 = \frac{1}{2}p(p-1)(\alpha \log_2 N + \beta N)$$

Recalling the facts that $\alpha \gg \beta$ and $N \gg p$, OH_{worst} can hardly be worse than $pN(\alpha + \beta)$. Now let us consider an average case where each processor contains the average number of tasks (N) at any moment during computation.[2] Consider a lowest-level cluster (v, w); i.e. v and w are terminal nodes in CT. By Theorem 6.2 and 6.1, the one-hop migration overhead is obtained as follows:

$$OH_1 = \frac{1}{1 - \log_2 \frac{\tau_v}{\tau_w}} \log_2 N \cdot \alpha + \frac{\tau_w - \tau_v}{\tau_w + \tau_v} N \cdot \beta$$

By Theorem 6.4, the balance ratio of a higher-level cluster is always less than the maximum of those of the two sub-clusters. That is, the maximum balance ratio among all clusters (v, w) at the lowest level is the maximum balance ratio of all clusters in an entire cluster tree. Let it be B_{max}. Then, no $(p-1)$ links in the topology can migrate more than $B_{max} \cdot N$ tasks. Therefore, the average case complexity of migration overhead is a lower bound of the following formula, where r_{max} is the maximum of $\frac{\tau_v}{\tau_w}$ for all clusters (v, w) at the lowest level in CT:

$$OH_{average} = \sum_{k=1}^{p-1} OH_1 = \frac{p-1}{1 - \log_2 r_{max}} \log_2 N \cdot \alpha + B_{max}(p-1)N\beta$$

Notice that $0 < r_{max} < 1$ and $0 < B_{max} < 1$. $OH_{average}$ is always better than $pN(\alpha + \beta)$. Furthermore, since $\alpha \gg \beta$ and $N \gg p$, it is significantly better in general.

Example 6.3 Let us consider Figure 6.2 (b) again. Each processor initially has N identical sub-tasks. Throughputs are the same as in Example 6.1: i.e. $\tau, 2\tau, 3\tau, 4\tau$ for $P1, P2, P3$ and $P4$, respectively. For brevity, suppose all processors have constant throughputs, and we assume no migration overhead for the time being. Then the following table shows each snapshot of load distribution under our load balancing method in case we chose $\gamma_{14} = \frac{4}{5}$ and $\gamma_{23} = \frac{3}{5}$ particularly.

	$P1$	$P4$	$P2$	$P3$
Initial Load	N	N	N	N
After $N/4\tau$	$3N/4$	0	$N/2$	$N/4$
After Load Migration	$3N/20$	$3N/5$	$N/2$	$N/4$
After $N/12\tau$	$N/15$	$4N/15$	$N/3$	0
After Load Migration	$N/15$	$4N/15$	$2N/15$	$3N/15$
After $N/15\tau$	0	0	0	0

Table 6.1: Snapshots of load distribution.

The table shows that total execution time is $\frac{N}{4\tau} + \frac{N}{12\tau} + \frac{N}{15\tau} = \frac{2N}{5\tau}$; in other words, the average throughput of this 4-processor cluster with $4N$ sub-tasks is 10τ. However, the real

[2] Obviously this is a harsher condition than what a real average case needs to be, since the number of remaining tasks gets decreased as time goes by. Therefore, our obtained complexity is an upper-bound of the average complexity.

behavior deviates from this ideal behavior because of migration overheads. We calculate the overhead for two different choices of γ: when γ is taken proportionally based on throughput (**Case 1**) and when all $\gamma = \frac{1}{2}$ (**Case 2**).

Case 1: As shown in Table 6.1, migrations occur twice of amount $3N/5$ and $3N/15$, respectively. Thus, the overhead is yielded by $\alpha + \frac{3}{5}N\beta + \alpha + \frac{3}{15}N\beta = 2\alpha + \frac{4}{5}N\beta$.

