A hybrid load balancing algorithm for coarse-grained applications.

Xinyu. Gan  
*University of Windsor*

Follow this and additional works at: [https://scholar.uwindsor.ca/etd](https://scholar.uwindsor.ca/etd)

**Recommended Citation**
[https://scholar.uwindsor.ca/etd/642](https://scholar.uwindsor.ca/etd/642)

This online database contains the full-text of PhD dissertations and Masters’ theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.
A Hybrid Load Balancing Algorithm for Coarse Grained Applications

by

Xinyu Gan

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science
in Partial Fulfilment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada
2003
© 2003 Xinyu Gan
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.
Abstract

A non-preemptive hybrid load balancing algorithm is proposed for heterogeneous distributed computing environment, since no single load balancing algorithm works well for all kinds of applications and environments. The agents' computing capabilities may also change during runtime because of the background load. This algorithm makes use of the idea of several sub-algorithms. The hybrid model initially classifies the computers and jobs into different groups. Two priority queues are maintained at each worker to record the processes' estimated computing time and the real time. Based on historical experiences, a centralized scheduler can dynamically change the parameters in order to improve the overall performance during runtime. The algorithm balances the work load of coarse-grained applications with interdependent processes such as matrix computation or image processing.

Keywords: load balancing, adaptive load balancing, centralized load balancing, prediction, priority based scheduling, distributed computing, heterogeneous computing environment, image processing, WebDedip, learning system.
Acknowledgements

I would like to express my sincere thanks to my supervisor Dr. Akshai Aggarwal for his guidance and direction throughout this thesis. I also thank him for giving me the opportunity to study for this Master's degree. Thanks Dr. Ahmed Tawfik for his comments and suggestions on the survey. I wish to thank Dr. Xiang Chen, Dr. Alioune Ngom and Dr. Arunita Jäckel for serving on my thesis committee. I thank them for their suggestions, valuable time and comments.

I also wish to thank my grandmother, my parents and my brother for their support. Thanks graduate secretary Ms. Mary Mardegan and all my colleagues and friends for their direct and indirect support helping me to complete my thesis in time.
# Table of Contents

Abstract iii

Acknowledgements iv

List of Figures ix

1 Introduction 1

1.1 Scheduling Approaches .............................................. 1

1.1.1 Heuristic Approaches ............................................. 1

1.1.2 Scheduling Based On Fuzzy Logic ................................. 2

1.1.3 Queuing Algorithm ................................................ 2

1.1.4 Threshold Based Load Balancing ................................. 3

1.1.5 Eager Scheduling, Lazy Scheduling and Parallel Scheduling .... 4

1.2 Grain Size .................................................................... 5

1.3 Motivation and Contribution of the Thesis .......................... 5

2 Previous/Related Work 7

2.1 Distributed Computing System and Metacomputer .................. 7
2.2 Load Balancing in Distributed Computing ........................................... 8

2.2.1 Static Approaches vs. Dynamic Approaches ................................. 9

2.2.2 Adaptive Load Balancing Algorithm's Components ..................... 10

2.2.3 Issues ............................................................................................. 14

2.3 Process Migration .............................................................................. 15

2.3.1 Preemptive Migration ................................................................. 16

2.3.2 Non-preemptive Migration ............................................................ 16

2.4 Implementation Challenges ............................................................. 17

2.4.1 Heterogeneity ............................................................................... 17

2.4.2 Expensive Communication ......................................................... 17

2.4.3 Security ......................................................................................... 18

2.4.4 Dynamic Process Management ................................................... 18

2.4.5 Compatibility ................................................................................. 18

2.4.6 Some Constraints ......................................................................... 19

2.5 Techniques Used to Meet These Challenges ..................................... 19

2.6 Metacomputing Frameworks ............................................................. 20

2.6.1 MPI and CALYPSO ...................................................................... 21

vi
2.6.2 PVM and JPVM ........................................... 22
2.6.3 Bayanihan and Javelin ................................. 23
2.6.4 Charlotte and Globus .................................. 24
2.6.5 DEDIP and WebDedip ................................. 25
2.6.6 Dome and UTOPIA ..................................... 27
2.6.7 TreadMarks and ATLAS ............................... 27
2.6.8 HARNESS and ParaWeb ............................... 28
2.6.9 Condor and SNIPE ..................................... 29
2.6.10 Legion and KnittingFactory ......................... 30

3 Hybrid Algorithm Model ................................. 32

3.1 Preliminary .............................................. 33
3.2 the Proposed Model ..................................... 33
3.3 System Overview ....................................... 38
3.4 Main Algorithm ......................................... 39
3.5 Complexity of the Algorithm ......................... 40

4 Implementation Issues and Case Study .................. 43
4.1 Case Study ......................................................... 43

4.2 Discussion of the Result ................................. 47

5 Conclusion and Future Work ............................. 48

References ......................................................... 49

Vita Auctoris ....................................................... 57
# List of Figures

1. Interdependency Graph of Application 1: An Example .......... 5
2. Load Balancing .................................................. 9
3. System Overview ................................................. 39
4. Flow Chart: Hybrid Algorithm ................................... 41
5. Interdependency Graph of Application 1 ......................... 44
6. Interdependency Graph of Application 2 ......................... 44
7. Interdependency Graph of Application 3 ......................... 44
8. Flow Chart: Eager Scheduling .................................... 45
9. Performance for Interdependent Processes ....................... 46
10. Performance for Independent Processes .......................... 46
11. Performance When Increasing the Number of Applications .... 46
1 Introduction

In a heterogeneous distributed computing environment, decomposition of a problem and load balancing is critical to the performance of applications. Each node of the whole network should have nearly the same amount of work load to minimize the execution time of the application. The abstract goal of load balancing discussed here is that "given a collection of tasks comprising a computation and a set of computers on which these tasks can be executed, find the mapping of tasks to computers that minimizes the run time of the computation [31]."

In a homogeneous environment, an equal partition of work will lead to good load balance, but will not work well in a heterogeneous environment [6]. Apparently faster computers should be allocated more jobs than slower ones so that the differences of finishing time of each computer can be minimized.

