

CALYPSO: An Environment for Reliable Distributed Parallel Processing^{*†}

Arash Baratloo[‡]
New York University

Partha Dasgupta[§]
Arizona State University

Zvi M. Kedem[¶]
New York University

October 2, 1995

Abstract

The importance of adapting networks of workstations for use as parallel processing platforms is well established. However, current solutions do not always satisfactorily address important issues that exist in real networks. External factors like the sharing of resources, unpredictable behavior of the network and machines including slowdowns and failures, are present in multiuser networks and cause poor performance. In using today's available toolkits for distributed programming, in general, the responsibility of handling these external factors is left to the programmer, a task that further complicates the development of an already difficult job of parallel programming on distributed systems.

CALYPSO is a prototype software system for writing and executing parallel programs on non-dedicated platforms, using Commercial Off-The-Shelf (COTS) networked workstations, operating systems, and compilers. Among notable properties of the system are: (1) simple programming paradigm incorporating shared memory constructs, (2) separation of the program and the execution parallelism to allow programs to scale as computers join an ongoing computation, (3) transparent utilization of unreliable shared resources by providing dynamic load balancing and fault tolerance, and (4) effective performance for large classes of coarse-grained computations.

In this paper we introduce CALYPSO, present its goals, and describe the design and the implementation of the current prototype. We also report on our initial experiments and performance results in settings that closely resemble the dynamic behavior of a “real” network. Under varying work-load conditions, resource availability and process failures, the efficiency of our test program ranged from 94% to 87% on five networked workstations, obtaining close to optimal speedups—highly competitive with systems that are less robust.

^{*}This research was partially supported by the National Science Foundation under grant numbers CCR-94-11590, and CCR-95-05519.

[†]A short preliminary version of this paper has appeared in *Proc. 4th IEEE Intl. Symp. on High Performance Distributed Computing*, August 1995.

[‡]Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 251 Mercer St., New York, NY 10012-1185, (212) 998-3350, baratloo@cs.nyu.edu.

[§]Department of Computer Science, Arizona State University, Tempe, AZ 85287-5406, (602) 965-5583, partha@cs.eas.asu.edu.

[¶]Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 251 Mercer St., New York, NY 10012-1185, (212) 998-3101, kedem@cs.nyu.edu.

1 Introduction

CALYPSO is a software system that provides a parallel processing platform on a network of standard workstations or personal computers. Several similar systems already exist. However, unlike those systems, CALYPSO uses a novel architecture to provide fault tolerance, automatic load balancing, and a shared address-space, while incurring low runtime overhead for coarse-grained computations. In addition, CALYPSO separates logical parallelism from physical parallelism. This allows programmers to write parallel programs based on the parallelism of the problem they are trying to solve, and not for a specific, and frequently unpredictable runtime environment.

The pragmatic reasons for using networks of workstations as a parallel processing platform are well established. To name a few, they are widely available, cheap, and “*ride the technology curve*” by benefiting from the continuing advances of the COTS (Commercial Off-The-Shelf) hardware, software and operating systems. Such a platform has been perceived by some as an inadequate substitute for a “real” parallel machine. Nevertheless, it is known that many applications run quite well on networks of workstations, and this has contributed to the popularity of systems such as, Linda [1, 14], MPI [24], and PVM [40, 22].

CALYPSO is an integrated system that allows the programming and execution of computationally complex applications on a set of workstations on a network. It can utilize publicly used workstations, and takes care of dynamic availabilities, slowdown and failures prevalent in such configurations. The programming and running of programs is simple and elegant, making user productivity high. In addition it is designed to be portable; the current version runs on Unix and a Windows NT port is underway.

CALYPSO 0.9 has been implemented in C++ using AT&T’s CC compiler version 3.0, and runs on Sun SPARCStations running SunOS 4.1.3. The system is simple to use and its implementation is simple and small. The entire system comprises about 3000 lines of code.

In this paper we describe the goals, the design, and the implementation of CALYPSO 0.9, and present some performance results. In the next section, we describe some unique and interesting characteristics of CALYPSO. We start by presenting previous and related results in Section 2. Then Section 4 presents the basic ideas and the system architecture in brief. Sections 5–7 cover CALYPSO programming and execution. In Section 8, after formally defining our test-bed and our performance metrics, we report the execution time of a specific parallel program using one to five identical workstations. Using these results as a yard-stick, we run *the same program*, (1) where there is a mixture of slow and fast workstations, (2) where workstations are added at different times during a computation, (3) where some workstations crash, (4) where the work-load of some workstations fluctuate in time, and (5) where some workstations stop, and after some time continue with the computation. We believe these settings resemble the dynamic behavior of a “real” world more closely—CALYPSO is targeted at this world.