Case 2: By Theorem 6.1, the number of tasks to migrate for $P1 \to P4$ and $P2 \to P3$ links is calculated as follows:

$$M_{P1\to P4} = \frac{4\tau - \tau}{4\tau + \tau}N = \frac{3}{5}N, \quad M_{P2\to P3} = \frac{3\tau - 2\tau}{3\tau + 2\tau}N = \frac{1}{5}N$$

Similarly, by Theorem 6.2, the number of migrations that occur for the two links is as follows:

$$k_{P1\to P4} = \log_{\frac{1}{2}(1-\frac{1}{4})}\frac{1}{N} = \log_{\frac{3}{8}}\frac{1}{N}, \quad k_{P2\to P3} = \log_{\frac{1}{2}(1-\frac{2}{3})}\frac{1}{N} = \log_{\frac{1}{6}}\frac{1}{N}$$

Thus, the overhead is yielded by

$$\begin{aligned} OH &= \alpha(k_{P1\to P4} + k_{P2\to P3}) + \beta(M_{P1\to P4} + M_{P2\to P3}) \\ &= \alpha(\log_{\frac{3}{8}}\frac{1}{N} + \log_{\frac{1}{6}}\frac{1}{N}) + \frac{4}{5}N\beta \approx 3.63\log N\alpha + \frac{4}{5}N\beta \end{aligned}$$

In either case, the overhead is much less than that of *self-scheduling*, which is $4N(\alpha+\beta)$. $\square$

6.3.3 Initial Load Distribution

While any initially distributed load should be balanced through a dynamic load balancing method, the resulting overhead is associated. We discuss now the initial load distribution issue that can lower overhead, compared with the equal distribution that was assumed for analysis in the previous sections.

When loops are predictable (see Section 6.1.1), there are two cases: one is when we know the amount of the required computation exactly, as in Figure 6.1 (a), (b), (c) and sometimes (d), and the other is when we can determine just the orderings, like in the DNA sequence search problem [CG89]. For the former case, as $L(i)$ is known in advance, if we distribute proportionately according to each processor's throughput, we can reduce the likelihood of migration. In other words, the processor P_i with τ_i will get $\tau_i \sum_i L(i)/\sum_k \tau_k$. Dynamic adjustments to this approximation are made by our load balancing method. In a lowest-level cluster (v, w) in CT, if we allocate $\lfloor \tau_i \sum_i L(i)/\sum_k \tau_k \rfloor$ to v, and $\lceil \tau_i \sum_i L(i)/\sum_k \tau_k \rceil$, Since v is slower than w, uni-directional migration is enough. If we cannot guarantee the faster processor finishes earlier, the migration paths must be bi-directional as in the following cases.

For the latter case, we cannot initialize in the above way as the value of $L(i)$ is unknown. The LPT (Largest Processing Time first) algorithm [BB90] is for this class of loop models. The tasks are sorted in descending order based on execution time $L(i)$. Each processor should process the largest task first. Otherwise, an unfortunate processor may happen to take a large task (say, about 100 times larger than the small ones) as a last one at the near end of all computations, which results in a load imbalance — other processors are idle because few tasks left to migrate at this moment.

When tasks are not "orderable" and quite irregular like in the Mandelbrot set computation problem, we can neither quantify the loads to proportionately distribute to processors of diverse throughputs nor sort in decreasing order and apply the LPT algorithm. No general heuristics can be used — random distribution does not need to be worse.

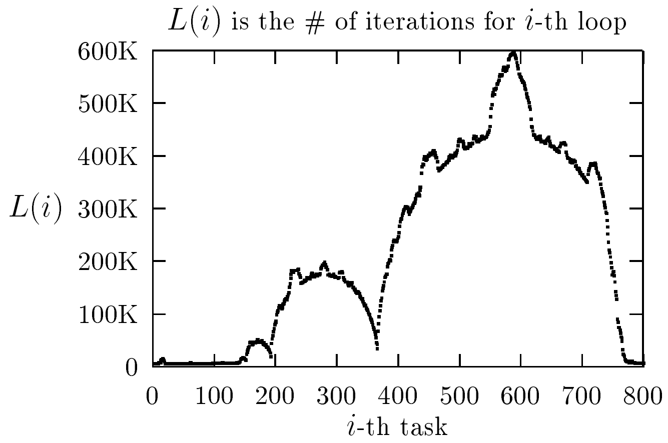


Figure 6.5: The load distribution pattern of a loop in the Mandelbrot set computation.