1.1 Scheduling Approaches

1.1.1 Heuristic Approaches

"The problem of optimal load balancing is NP complete [24]." Heuristic approaches [52][24][59][34] are simple but suboptimal solutions. They recalculate the algorithms' parameters based on historical experiences. They work well under certain circumstances such as a linear combination of the n-dimensional input vector and the weight vector [34]. When considering a more general situation, it is difficult to incorporate
them into a perfect mathematical model. After a certain period of time, we use
collected system state information to change some of the algorithm’s parameters in
order to make a better prediction of the next round of execution. For example, the
processing ability of each node may vary from time to time if there are some back-
ground loads of other applications. With the updated parameters, the scheduler can
make better prediction as time goes by.

1.1.2 Scheduling Based On Fuzzy Logic

There are a number of factors that affect the decision of scheduling a process. It is
impossible and inefficient to take into account all the variables in a load balancing
algorithm. Fuzzy logic [53] may be one of the ways to solve this problem. Both
resources and processes can be categorized into coarse grained classes. For example,
processes can be differentiated into 2 priority levels: normal and top priority [28].
A fuzzy controller determines when and where a certain type of process is to be
allocated. By its nature, it can not balance the load perfectly. However, it is a good
method for complex systems because the algorithm itself is simpler and consumes less
computing resources. In the proposed hybrid algorithm, we use this idea to classify
the computers and jobs into coarse grained groups.

1.1.3 Queuing Algorithm

Predefined inter-dependency processes are denoted in a tree architecture. The system
has to maintain two queues: wait and ready-to-run [28]. FIFO [38] and SJF [48]
(shortest job first) are two basic policies for managing queues. Normally shorter processes should be executed early so that they have less chance to be delayed by longer processes.

Before execution of the applications, the operator may have some idea of the nature of the processes. In WebDedip [29] system, the operator can manually assign them to the appropriate position of a waiting queue. After some applications have started execution, newly arrived processes (CPU-bound processes) may be allocated by the central scheduler to the most light-loaded node and put to a position according to FIFO [38] policy. In the proposed hybrid algorithm, two priority queues are maintained at each worker to record the processes' estimated computing time and the real time.

1.1.4 Threshold Based Load Balancing

Work load description may be very complex. Several previous research works [35] [17] indicate that "simple work load descriptions and load sharing policies perform as well as or even better than complex ones [35]". When job transfer and system state information exchange overheads are nonnegligible at heavy system loads, static load balancing policies can provide more stable performance and better than dynamic policies [61]. Again, the reason is the increasing overhead of the sophisticated descriptions and policies. Especially when the nodes of the whole system scale to a large number, it is more difficult to measure and broadcast system state information using sophisticated descriptions. However, dynamically changing threshold value of work-
ing nodes are necessary for the scheduler to make process migration decisions. In the proposed hybrid algorithm, processes' average estimated computing time is used as the metric of the work load.

1.1.5 Eager Scheduling, Lazy Scheduling and Parallel Scheduling

A job management server is required for eager scheduling [3]. It decomposes the whole application into several sub jobs or processes and allocates them to available client machines. The identical copy of a sub job may be allocated to other machines if the server can not get the result of the sub job in a period of time. In this way, faster machines may be able to do more jobs and slower machines are also utilized. The system also has tolerance if client machines (workers) happen to be down. The weaknesses of the algorithm are that there is a single point of failure (the server) and the user has no direct control of resource allocation.

Lazy scheduling postpones scheduling tasks because tasks with different size may block each other with first-come-first-serve scheme. One of the methods is to maintain separate queues in the system and delay scheduling jobs if there are any other jobs with the same size running [60].

In parallel scheduling, all processors cooperate together to schedule the workload based on global load information. The idea is to use several schedulers to identify a set of tasks and to allocate them in parallel [60].

We use eager scheduling for comparing with the proposed hybrid algorithm.
1.2 Grain Size

Grain size can be interpreted as the amount of computation each task (or process) requires.

![Diagram of interdependency graph]

Figure 1: Interdependency Graph of Application 1: An Example

Figure 1 shows the interdependency relationship of processes in an application. Generally, increasing grain size reduces communication between processes. However, increasing grain size also reduces parallelism [54]. Normally, we will decompose a problem in larger grains rather than in too small grains because of the complexity and the communication overhead [6].

1.3 Motivation and Contribution of the Thesis

Since no single scheduling method can solve all problems. We use a combination of the selected algorithms and take advantage of their strengths. Several metacomputing frameworks have been presented in literature which use one or more of the above scheduling methods. We will also provide a working solution which uses the hybrid
load balancing algorithm.

The contribution of this thesis is that we propose a new hybrid load balancing algorithm which can be implemented over heterogenous distributed computing environment. Estimated processes' computing time is used as the metric of work load. Stand alone java applications act as a software layer on top of the existing operating systems which take care of both independent and interdependent processes. Experiment shows that the proposed algorithm performs relatively better under both circumstances.

The rest of the thesis is organized as follows: In Chapter 2, we present a review of different load balancing approaches and metacomputing frameworks. A detailed description of the proposed algorithms is presented in Chapter 3. Chapter 4 takes care of implementation issues and case study while Chapter 5 is the conclusions and discussions on future work.
2 Previous/Related Work

Several load balancing techniques for distributed computing environment have been presented in literature. However no single approach and framework can solve all the problems and provide really easy to use interfaces.

2.1 Distributed Computing System and Metacomputer

Independent computers, which run their own operating systems, are connected together to become a distributed computing system [46]. In distributed systems over the networks of workstations (NOW), normally one needs to install the same software for each node to manage the resources of the whole system. In this way the user can easily access system resources such as memory, processors, and the network transparently, irrespective of whether they are available locally or remotely. At anytime, a point of failure should be recoverable without affecting the user’s applications.

Metacomputers have much in common with both distributed and parallel systems. However, the heterogeneous and dynamic nature of metacomputing systems limits the applicability of current parallel computing tools and techniques [20]. We might consider the Internet as one giant metacomputer.