All performance results presented are actual experiments and not simulations. The *complete* source code of the program is shown on page 14.

2 Previous Work

CALYPSO has its roots in results by us and by our colleagues addressing fault tolerance, parallel program execution on fault-prone asynchronous abstract machines, and distributed systems [37, 38, 16, 32, 17, 30, 20, 27, 29, 28, 4, 31, 3, 21, 19]. The research leading to CALYPSO started

as formal work which developed provable methods for executing parallel computations, initially on abstract machines with crash-failing processors, and later on abstract machines with asynchronous processors. An outline of a network of workstations-based system for parallel computing based on earlier formal work was presented in [19]. CALYPSO is an evolution of this design, and is the result of considerable redesign and extensive experimentation on progressively more and more sophisticated implementations. Major new developments in CALYPSO 0.9 include: (1) a programming strategy that allows dynamic thread segment declarations, thus providing programming scalability; (2) dynamically evaluated termination condition to increase efficiency; (3) addition of sequential steps to handle external interactions and side effects; (4) a global management mechanism which uses buffering, collating of updates and transmits updates as differences thus allowing arbitrary logical data granularity; (5) memory locality management. However, it does not implement a fault-tolerant manager, described in [19], which relied on dispersal and evasion.

We now briefly summarize other related work. A large body of experimental results exist in the attempt to make parallel programs run on distributed hardware [22, 33, 40, 5, 14, 8, 10, 6, 7]. These systems can be loosely divided into two types, those that depend on a message passing scheme and those that use some form of global address spaces.

Many systems provide message passing, or Remote Procedure Call facility built on top of a message passing. These include PVM [40, 22], Orca [7], GLU [25], Isis and Horus [13, 11], Concert/C and so on. These systems provide a runtime library (and sometimes compiler support) to enable the writing of parallel programs as concurrently executable units. These units are then spawned on different machines on the network, typically by a control site. If parts of the program need access to shared data, they obtain it either by passing the data around in messages or by reading them from files made available to all sites via Sun-NFS, or a similar protocol. These frequently require significant rewriting of programs. They are also difficult to debug. In spite of such drawbacks, PVM is one of the most popular parallel programming systems for distributed hardware. This demonstrates the acute need for providing such facilities to the end user.

Using global memory to make programs communicate has been established as a “natural” interface for parallel programming. Distributed systems do not support global memory in hardware, and hence, this feature has to be implemented in software. While systems built around Distributed Shared Memory (DSM) like IVY [34], Munin [9], Clouds [16, 20, 18], Mether-NFS [35]) provide a more natural programming model, they still suffer from the high cost of distributed synchronization and the inability to provide suitable fault tolerance.

A mature system that uses a variant of the DSM concept is Linda [14]. Instead of having a global address space, Linda employs a database type environment, the *tuple-space*. The tuple space (managed by tuple-servers) provide the functions of shared memory, data storage service, control information provider and synchronization. Worker processes in the network retrieve work tuples from the tuple-space and generate result tuples. The system is interesting, and receptive to additional enhancements. However, programs still have to be re-written to use Linda.

Pirhana [23] provides features similar to CALYPSO in that it allows dynamic load sharing via the ability to add and subtract workers on the fly. However the programming strategy is different, deleting workers need backing up tuples, and fault-tolerance is not supported.

The issues of providing fault tolerance have generally been addressed separately from the issues of parallel processing. There have been three major mechanisms: *checkpointing*, *replication*, and *process groups*. Such approaches have been implemented in CIRCUS [15], LOCUS [36] and Clouds [20], Isis [12, 39, 11], FT-PVM [33], FT-Linda [5], and PLinda [2, 26]. However, all these systems

add significant overhead, even when there is no failure.

More recently several prominent projects have similar goals to us. These include the NOWs project at Berkeley, the HPC++ project and the Dome project at CMU. All these projects however use approaches that are somewhat conventional (RPC or message based systems with provisions for fault detection, checkpointing, etc). Hence they are quite different internally from CALYPSO. The language used by CALYPSO, called CSL (details later), however has some similarities to C++ developed at Caltech.