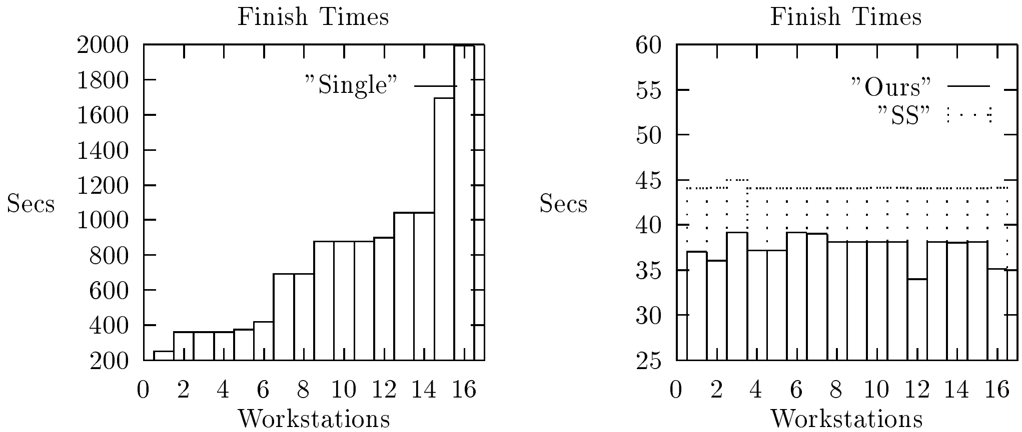


Figure 6.6: Execution times: Mandelbrot set computation on [0.5,-1.8] to [1.2,-1.2]

6.4 Experiments

To demonstrate the performance of our method, we conducted our experiment on 16 workstation clusters using PVM message passing systems. The example program was Mandelbrot set computation on $[0.5, -1.8]$ to $[1.2, -1.2]$ using a 800×800 pixel window. This program contains unpredictably irregular loops as shown in Figure 6.5, which cannot be analyzable as in Section 6.3. The x-value indicates the x-th row in an outer loop. The y-value is the number of inner iterations ($L(x)$) to compute the corresponding x-th row. The total number of sub-tasks are 800, and the result size of a sub-task is 800 in integers: one integer per pixel.

We have initially distributed those tasks in a round-robin style. A variety of heterogeneous workstations have been used as shown in Figure 6.6 (a) which shows the execution time for each of 16 workstations[3] to compute the given Mandelbrot set; the range is from 250 seconds to 2000 seconds. The results by 16-workstation cluster are given by Figure 6.6 (b). The dotted boxes represent the finish times of each workstation under the pure self-scheduling method, which substantiate the expected good load balance. The result by our method is seemingly imbalanced but the actual finish time is much improved. Perfect balance may be good but the evaluation should be based on how much its overheads negate its resulting benefits.

taskcnt	1	2	3	4	5	6	7	10	11	12	17	19	20	30
freq	13	6	3	3	3	1	4	1	1	1	1	1	1	1

Table 6.2: The sizes of migration units and the frequencies of migrations

Table 6.2 summarizes the size of each migration and its frequency that are counted in our experimentation. For example, the single-task migration occurred 13 times, and the 30-tasks migration occurred once, etc, during the entire task migration attempts. In the table, we can compute the total occurrences of migrations by summing all frequencies up, that is 40. If we calculate this figure from our formula on $OH_{average}$, that is $\frac{p-1}{1-\log_2 r_{max}} \log_2 N$, where $p = 16$, $N = 800/16 = 50$, $r_{max} = 692/693 \approx 1$. This formula gives $15 \log_2 50 \approx 84.7$. Considering this formula is obtained as an upper bound, the experimental value is said to conform to the theoretically obtained value. Although the theoretical model does not exactly match with our experimental environments, the model gives us a reasonable implication about the migration behaviors in general cases.