The user’s view of a metasystem or distributed computing environment should be one of a monolithic virtual machine that provides computational and data storage services. The user does not want to be concerned with the details of machine and
processor type, data representation and physical location of processors and data, or the existence of other competing users [6]. The users will have the illusion of a very powerful computer on the desk. They will sit at the terminal and manipulate objects. The objects will represent data resources such as digital libraries or video streams, applications such as teleconferencing or physical simulations [25].

2.2 Load Balancing in Distributed Computing

As mentioned before, each node of the whole system should have nearly the same amount of workload in order to minimize the total execution time. One of the methods to indicate load imbalance is to measure the time difference between the first and last processor to finish their part of the job [45]. For example in figure 2, the job is to minimize the difference between $t_0$ and $t_1$. Load balancing strategies can be categorized into two groups: static approaches and dynamic approaches.
2.2.1 Static Approaches vs. Dynamic Approaches

A static approach decides the allocation of the application components before execution and can take into account both execution and communication needs [5]. Static approach works well for certain problems for which the application's behavior can be accurately predicted. Dynamic load balancing is more suitable for irregular problems.

A dynamic approach has to collect the system state information and make its allocation decisions at run time [5]. Dynamic load balancing has been considered in different contexts, e.g. scheduling and migration of tasks in operating systems [9][16][39][57], distributed solving of combinatorial optimization problems [11] [41][42], distributed solving of problems in the area of scientific computing [21] [58], and others. Several classification schemes have been proposed [33][49], e.g. local vs. global decision and
local vs. global migration, centralized vs. distributed algorithms [12].

The chief advantage of the static policies is their simplicity. (i.e., no need to collect system state information). The disadvantage of these policies is that they cannot respond to changes in system state (thus the performance improvement is limited). Adaptive policies react to the system state changes dynamically and therefore provide substantial performance improvements over the static policies. However, this additional performance benefit is obtained at a cost of collecting and maintaining the system state information [16]. It is the designer, the programmer or the user's responsibility to decide the tradeoffs. Ideally the users can easily configure the whole system and make their own decisions.

2.2.2 Adaptive Load Balancing Algorithm's Components

1. Processor load measurement:

It is extremely difficult to measure the work load in distributed systems. A number of possible metrics have been studied in the past including processor queue length, average processor queue length over some period of time, amount of memory available, context switch rate, system call rate, CPU utilization, etc [26]. A candidate load index should be easy to compute and correlate well with the parameter (such as the process response time) that one wishes to optimize. Some systems use multiple load indices for making process placement decisions. In a heterogeneous environment, the load indices from different nodes must be adjusted to make them comparable. For example, if two different nodes
have different processing power, their CPU utilization may have to be divided by their processing power for comparison of their CPU utilization load index values [26].

2. The system load information exchange policy:

Some issues related to information policy, for example, are what information is to be collected, where the information is to be collected and when to collect the information [26]. The information can be collected in a single node in the system leading to a centralized algorithm for allocation, or can be totally or partially collected in every node, leading to distributed algorithms. The trade-off in information collection is that collecting information frequently helps make better policy decisions but also leads to greater overhead. In situations of heavy traffic, the nodes should refrain from information exchange activity and resort to estimating the system state using local information or by predicting the state of the network using previous information [51].

The information collection can be demand driven where the node gathers information about the other nodes only when needed or periodic. Or state driven where nodes disseminate information about their states whenever their states change by a certain degree.

For some applications a good scheduler can reduce the interprocessor communications to the point where the modest communication performance does not degrade the overall application performance [10]. The best choice of frequency of activation is highly application and platform dependent and cannot be found
3. The transfer policy:

The transfer policy decides which processes (large, small or newly arriving) are best fit for migration. For example, blocked processes are not available for migration, since this usually does not affect the local processor load. Limiting the number of migrations to a predefined value could be a conservative approach to maintain stability and can also achieve a better performance [51]. Threshold policy designates a node as a sender if the load index of the node exceeds a threshold value (expressed in the unit of load). Another transfer policy is the relative policy, which considers the load of a node in relation to loads at other system nodes [26].

4. The cooperation and location policy:

The choice involved is whether the process allocation decision is taken by a single node or the responsibility is shared between nodes in the system. Centralized algorithms are simpler to implement. However, they do not scale up well since the coordinator can become the bottleneck.

In the non-cooperative allocation, individual processors act as autonomous entities and arrive at decisions regarding the use of their resources independent of the effect of their decision on the rest of the system. In the cooperative case, each processor has the responsibility to carry out its own portion of the scheduling task, but all processors work together to achieve a system-wide goal.
Location policy answers the question "where is the best location for migrating a process [51] ?":

- Sender initiated: An overloaded processor initiates locating a suitable under-loaded processor.

- Receiver initiated: The under-loaded processor tries to "steal" some work from the busy one.

At low or moderate system loads, the probability of finding a lightly loaded node is higher than that of finding a heavily loaded node. So the sender-initiated policies works well. At high system loads, on the other hand, receiver-initiated policies are better because it is much easier to find a heavily loaded processor in these system loads.

The location policies can also employ either a centralized or a distributed location policy. In the centralized policy, state information is collected by a single node and other nodes would have to consult this node for advice on the system state. In the distributed policy, system state information is distributed to all nodes in the system. However, distributed policy may cause performance problems as the state information may have to be transmitted to all nodes in the system [16].

A common location policy is polling. Polling could be serial or parallel with regard to how many nodes can be polled at a time (parallel polling requires
multi-cast or broadcast message passing facilities) and random or based on the nearest neighbor. Among the other options for location policy are broadcast queries where a node broadcasts a message and the eligible recipients reply, from which one is selected as the peer [26].

5. Conclusion of load balancing policies

No single load balancing policy can work equally well to solve all kind of problems. Some policies produce better results for slowly dynamic applications while others may be suitable for highly dynamic ones. Automatically collecting the system state information and changing the policies accordingly may be the way to achieve a "better" load balancing policy [5].

2.2.3 Issues

The overhead incurred with a load-balancing algorithm sometimes may trade off any improved performance gained by this scheme. Load sharing scheme is a weaker version of load balancing which tries to initiate a process to lightly loaded node and hence distribute the overall load of the system to its individual nodes using only non-preemptive transfer of processes.