3 Goals and Properties of CALYPSO

While large networks of workstations (or PC's) exist in almost every organization, and the numbers are growing rapidly, and these machines are largely *idle* they seem to be an attractive resource that is not used to their potential. Given that there are many compute intensive tasks that can be cost-effectively utilize such a resource, why is it that programs that can make use of networks of workstations have not proliferated?

A major reason is that the cost to *harness* this power is too high. That is, although networks of workstations are a good value in terms of raw computing power (meaning hardware), the cost to harness this power (meaning software development) still remains high and unattractive.

There are systems to utilize this hidden power, but they often do not address some important issues. For example they require extensive changes to the programming model, or they are unable to handle the separation of programming and execution parallelism, or they cannot deal with failures, or they lack adequate load distribution and balancing to account for slow and fast machines.

The challenging problems in providing a satisfactory environment for parallel processing on networks of workstations are well known. Such problems include programmability, high-performance, scalability, load balancing, and fault-masking. We have addressed most of these in the design and the implementation of CALYPSO.

Furthermore, as the commercial and the administrative realities prohibit the vast majority of users from buying special purpose hardware and running "private" operating systems, CALYPSO utilizes standard hardware and standard software. The current prototype runs under SunOS, and the system has been designed and implemented to be portable. We expect ports to run on most Unix-based operating systems as well as Windows NT.

3.1 Features

CALYPSO runs parallel programs on a set of workstations connected by a standard network and running a standard operating system. A CALYPSO computation utilizes a central *manager* process and a dynamically changing set of *worker* processes.

In our system, we provide the following high-level features:

- **Ease of Programming:** The programs are written in a language we called CALYPSO Source Language (CSL). CSL is essentially C++, with an added construct to express parallelism (i.e. `parbegin - parend`). It is based on a shared memory model and is very simple to learn and use. Important aspects contributing to ease of programming are the the elimination of data partitioning and the need to specify how and when to move data between workstations. Section 5 presents the syntax and the semantics of CSL.

- **Separation of Logical Parallelism from Physical Parallelism:** Programs are logical entities. Thus, the parallelism expressed by a programmer is independent of the parallelism provided by the execution environment, which is tied to the availability of workstations. This mapping between the program parallelism and the execution parallelism is transparent in CALYPSO.
- **Fault Tolerance:** CALYPSO executions are resilient to failures. Worker processes can fail, and possibly recover, at any point without affecting the correctness of the computation. Unlike other fault-tolerant systems, there is no significant additional cost associated with this feature—in the absence of failures, the performance of CALYPSO is comparable to a non-fault-tolerant system. The impact of fault tolerance on performance is discussed in Section 8.
- **Dynamic Load Balancing:** CALYPSO automatically distributes the work-load depending on the dynamics of the participating machines. The result is that faster workers do more work than slower workers. Not only there is no cost associated with this feature, but it actually *speeds up the computation*, as fast workers are never blocked waiting for slower workers to finish their work assignments—they bypass the slower ones.
- **High Performance:** While providing the features listed above, our initial experiments indicate that the overhead is surprisingly small for mid- to coarse-grained computations. We elaborate on our performance metrics in Section 8.) In our experiments, under varying conditions of failures, slow-downs and changing number of available machines, the total overhead varied between 4.0% and 12.7%.

Some of the mechanisms used in CALYPSO are described next.

3.2 Mechanisms

The paradigm that CALYPSO embodies in a working system, was first described in [32]. That paper details a methodology for instrumenting general parallel programs to automatically obtain their fault-tolerant counterparts that can run on a abstract, shared memory multiprocessing machine. While the solutions in [32] were formulated in the context of synchronous faults, they were later applied in [30] to a certain variant of asynchronous behavior i.e. could run on a machine whose processors take arbitrary amounts of time to execute each step. This research formed the basis of the ideas incorporated into CALYPSO

A unified set of mechanisms, *eager scheduling* and *collating differential memory*, is used to provide the functionality of CALYPSO. The *idempotence* property is fundamental in CALYPSO: a code segment can be executed multiple times (with possibly some partial executions), with exactly-once semantics.