[3] 1 SPARCstation 20, 3 SPARCstation 5's, 2 SPARCstation 10's, 2 DECstation 5000/25's, 4 SPARCstation IPX's, 2 DECstation 23/100's, 2 SPARCstation IPC's are used.

Chapter 7

Conclusions and Future Works

For many applications, the message passing style programming is plainly too difficult to cope with all kinds of control parallel intricacies. Our approach toward distributed program optimization is based on constructing statically configurable programming environments. We have developed an automatic adaptation system that allows configuration-level optimization of RPC-based distributed programs. Because it automatically adapts the application at the source level, it encourages programmers to experiment with various performance improvement strategies in order to discover the best for their environment and data. Programming directly in terms of message passing primitives may still give programmers the maximum ability to write high-performance programs in distributed environments, but this freedom comes at a high price in programmer time and effort, and reduces the programmer's freedom to port, upgrade or reuse the component program units. These benefits have been available because many types of performance factors are isolated from the module programming level and deferred such decisions to the configuration level. Therefore, our approach helps to decrease the code of developing and tailoring application programs, while at the same time achieving overall performance comparable to manually tailored counterparts, which has been sacrificed before.

Exploiting parallelism is a complicated process. Data parallelism has been widely accepted while control parallelism has been used only at the low-level procedural message passing systems like PVM and MPI. Higher level systems still have not displaced such a low-level system, although the low-level systems are usually understood as error-prone and tedious to use. As we have discussed in Chapter 3, many practical distributed and parallel applications can be expressed in a modular way using procedure call abstraction. The previous unavailability of proper optimization methods discouraged programmers from using the RPC paradigm for higher performance oriented distributed programming, in spite of its convenience and simplicity owing to its high abstraction power. We have presented a source-level transformation framework for RPC-based distributed programs, whose goal is automatically extracting RPC task-level parallelism and reducing the communication paths according to the constraints of data and control dependences.

We also have presented a new decentralized load balancing method for parallel tasks in heterogeneous workstation clusters to deal with various patterns of parallel loops. We discussed why the conventional global dynamic load balancing methods are not adequate to our application areas. Loop scheduling schemes that have been useful under shared-memory multiprocessor machines cause a bottleneck in workstation cluster environments because the communication overheads are higher. To our knowledge, migration topology for load balancing is considered for the first time. The topology has not been considered important heretofore because sometimes it is given in a hard-wired form [LK87] or it is meaningless where distributed load patterns cannot be assumed to be known in advance [ELZ86b, KS94]. We have shown analytically that the overhead of our method is lower than that of the self-scheduling scheme when an "predictability" condition is given. We have also provided some experimental data for cases when the loop pattern is unpredictably irregular. Most striking fact from our new load balancing scheme is its relevance to the configuration programming. As our suggested topology is of binary tree form, more interesting topologies can be studied analytically or experimentally in the future.

Assessing the usability of a parallel programming system is one of important research areas from the viewpoint of software engineering. Wilson [Wil94] proposes 9 applications that can assess how well a parallel programming system can support large scale software engineering and how easily systems can be learned or how quickly code can be developed. Prior to that Feo [bJTF92] suggested *Salishan Problems* and collected results for the problems from various parallel programming systems — all at the module programming level. The comparison goals in the assessments were two-fold. First, how well parallel programming system can support large-scale software engineering concerns based on software engineering metrics like LOC (Lines of Codes), Halstead's "program volume" measure, and McCabe's "cyclometic complexity decomposition of flow graphs." Second, how easily systems can be learned or how quickly code can be developed by measuring the time taken from novice to expert. Assessing our framework using the same problems and the same criteria will substantiate the usability of our framework.