Some of the goals are contradictory to each other. For example, in order to maximize throughput, load needs to be evenly spread among the processors of system but it may increase inter-processor communication cost if highly interacting processes are assigned to different processors. Similarly algorithms susceptible to changes in load
will be more complex [26]. Often it is seen that a load sharing policy performs well in low system load condition while the performance degrades drastically at higher load. This is because processor cycles wasted by the allocator in doing a "good" allocation may be too expensive at higher loads. Hence good policies must be stable and provide reasonable response at all system loads.

In summary, suitable load balancing techniques and process migration are necessary and the cost to achieve a balanced system must also be considered.

2.3 Process Migration

Process migration is necessary when system tries to balance the workload. It can be implemented at the user level or at the kernel level. At the kernel level, we have to modify and recompile the system kernel itself. The advantage of user level implementations is the ease of portability to different operating systems. However, these systems are typically slower than ones in which migration is done at the kernel level. It takes time to transfer processes from one node to the other. It also takes time for the source host and target host to package and unpack the transferred pages [27]. Moreover, user level implementation also has certain restrictions because of access restrictions to kernel data structures [26].
2.3.1 Preemptive Migration

Preemptive migration requires the system to checkpoint the state of a process already in execution and then transfers this state to the target machine. Clearly this is a very hard problem if the two machines are architecturally different. Further, the migration process is also more costly in terms of time since the entire state (which might be quite large) has to be transferred to the destination machine.

For preemptive process migration the selection policy should choose processes which are small (so that the overhead incurred by the transfer is less), long lived (so that it is worthwhile to incur the transfer overhead) and that are likely to make fewer location dependent system calls [26].

2.3.2 Non-preemptive Migration

Non-preemptive migration does not incur much additional cost since in the simplest possible case the selection policy selects a newly initiated process as the one to be transferred. However, in most practical cases, non-preemptive scheduling leads to poor performance due to excessive idle time or due to a long job getting assigned a slow machine [14]. Preemptive migration, on the other hand, can migrate long jobs in their lives to avoid their impeding many small jobs [27].
2.4 Implementation Challenges

Software systems have to be implemented to make use of the resources in heterogeneous computing environment. However, it is a challenging job to design and implement software toolkits and framework over networks of workstations or Internet.

2.4.1 Heterogeneity

Each host may have a different type of CPU, processing power, memory space, disk storage. Thus the system facilities on different hosts vary and may be incompatible. Even in a homogeneous hardware environment there may be software heterogeneity. Software heterogeneity includes differences in the host operating system (the services and interfaces provided), the process support, interprocess communication, the compilers available (the languages and versions of languages available, as well as the quality of the code), the file systems, and database systems available. These differences must be masked by the distributed environment [6].

2.4.2 Expensive Communication

On the Internet, communication latency is large and network status may be unpredictable. Communication between hosts may take a long time. A distributed memory parallel processor runs most efficiently when the amount of communication between processors is minimized and the communication occurs in large messages rather than many small messages [58]. Because it takes time to setup and tear down the connec-
tions between processors.

2.4.3 Security

The owner of the host may not trust the foreign process running on his computer. Different organization may have different methods to ensure security. Runtime system must provide mechanism for users to manage their own security needs. As long as computers are connected to the Internet, no matter what security method is being used, it is hard to guarantee they are fool-proof security systems.

2.4.4 Dynamic Process Management

It is hard to manage thousands of interdependent processes which may also spawn their own child processes during runtime. How to manage those processes and how to migrate the processes to suitable machines is what we need to consider to contribute a good system software.

2.4.5 Compatibility

It is necessary that distributed computing systems can support legacy codes and use the current available network protocols. The system must live with the available network resources. However, it can layer better protocols over existing ones [25]. It can also layer middleware over existing operating systems.
2.4.6 Some Constraints

When implementing a load balancing algorithm, it is not practical to replace host operating systems. To layer a software system on top of the existing OS is one of the common solutions [6].

The system cannot run as "root" (or the equivalent). Indeed, quite the contrary - to protect the user themselves, most users will want it to run with the least possible privileges [25]. However, in some local networks, it may require the system to run at high privileges to solve particular problems or gain better performance.

2.5 Techniques Used to Meet These Challenges

One common approach to these problems is the use of volunteer-based systems such as Bayanihan [37] and Javelin [47] which are based on java applets that execute within the context of a web browser. In these systems, applications are decomposed into sub-tasks that can be downloaded in the form of java applets by clients who wish to volunteer computing resources. When the task completes, its result is uploaded to the server. The sub-task computation may not affect too much of the volunteer computers because they are well restricted in a sandbox. The primary drawback of these approaches is the significant restrictions placed on communications by the java security model for applets. The restriction that downloaded applets may communicate only with their server essentially limits these systems. It rules out most potentially successful network parallel applications [18].
Fundamental in all of these issues is the need for mechanisms that allow applications to obtain real-time information about system structure and state, use that information to make configuration decisions, and be notified when information changes. Required information can include system load, network activity, available network interfaces, processor characteristics, and authentication mechanisms. Decision processes can require complex combinations of these data in order to achieve efficient end-to-end configuration of complex networked systems [20].

The common framework that enables a coherent solution to these problems is object-orientation. Use of an object-oriented foundation, including the paradigm's encapsulation and inheritance properties, will make accessible a variety of the benefits often associated with the paradigm, including software reuse, fault containment, and reduction in complexity [25]. The software should be able to manage the resources of interconnected computers.

No single software system can satisfy every user. However, a good system should allow users have the greatest flexibility to configure it for their particular needs. Users should be able to decide and select both the kind and the level of functionality and make their own trade offs between these and cost [25].