There is also an interesting research direction in regards with extending to a visual programming environment. Textual MIL programming can still be a nuisance to programmers who write large distributed applications that consist of many software components. If hundreds of slave processes are involved, the load balancing expressions in the configuration program may exceed hundreds of lines as well. A visual approach is the alternative such as Newton's graphical environment for parallel programming [New93]. A graph editing tool that is capable of processing all necessary attributes in configuration-level programming can help to deal with a large program of many components. The tool may produce the textual equivalent MIL program as an output.

Another important research direction is improving the program performance through more aggressive program transformations that have been successful in general compiler-assisted optimization areas. Especially, it is interesting to see there is an analogy between the transformation-based RPC compilation and RISC (Reduced Instruction Set Computers) compilers. It is illustrated in Figure 7.1 and Figure 7.2. Figure 7.1 (b) shows that the CISC code needs six memory accesses while the RISC one needs only four at (1), (2), (4), (6) in Figure 7.2 (c). This is due to an optimization known as spill code reduction. Figure 7.2

`(1) A := B + C`	`(1) ADD A, B, C`	`(1) LD  rB,B`
`(2) B := A + C`	`(2) ADD B, A, C`	`(2) LD  rC,C`
		`(3) ADD rA,rB,rC`
		`(4) ST  A,rA`
		`(5) ADD rB,rA,rC`
		`(6) ST  B,rB`
`(a) High-level`	`(b) CISC`	`(c) RISC`

Figure 7.1: Generated codes with RISC and CISC instructions.

`(1) a = f(a, b)`	`(1) Send(f(), a, b)`	`(1) Send(f(), a, b)`
`(2) b = g(a, c)`	`(2) a = Receive(f())`	`(2) Send(g(), c)`
	`(3) Send(g(), a, c)`	`(3) a = Receive(f())`
	`(4) b = Receive(g())`	`(4) b = Receive(g())`
`(a) RPC code`	`(b) Compiled code`	`(c) Optimized code`

Figure 7.2: Spill code reduction in a distributed program.

shows the effect of the similar optimization, that is already presented in **Algorithm 5.2**. This is because a memory access in RISC programs appears in a form of message passing in RPC programs. More important aspects of RISC compilers are on instruction scheduling. Basically RISC programs contain many pipeline bubbles (NOP: No OPeration) to avoid pipeline interlocks [Pat85]. Instruction scheduling is how to fill up those bubbles with useful operations safely without changing program semantics. Moreover, message passing primitives also have varieties in its functional complexity. For example, in PVM, `pvm_psend()` packs and sends a set of data while there are specific primitives of `pvm_pack()` for packing and `pvm_send()` for sending. Using reduced primitives allows more leeway for aggressive transformations.

Our research strategy shows how to enhance current interconnection technologies to support automatic analysis and tailoring of distributed applications for use in a wide range of target application environments. As a result of our work, programmers can build applications using simple structures that are easy for them to reason about, yet still have performance improvements that would have been very complex and costly for them to build-in manually, with an aid of compiler optimization techniques.

Bibliography

[ABLL92] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[AL93] S. P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 126–138, June 1993.

[And91a] G. R. Andrews. *Concurrent Programming: Principles and Practice*. The Benjamin/Cummings Publishing Co., Inc., 1991.

[And91b] G. R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, Vol. 23(1), March 1991.

[ARZ92] F. Allen, B. K. Rosen, and K. Zadeck. *Optimization in Compilers*. ACM Press, 1992.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[ATK91] A. L. Ananda, B. H. Tay, and E. K. Koh. Astra – An asynchronous remote procedure call facility. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 172–179, 1991.

[Bar78] J. M. Barth. A practical interprocedural data flow analysis algorithm. *Communication of the ACM*, Vol. 21(9):724–736, September 1978.

[BB90] K. P. Belkhale and P. Banerjee. An approximate algorithm for the partitionable independent task scheduling problem. In *Proceedings of '90 International Conference on Parallel Processing*, August 1990.

[BDZ88] P. A. Buhr, Glen Ditchfield, and C. R. Zarke. Adding concurrency to a statically type-safe object-oriented programming language. In *Proceedings of the ACM SIGPLAN Workshop on object-based concurrent programming*, pages 18–21, September 1988.

[BELL89] B. N. Berstad, T. E.Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. In *Proceedings of 12th Symposium on Operating Systems Principles*, pages 102–113, 1989.

[BELL90] B. N. Berstad, T. E.Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, Vol. 8(8):37–55, February 1990.

[bJTF92] Edited by J. T. Feo. *A Comparative Study of Parallel Programming Languages: The Salishan Problems*. North-Holland, 1992.

[BL93] R. Butler and E. Lusk. Monitors, messages and clusters: The p4 parallel programming system. Technical Report MCS-P362-0493, Argonne National Laboratory, Argonne, IL, 1993.

[BLA+93] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual memory mapped network interface for the shrimp multicomputer. Technical Report TR–437–93, Princeton University Computer Science Department, November 1993.

[BMW85] W. C. Brantley, K. P. McAuliffe, and J. Weiss. RP3 processor-memory element. In *Proceedings of the 1985 International Conference on Parallel Processing*, August 1985.

[BN84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, Vol. 2(1):39–59, February 1984.

[BR89] L. Bomans and D. Roose. Benchmarking the iPSC/2 hypercube multiprocessor. *Concurrency: Practice and Experience*, Vol. 1(1):3–18, September 1989.

[BST89] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, Vol. 21(3):260–322, September 1989.

[Cal88] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 47–56, June 1988.

[Car93] John B. Carter. *Efficient Distributed Shared Memory Based On Multi-Protocol Release Consistency*. PhD thesis, Rice University, sep 1993.

[CFR+91] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and systems*, 13:451–490, October 1991.

[CG89] N. Carriero and D. Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, Vol. 21(6):322–356, September 1989.

[CG90] N. Carriero and D. Gelernter. *How to write parallel programs: A first course.* MIT Press, 1990.

[CLZ95] M. Cierniak, W. Li, and M. J. Zaki. Loop scheduling for heterogeneity. In *Proceedings of the 4th International Symposium on High-Performance Distributed Computing*, August 1995.

[CMZ92] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, 1992.

[Coo80] R. P. Cook. *MOD – A language for distributed programming. *IEEE Transactions on Software Engineering*, Vol. 6(6), November 1980.

[Coo85] E. C. Cooper. Replicated Distributed System. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 63–78, 1985.

[Cor91] John R. Corbin. *SUN RPC: The art of distributed applications: programming techniques for remote procedure calls.* Springer-Verlag, 1991.

[CP91] J. R. Callahan and J. M. Purtilo. A packaging system for heterogeneous execution environments. *IEEE Transactions on Software Engineering*, Vol. 17(6):626–635, June 1991.

[CS93] Clemens H. Cap and Volker Strumpen. Efficient parallel computing in distributed workstation environments. *Parallel Computing*, Vol. 19:1221–1234, 1993.

[DK76] F. DeRemer and H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, Vol. 2(2), June 1976.

[DMSM94] H. G. Dietz, T. Muhammad, J. B. Sponaugle, and T. Mattox. PAPERS:Purdue's Adapter for Parallel Execution and Rapid Synchronization. Technical Report TR–EE94–11, Purdue University School of Electrical Engineering, March 1994.

[ELZ86a] D. L. Eager, E. D. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance Evaluation*, Vol. 6:53–68, 1986.

[ELZ86b] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, Vol. 12(5):662–675, May 1986.

[Fel79] J. A. Feldman. High level programming for distributed computing. *Communication of the ACM*, Vol. 22(6), June 1979.