### 2.6 Metacomputing Frameworks

With the rapid development of the Internet, millions of mostly idle computers are connected together [3]. Among the key attractions of the Web has been the promise
of easy access to a wealth of information around the world. Currently, however, the Web supports only a simple model of computing: information residing at a host computer can be automatically downloaded and executed on the client machine. This model, although extremely powerful, exploits only a fraction of the potential of the Internet. A more compelling vision is one that views the Internet as a source of both information and computing resources [56]. There are several tools and frameworks like Paraweb [56], Javelin [47], ATLAS [32] etc. that view the Internet as a worldwide giant distributed system. The common goal of those infrastructures is to allow users to execute serial programs remotely, on faster computers, or they may execute parallel programs on a variety of platforms across the Internet [56].

2.6.1 MPI and CALYPSO

MPI [19] (Message Passing Interface) is a widely used standard (specification) for writing message-passing programs. Such interfaces attempt to establish a practical, portable, efficient, and flexible standard for message passing.

The main advantages of establishing a message-passing standard are portability and ease-of-use. In a distributed memory communication environment in which the higher level routines and abstractions are building upon lower level message passing routines the benefits of standardization are particularly apparent.

The MPI is originally written for C and FORTRAN languages, which are suitable for large and computation oriented applications. The Java language is suitable and simple
for network programming because it provides a lot of reusable components that makes programming work simple. The execution speed may be slower. However, there are some research efforts try to use java as a tool for scientific calculation.

CALYPSO [2] is a prototype software for writing and executing parallel programs on non-dedicated platforms, based on COTS networked workstations, operating systems, and compilers.

2.6.2 PVM and JPVM

PVM [55] (Parallel Virtual Machine) is a software package that permits a heterogeneous collection of UNIX and/or Windows computers hooked together by a network to be used as a single large parallel computer. It provides a general interface that permits the components of an application to be executed over heterogeneous computing elements. It supports both Message Passing Interface and Distributed Shared Memory for inter-process communication.

PVM led to a new research direction for parallel and distributed processing. Most of the previous research efforts have concentrated either upon computational models, parallel versions of algorithms or machine architectures, while PVM gives more attention to software development environments or program construction techniques that are required to translate algorithms into operational programs.

PVM does provide some general interface and straightforward constructs for access to various resources. However, it does not have an easy to use user interface and does
not have the ability to optimize message passing dynamically [55].

JPVM [18] is implemented entirely in Java, and is thus highly portable among platforms supporting some version of the java virtual machine. This feature opens up the possibility of utilizing resources commonly excluded from network parallel computing systems such as Macintosh and Windows-NT based systems. Although network-based system such as standard PVM are generally portable among Unix variants, they are more difficult to port to other readily available environments such as PCs running Windows NT. Java essentially eliminates this issue by providing a uniform, portable interface to operating system services such as network communication.

2.6.3 Bayanihan and Javelin

Bayanihan [37] and Javelin [47] are based on java applets that execute within the context of a web browser. In these systems, applications are decomposed into sub-tasks that can be downloaded in the form of java applets by clients who wish to volunteer computing resources.

Project Bayanihan [37] seeks to develop the idea of volunteer computing, which allows people to easily pool together their computers’ processing power and cooperate in solving parallel problems. Potential use of volunteer computing-multiplying not just computation power, but communication power as well. Downloading something can take a long time, even on a fast processor, if network bandwidth is limited. It is possible to use volunteer computing to get other machines, with their own (possibly
faster) network connections, to do the downloading.

Java offers the basic infrastructure needed to integrate computers connected to the Internet into a seamless parallel computation resource: a flexible, easily-installed infrastructure for running coarse-grained parallel applications on numerous, anonymous machines.

Javelin [47] is a Java-based infrastructure for global computing. Its architecture and implementation require participants to have access only to a Java-enabled Web browser. Javelin allows machines connected to the Internet to make a portion of their idle resources available to remote clients and at other times utilize resources form other machines when more computational power is needed. The future versions will solve the remaining challenges, such as result verification, fault tolerance, and client privacy.

2.6.4 Charlotte and Globus

Charlotte [3] comprises a virtual machine model which isolates the program from the execution environment, and a runtime system realizing this virtual machine on the Web. It uses eager scheduling to provide load balancing and fault masking.

Parallel computing on local networks is generally based on mechanisms that specifically target the properties of the local area network environment. However, those mechanisms do not effectively extend to wide area networks due to issues such as heterogeneity, security, and administrative boundaries. Charlotte comprises a virtual
machine model which isolates the program from the execution environment, and a runtime system realizing this virtual machine on the Web. Load balancing and fault masking are transparently provided by the runtime system.

Current solutions to heterogeneity are generally low level and based on message-passing. High level solutions based on shared memory generally do not support heterogeneous environments. Ideally, any machine on the Internet should be able to contribute to any ongoing computation without having security concerns.

The advantage of eager scheduling used in Charlotte is that problems such as load balancing and fault tolerance are handled by the runtime system; the disadvantage is that the programmer does not have an explicit control over resource utilization. However, the system has a single point of failure, the manager.

Globus [20] is another meta-computing environment which provides low level toolkit as well as high level services for general applications to use heterogeneous computing resources.

2.6.5 E IP and Web edip

Development Environment for Distributed Image Processing (DEDIP [7]) is a generalized environment for distributed image processing. It is based on Master-Slave model that provides a user-friendly image processing operational environment.

DEDIP supports a network of multiple heterogeneous systems. The user has to simply provide the required information in his configuration file. DEDIP supports automated
file transfer among the nodes as required by the applications’ components. It also supports multiple process scheduling and robust error handling [7].

Current DEDIP system does not have load balancing feature. It would be preferable to add some scalable load balancing techniques such as eager scheduling to incorporate automatic resource optimization. Ideally, it should be extended to a web based system and can run any coarse-grained parallel applications [7].

The basic idea is to install a stand-alone Java application on volunteer computers on the Internet. Then every computer with this software may have the chance to participate in the computation. With the development of the compiler and other techniques used to improve the performance, Java as a language may be widely used in distributed computing systems.

WebDedip [29] is a generalized distributed computing environment especially suitable for implementing coarse grained applications such as image processing. It provides graphical interfaces for users to manage and schedule processes of the application. It automatically takes care of the inter-process communication and interdependency. For now, there is no resource optimization features and load balancing algorithms applied to WebDedip. That may limit this system to utilize resources over heterogenous computing environment.
2.6.6 Dome and UTOPIA

Dome [30] differs from operating systems approaches to load balancing is that the tasks have intricate intercommunication dependencies and tend to be long running. The load balancing is performed by remapping data based on the time taken by each task during the last computational phase. Thus, dome [30] load balancing does not require a complicated interrupt scheme.