[FJL+88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice Hall, 1988.

[For93] The MPI Forum. MPI: A Message Passing Interface. In *Proceedings Supercomputing '93*, pages 878–883, 1993.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and systems*, 9(3):319–349, July 1987.

[FvDF+93] J. D. Foley, A. van. Dam, S. K. Feiner, J. F. Hughes, and R. L. Phillips. *Introduction to Computer Graphics*. Addison-Wesley Publishing Company, 1993.

[GBSS89] J. L. Gustafson, R. E. Benner, M. P. Sears, and T. D. Sullivan. A radar simulation program for a 1024-processor hypercube. In *Proceedings of SuperComputing 1989*, pages 96–105, 1989.

[Geh84] N. H. Gehani. Broadcasting Sequential Processes (BSP). *IEEE Transactions on Software Engineering*, Vol. 10(4), July 1984.

[Geh86] N. H. Gehani. Concurrent C. *Journal of Software Practice and Experience*, Vol. 16:821–844, September 1986.

[Geh90] N. H. Gehani. Message passing in concurrent C: Synchronous versus asynchronous. *Journal of Software Practice and Experience*, Vol. 20(6):571–592, June 1990.

[Gen81] W. M. Gentleman. Message passing between sequential processes: The reply primitive and the administrator concept. *Journal of Software Practice and Experience*, Vol. 11:435–466, May 1981.

[GG88] D. K. Gifford and N. Glasser. Remote pipes and procedures for efficient distributed communication. *ACM Transactions on Computer Systems*, Vol. 6(3):258–283, August 1988.

[Gib87] Philip B. Gibbons. A stub generator for multilanguage RPC in heterogeneous environments. *IEEE Transactions on Software Engineering*, Vol. 13(1), January 1987.

[GP92] M. Girkar and C. D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3(2):166–178, March 1992.

[GWWECL94] A. S. Grimshaw, J. B. Weissman, E. A. West, and Jr. E. C. Loyot. Metasystems: An approach combining parallel processing and heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, Vol. 21:257–270, 1994.

[Hal85] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and systems*, Vol. 7(4):501–538, October 1985.

[Hal90] Mary W. Hall. *Managing Interprocedural Optimization*. PhD thesis, Rice University, oct 1990.

[HKT92] S. Hiranandani, K. Kennedy, and C.-W Tseng. Compiling fortran D for MIMD distributed-memory machines. *Communication of the ACM*, 35(8):66–80, August 1992.

[HL82] M. Herlihy and B. Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and systems*, 4(4):527–551, October 1982.

[Hoc94] R. W. Hockney. The communication challenge for MPP: Intel paragon and meiko CS-2. *Parallel Computing*, 20(3):389–398, March 1994.

[HQ91] P. J. Hatcher and M. J. Quinn. *Data-parallel programming on MIMD computers*. MIT Press, 1991.

[JZ93] D. B. Johnson and W. Zwaenepoel. The Peregrine high–performance RPC system. *Journal of Software Practice and Experience*, Vol. 23(2):201–221, February 1993.

[KLS+94] C. Koelbel, D. Loveman, R. Schreiber, Jr G. Steele, and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[KP95] T.-H. Kim and J. M. Purtilo. Configuration-level optimization of RPC-based distributed programs. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, May 1995.

[KP96a] T.-H. Kim and J. M. Purtilo. Load balancing for parallel loops in workstation clusters. In *Proceedings of the 25th International Conference on Parallel Processing*, pages III:182–190, August 1996.

[KP96b] T.-H. Kim and J. M. Purtilo. A source-level transformation framework for RPC-based distributed programs. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, pages 78–87, August 1996.

[Kra90] J. Kramer. Configuration programming — a framework for the development of distributable systems. In *Proceedings of the IEEE International Conference on Systems and Software Engineering (CompEuro 90)*, May 1990.