UTOPIA [50] is a load sharing facility specifically built for large and heterogeneous systems. The system has no restriction on the types of tasks that can be remotely executed, involves few application changes and no operating system change, supports a high degree of transparency for remote task execution, and incurs low overhead.

2.6.7 TreadMarks and ATLAS

High-speed networks and rapidly improving microprocessor performance make networks of workstations and the whole Internet an increasingly appealing vehicle for parallel computing. By relying solely on commodity hardware and software, networks of workstations offer parallel processing at a relatively low cost.

TreadMarks [13] supports parallel computing on networks of workstations by providing the application with a shared memory abstraction. Shared memory facilitates the transition from sequential to parallel programs because most of the data structure can be retained without change. Only synchronization needs to be added.
The ATLAS [32] system is a marriage of existing technologies from Java and Cilk [8] together with some new technologies needed to extend the system into the global domain. The goal of the ATLAS system is to exploit the networked resources of the world as a giant distributed computer. It uses "work stealing" to achieve run time load balancing.

2.6.8 HARNESS and ParaWeb

HARNESS [15] is an experimental metacomputing system which will enable programs to customize their operating environment to achieve their own custom performance/-functionality tradeoffs. It has become clear that one monolithic system cannot efficiently handle all the desired communication styles. The way to success will be to design plug-in communication stacks. Runtime configurability will allow an encrypted link to be installed without user code modification. The next generation DVM (distributed virtual machine) will have to strike a better balance among performance (or the ability to optimize performance), extensibility, and interoperability.

ParaWeb [56] utilizes Internet or intra-net computing resources in a seamless fashion. It provides extensions to the java programming environment (through a parallel class library) and java run time system that allow programmers to develop new java applications with parallelism in mind.

First, Paraweb permits clients to download and execute a single java application in parallel, on a network of stations. Second, clients can automatically upload and exe-
cute programs on remote computer servers. The program is automatically uploaded and executed on a computer server and results are returned to the client. In the case of a parallel application, the client may upload code to many heterogeneous computer servers within an organization or even throughout the internet.

The motivation for the project is derived from an interest in integrating the field of high performance computing with Web technology. The introduction of Java, with its architecture-independent and Web-centric programming framework, has sparked considerable interest among the distributed and parallel programming communities.

2.6.9 Condor and SNIPE

Condor [4] is a software package for executing long running "batch" type jobs on workstations which would otherwise be idle. "Owners" of workstations have complete priority over their own machines. If a user submits a job to Condor which runs on somebody else's workstation, but the job is not finished when the workstation owner returns, the job will be checkpointed and restarted as soon as possible on another machine. Disk space must be available to store the checkpoint file both on the submitting and executing machines.

SNIPE [22] is a metacomputing system that aims to provide a reliable, secure, fault-tolerant environment for long-term distributed computing applications and data stores across the global Internet. To facilitate this the system supports: distributed data collection, distributed computation, distributed control and resource manage-
ment, distributed output and process migration. The underlying system supports multiple communication paths, media and routing methods to aid performance and robustness across both local and global networks. Since some computational resources may not be available on a continuous basis, applications may have to adapt to varying computational power. The potential for hostile attack to such systems requires that they have a high degree of security, both for authentication of data and privacy of sensitive information.

2.6.10 Legion and KnittingFactory

Legion [25] is an ambitious middleware project that will provide a solid, integrated, conceptual foundation on which to build applications. Eventually there will be Legion-like metasystem software. It is a necessary condition for a large scale digital society.

The design and implementation of KnittingFactory [44] demonstrates its benefits by applying it to Web-based computing. Java in combination with Web browser’s abilities to load and execute untrusted Java applets in a secure fashion has made computing over the Web a possibility. Some of the obstacles common to both Web-based parallel computing and collaborative work are the heterogeneity of the participating systems, difficulties in administering distributed applications, security concern of users, and often high communication delays. The applet security model, which in most parts enables browsers to execute untrusted applets in a trusted environment alleviate some of the user’s security concerns. The browser is the perfect candidate
to seamlessly bring distributed computing to every-day users.
3 Hybrid Algorithm Model

The ultimate goal of the systems described in Chapter 2 is to provide a monolithic virtual machine. The users will have the illusion of a very powerful computer on the desk and they can execute whatever complicated applications they like on the terminal. Every computer connected to the Internet should be able to participate in any ongoing applications without user's interference. That is still a long way to go because it depends upon the development of appropriate technologies.

Since load balancing algorithm's performance varies a lot under different applications and environments, a hybrid model which may be able to change the parameters of several sub-algorithms during runtime is proposed.

As mentioned earlier, simple algorithm and policies does not always mean bad performance. It may be too expensive for the scheduler to do a "perfect" job allocation, which may also increase unnecessary inter-process communication [26].

For this reason, the proposed algorithm uses non-preemptive migration. The hybrid algorithms is based on [28], which is an application centric strategy. However the current implementation of [28] has been tested only on a homogeneous environment. We propose the algorithm that aims at making use of the strengths of each sub algorithm and avoid their weaknesses during different stages of the whole application. The new algorithm and strategy should be able to work over a heterogeneous environment. In order to implement this hybrid algorithm, we have to define a set of scheduling and allocating policies.
3.1 Preliminary

First of all, we analyze the workload and the execution time in theory. Then, in the next sub-section, we will present a more practical model. We denote $T_{NLB}$ is the total execution time. So that $T_{NLB}$ will be the maximum elapsed time among all workers [36]:

$$T_{NLB} = \max_{0 \leq i < n} T_i$$

where $T_i$ is the elapsed time of worker $i$. If $W_i$ is the amount of workload assigned to worker $i$ and $P_i(t)$ is the processing capability of worker $i$ at time $t$, then

$$\int_0^{T_i} P_i(t) dt = W_i$$

Notice $W_i$ is the total work load of worker $i$ which includes background load. In practice, $P_i(t)$ is not known in advance so these equations only serve for analysis [36]. To keep in mind, we should distribute the workload as evenly as possible and adaptive to the changing conditions.

3.2 the Proposed Model

Based on the above analysis, we proposed the more practical model:

Assumptions: The users know their application's components and interdependency relationships. We can also classify the workers (nodes) into several classes based on
the capability of the machines. These information should be recorded during runtime [23].

The processes are also categorized as their priorities based on the inter-dependency graph.

A global task scheduler decides task allocation and also collect processing information from each working node. It also informs the user about the status of his job. The state information may be stored as a table with the following column in the server:

Node ID, Finished Processes(actual computing time),Running Processes(estimated computing time),Wait Queue.

As described in [28], we also made the following assumptions:

1. The user provides the processes' interdependency information (interdependency graph) and knows the nature of the processes.

2. The user provides approximate processing requirement of each process.

3. Jobs arrives according to Poisson arrival rate.

Based on GR.batch [40] Model, we changed the node processing throughput to service rate using standard process execution time. The GR.batch Model does not specifically consider the process interdependency relationships while the proposed model will take that into account. The standard process we choose is Linpack [1] bench mark. We normalize our node types and classes of sub task as follow:
Suppose, there are \( m \) different node types. Each node type \( T_i \), where \( i \) is in \( M \) and \( M = \{1, 2, \ldots, m\} \), is represented by a 3-tuple \( (T_i, \mu_i, \psi_i) \) with \( \mu_i \) and \( \psi_i \) being the service rate and the number of nodes of type \( T_i \) respectively. The processing node composition in the system can be represented as a set of 3-tuples:

\[
\{(T_1, \mu_1, \psi_1), (T_2, \mu_2, \psi_2), \ldots, (T_m, \mu_m, \psi_m)\}
\]

The total number of nodes in the system is:

\[
\Psi = \sum_{i \in M} \psi_i
\]

The node ids of type \( T_1 \) are \( (N_1, N_2, \ldots, N_{\psi_1}) \), and of type \( T_2 \) are \( (N_{\psi_1+1}, N_{\psi_1+2}, \ldots, N_{\psi_1+\psi_2}) \), and so on. \( T(N_j) \) is used to refer the node type of a processing node having id \( N_j \). Relative service rate of node type \( T_i \) with respect to node type \( T_j \) is denoted as \( \mu_i^j \),

\[
\mu_i^j = \frac{\mu_i}{\mu_j}
\]

We define

\[
\mu_i = \frac{1}{TS_i}
\]

where \( TS_i \) is the time to execute the standard process (Linpack) on node type \( i \) and \( \mu_i \) is the service rate of node type \( i \).

Job arrival rate is actually the application arrival rate. we denote \( \lambda \) is the application arrival rate. Each application follows Poisson distribution and the standard Poisson
arrival rate will be multiplied by $\lambda$. Application arrival rate is defined by user and stored in a configuration file (appinfo.txt).

Application tasks are categorized into $n$ classes. Each task class $C_i$ where $i$ is in $J$ and $J=\{1,2,\ldots,n\}$, is represented by three attributes. (1) Service Demand, denoted as $\omega_i$ is the processing requirement of a class $C_i$ task. (2) Code Length, denoted as $l_i$ is the length in Kbytes of the task transfer message generated for a class $C_i$ task if assigned remotely. The task class composition can then be represented as a set of 3-tuples:

$$\{(C_1, \omega_1, l_1), (C_2, \omega_2, l_2), \ldots, (C_n, \omega_n, l_n,\)\}

We may run an arbitrary users’ process in one job type on a standard machine and get the execution time in millisecond. We can define that time is the service demand of a certain job type. The service demand of a job is an estimated one and may not necessarily be very accurate. We should have an relatively accurate service rate at the beginning and then as the time goes on, we assume the service demand of a process will not change and the service rate of a node will be tuned dynamically because we are only interested in the node’s capability of computing the user’s processes.

We define time to execute the standard process on a standard machine is $TS$, time to execute the user process this machine is $TU$, so that

$$\omega = \frac{TU}{TS}$$

where $\omega$ is the service demand of the user process.
We also use the standard process (Linpack) as the basic unit of a sub-job. The service rate we define here are only for the processes within the target application. In case there is any background load, the service rate for our interested processes may be deducted accordingly. In this way, we can compare the service demand and service rate over different machines so that:

Time needed to complete a task is \( \frac{\omega_i}{\mu_i} \).

We assume that all the user’s jobs will arrive at the server before the central scheduler can distribute those jobs to other working nodes. So far we can define the actual work load and virtual work load as follows:

Actual work load

\[
W_j = \sum_{i=1}^{k} \frac{\omega_i}{\mu_j}
\]

Virtual work load

\[
VW_j = \sum_{i=1}^{q} \frac{\omega_i}{\mu_j} + \sum_{i=q}^{k} \frac{\omega_i}{\mu_j}
\]

Where \( k \) is the number of processes in Ready-to-run Queue and \( q \) is the number of processes in Wait Queue.

Average work load of one node

\[
AW = \frac{\sum_{i=1}^{\Psi} VW_i}{\Psi}
\]

The system will recalculate \( \mu_i \) after it finishes processing \( h \Psi \) processes, where

\[
\mu_i^k = \frac{ET_i}{AT_i} \mu_i^{k-1}
\]
$AT_i$ is the average actual processing time of node $i$ and $ET_i$ is the average estimate processing time of node $i$. $\mu_i^{k-1}$ is the previous service rate value of node $i$.

$h$ is the algorithm parameter (an integer) that determines the interval time to collect system state information.

\[
AT_j = \frac{\sum_{j=1}^{h} at_j}{h}
\]

\[
ET_j = \frac{\sum_{j=1}^{h} ct_j}{h}
\]

where $at_j$ is the actual processing time of the $j^{th}$ processes in finished queue of node $i$ and $ct_j$ is the estimated processing time of the $j^{th}$ processes in finished queue of node $i$.