[KS94] Philip Krueger and Niranjan G. Shivaratri. Adaptive location policies for global scheduling. *IEEE Transactions on Software Engineering*, Vol. 20(6):432–444, June 1994.

[KW85] C. P. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, Vol. 11(10):1001–1016, October 1985.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communication of the ACM*, Vol. 21(7):558–565, July 1978.

[LHG86] B. Liskov, M. Herlihy, and L. Gilbert. Limitations of synchronous communication with static process structure in languages for distributed computing. In *Proceedings of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 150–159, 1986.

[LK87] Frank C. H. Lin and Robert M. Keller. The gradient model load balancing method. *IEEE Transactions on Software Engineering*, Vol. 13(1):32–38, January 1987.

[LS88] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 260–267, June 1988.

[MBR87] Bruce Martin, Charles Bergan, and Brian Russ. PARPC: A system for parallel remote procedure calls. In *Proceedings of the International Conferences on Parallel Processing*, pages 449–452, 1987.

[New93] Peter W. Newton. *A Graphical Retargetable Parallel Programming Environment and Its Efficient Implementation*. PhD thesis, The University of Texas at Austin, dec 1993.

[Pat85] D. Patterson. Reduced Instruction Set Computers. *Communications of ACM*, pages 8–21, Jan 1985.

[PK87] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computer*, Vol. C-36(12):1425–1439, December 1987.

[Pol88] C. D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer Academic Publishers, 1988.

[Pur94] J. M. Purtilo. The polylith software bus. *ACM Transactions on Programming Languages and systems*, Vol. 16(1):151–174, January 1994.

[PW86] D. A. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communication of the ACM*, Vol. 29(12):1184–1201, December 1986.

[RB93] S. Ramaswamy and P. Banerjee. Processor allocation and scheduling of macro dataflow graphs on distributed memory multicomputers by the PARADIGM compiler. In *Proceedings of the 22nd International Conference on Parallel Processing*, 1993.

[SA89] M. Sullivan and D. Anderson. Marionette: a system for parallel distributed programming using a master/slave model. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 181–188, 1989.

[SB90] M. Schroeber and M. Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[SLR+95] E. Su, A. Lain, S. Ramaswamy, D. Palermo, E. Hodges IV, and R. Banerjee. Advanced compilation techniques in the PARADIGM compiler for distributed-memory multicomputers. In *Proceedings of the International Conference on Supercomputing '95*, 1995.

[Son94] Jianjian Song. A partially asynchronous and iterative algorithm for distributed load balancing. *Parallel Computing*, Vol. 20:853–868, 1994.

[SS86] M. Satyanarayanan and E. H. Siegel. MultiRPC: A parallel remote procedure call mechanism. Technical Report CMU-CS-86-139, Carnegie-Mellon University, 1986.

[SS94] B. K. Schmidt and V. S. Sunderam. Empirical analysis of overheads in cluster environments. *Concurrency: Practice and Experience*, Vol. 6(1):1–32, February 1994.

[Sun90] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, Vol. 2(4):315–339, December 1990.

[TN91] T. H. Tzen and L. M. Ni. Dynamic loop scheduling for shared-memory multiprocessors. In *Proceedings of '91 International Conference on Parallel Processing*, pages II:247–250, August 1991.

[TY86] P. Tang and P. C. Yew. Processor self-scheduling for multiple nested parallel loops. In *Proceedings of '86 International Conference on Parallel Processing*, pages 528–535, August 1986.

[vHK94] R. von Hanxleden and K. Kennedy. Give–N–Take — A balanced code placement framework. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 107–120, June 1994.

[WFN90] E. F. Walker, R. Floyd, and P. Neves. Asynchronous remote operation in distributed systems. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 253–259, 1990.

[Wil94] Gregory V. Wilson. Assessing the usability of parallel programming systems: The cowichan problems. In *Proceedings of the IFIP WG 10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems*, April 1994.

[YBS86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *Proceedings of OOPSLA '86*, pages 258–268, 1986.