### 3.3 System Overview

From figure 3, we can see that communication daemon and ftp server should be installed on every node of the system. The server has the whole picture of the application status. Central scheduler on the server side is responsible for allocating processes to workers according to current work load information.
3.4 Main Algorithm

Server Side:

1. Generate the workload and make them arrive in Poisson arrival rate.

2. Put the applications in server queue (squeue.txt).

3. Find out the processes’ interdependent information of each application and put them into server process queue (swq.txt).

4. Calculate each workers’ virtual work load.

5. Newly arrived processes (during run time) are allocated by the central scheduler to the most light-loaded (Virtual Work Load $\min_{j=1,2,\ldots,\Psi} (VW_j)$) node’s Wait Queue according to FIFO [38] policy.
6. Processes with no parent processes will be put directly in the Ready-to-run Queue. Other processes will be put in the waiting queue of the worker.

**Worker Side:**

1. When processes in Ready-to-run Queue finishes its job, they saves the intermediate result locally.

2. When all the processes in Ready-to-run Queue finished their job, the worker will put the first process of Wait Queue to Ready-to-run Queue.

3. Whenever a process at the front position of the waiting queue can not be running because of the interdependency restriction, it will notify the server and the server looks for the parent processes.

4. Worker recalculates its process capability periodically and informs the server.

### 3.5 Complexity of the Algorithm

The complexity of the hybrid algorithm is $O(n)$. $n$ is the number of applications in the server queue. Figure 4 shows the flow chart of the hybrid algorithm. There is a single loop in the program which depends upon the input size of the applications.
Figure 4: Flow Chart: Hybrid Algorithm

There are other factors which affect the complexity:

1. Disk accessing time will increase when the amount of data in the intermediate results increase.

2. The communication overhead will increase when the number of nodes in the system scales up.

3. It is also more difficult for the server to maintain the system state information and coordinate workers when the number of workers increases.

4. Unpredictable Internet traffic may conflict with the messages we use to transfer system state information. Due to the nature of Ethernet connection, the program will try to send the identical message again if it fails to transfer. The number of retries is also unpredictable.
The above factors depend upon the traffic of the network, the number of working nodes and the nature of applications. Other costs within the program remain constant. We expand the nodes into a certain number to see the speed up.
4 Implementation Issues and Case Study

Central scheduler should be aware of realtime system state and network state information in order to make the right decision. The communication daemon should have at least one port continuously listen to connection requests and create new ports to do the real data transfer in order to avoid deadlock.

4.1 Case Study

Matrix addition and multiplication are often required for image processing. So I use matrices which contain different amount of data as the sub jobs or processes of the whole application. These sub jobs can be replaced with any user provide source code or executable binary code. We assume the user knows what kind of operating system those code can be run on. We also assume the users know their problem well and have appropriately decomposed their problem into sub jobs (sub problems).

The standard process (Linpack) we use is also to solve a 500*500 linear equations.
Figure 5: Interdependency Graph of Application 1

Figure 6: Interdependency Graph of Application 2

Figure 7: Interdependency Graph of Application 3
Figure 5, figure 6 and figure 7 show the interdependent relationships of application 1, 2 and 3 respectively. Application 1 and 2 are used to test the system's reaction to interdependent processes. Application 3 is used to see the system's reaction to independent processes. An implementation of eager scheduling is used to compare the performance with the hybrid algorithm because it is a well-known strategy and it is also a client-server architecture which can deal with interdependent processes. Figure 8 shows the flow chart of the implemented eager scheduling algorithm. The main differences are that there is no work load description and no queues in the eager scheduling model. Figure 9 and figure 10 are the result when executing eager scheduling and the hybrid algorithm for interdependent and independent processes.

**Figure 8: Flow Chart: Eager Scheduling**
Figure 9: Performance for Interdependent Processes

Figure 10: Performance for Independent Processes

Figure 11: Performance When Increasing the Number of Applications
4.2 Discussion of the Result

Figure 11 shows the total execution time increases when the number of applications increases. Figure 9 shows the performance when executing interdependent processes while figure 10 shows the result when executing independent processes. From the these two figures, we can see that the hybrid algorithm performs relatively better for both independent processes and interdependent processes. Due to frequent communication between the available nodes of the system, eager scheduling does not provide good speedup when the number of working nodes scales up.
5 Conclusion and Future Work

A new hybrid load balancing algorithm is proposed to make use of the idea of several algorithms in literature. Estimated processing time is used as the metric of the workload. It automatically takes care of the process interdependency relationships and background load. Algorithm parameters are tuned during runtime in order to make a better prediction for the next round of execution. Using Linpack [1] for measuring and comparing computer's processing capability is realized by us for the first time.

Stand alone java applications are placed on top of the existing operating systems to test the load balancing algorithm. By java's platform independent nature, this work can easily be transported to different operating systems and it can be implemented over heterogeneous computing environment. The hybrid algorithm performs relatively better for both independent and interdependent applications.

The current implementation of this algorithm does not provide a graphical interface. Easy to use metacomputing systems which can transparently manage all the computing resources over the Internet is the goal of all frameworks. We may also use some bandwidth measurement tools to improve the overall performance in the future.
References


<table>
<thead>
<tr>
<th><strong>Name</strong></th>
<th>Xinyu Gan</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Year of Birth</strong></td>
<td>1974</td>
</tr>
<tr>
<td><strong>Place of Birth</strong></td>
<td>Chengdu, China</td>
</tr>
<tr>
<td><strong>Education</strong></td>
<td>M.Sc., Computer Science</td>
</tr>
<tr>
<td></td>
<td>University of Windsor</td>
</tr>
<tr>
<td></td>
<td>Windsor, Ontario</td>
</tr>
<tr>
<td></td>
<td>Canada</td>
</tr>
<tr>
<td></td>
<td>2001 - 2003</td>
</tr>
<tr>
<td></td>
<td>B.Sc., Computer Science</td>
</tr>
<tr>
<td></td>
<td>Beijing Polytechnic University</td>
</tr>
<tr>
<td></td>
<td>Beijing, China</td>
</tr>
<tr>
<td></td>
<td>1992 - 1997</td>
</tr>
</tbody>
</table>