The importance of idempotence, and the utilization of the eager scheduling to take advantage of it, was discovered in [32] in an abstract context. (The term “eager scheduling” itself was coined later.) Eager scheduling is a mechanism for assigning concurrently executable tasks to the available machines. Any machine can execute any “enabled” task, independent of whether this task is already under execution by another machine. As a consequence, free machines end up doing more work than loaded ones, leading to a balanced system. Secondly, computations *do not stall* while dealing with system’s asynchrony and faults. Thirdly, newly available machines can be transparently and productively assimilated into an executing computation. And finally, any of the machines that are “helping out” the parallel computation can fail or slow down at any time.

The mechanism of collating differential memory provides logical coherence and synchronization while avoiding false sharing. It is an adaption and refinement of the *two-phase idempotent execution strategy* [32] among others. Memory updates are collated to assure exactly-once logical execution, and they are transmitted as bitwise differences, preventing false sharing. This supports efficient implementation of idempotence in addition to other performance benefits that we shall see later. Details about these mechanisms are discussed later.

We rely on these three mechanisms to ensure that the available machines are *used where they are needed the most and without delays associated with false sharing*. A “participating” machine is used to mask faults and/or increase the parallelism depending on a transient state of the system. A critical feature of the CALYPSO system is that it is neither a fault-tolerant system extended for parallel processing nor is it parallel processing system extended for fault tolerance. *A single unified set of mechanisms provides both parallel processing and fault tolerance.*

4 System Architecture in Brief

In this section we integrate the two mechanisms described earlier, and briefly describe the system architecture. Section 6 will further elaborate and describe our implementation.

For simplicity we assume an execution of a single parallel program on a network of workstations. The workstations are individually administered, and the share of the resources they devote to our parallel program is transient and beyond our control. Our parallel computation is written as a CSL program.

4.1 Programming Model

CSL programs are written by inserting *parallel* tasks into a *sequential* program. The parallel tasks performs the computationally intensive work and the sequential code does the high-level control flow, I/O, and other important, though (presumably) not computationally intensive activities.

We refer to an execution of such a parallel task as a *parallel step*. We refer to the sequential execution fragment between two consecutive parallel steps as a *sequential step*. Thus, the execution consists of alternating steps: sequential, parallel, sequential, parallel, The execution always starts with a (possibly empty) sequential step and ends with a (possibly empty) sequential step. The execution of a parallel step consists of several concurrent *thread segments*. Figure 1 illustrates a small fragment of an evolving execution.

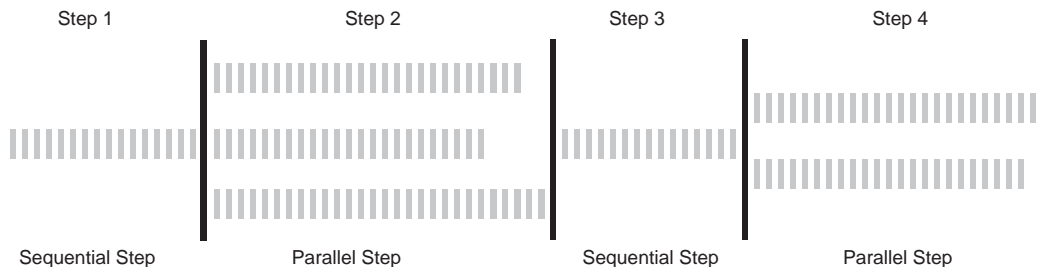


Figure 1: A fragment of an evolving execution.

Within a parallel step, the access to the global data is CR&EW (concurrent read *and* exclusive write, or multiple readers and a single writer)—a data item can be read by any number of thread segments *and* written by at most one. (It is trivial to extend the system so that it supports the “Common” variant of the CRCW model. That is, there could be several thread segments writing the same variable, but they all must write the same value.)

4.2 An Overview of the Execution

In general, the execution of a CSL program is distributed over a dynamically changing set of interconnected *host machines*. We do not need a specific network protocol for CALYPSO. For instance, it is enough that the machines can *ping* each other, and of course allow users certain privileges, such as the privilege to execute programs. There is *no need for a shared file system*, nor for the network to be directly connected. In fact, we have run a CSL program on a set of machines, some in New York and some in Phoenix, with the machines at each location connected by an Ethernet, and the two locations connected by the Internet.

A CALYPSO computation is executed by exactly one manager, and a dynamically changing set of worker processes. In the current implementation the manager must be a completely reliable process (that does not fail). The workers may come and go, speed up and slow down in an unpredictable manner, depending on the transient availability of resources and *not* on the properties of the computation. In Figure 2, we sketch the software architecture of CALYPSO 0.9.

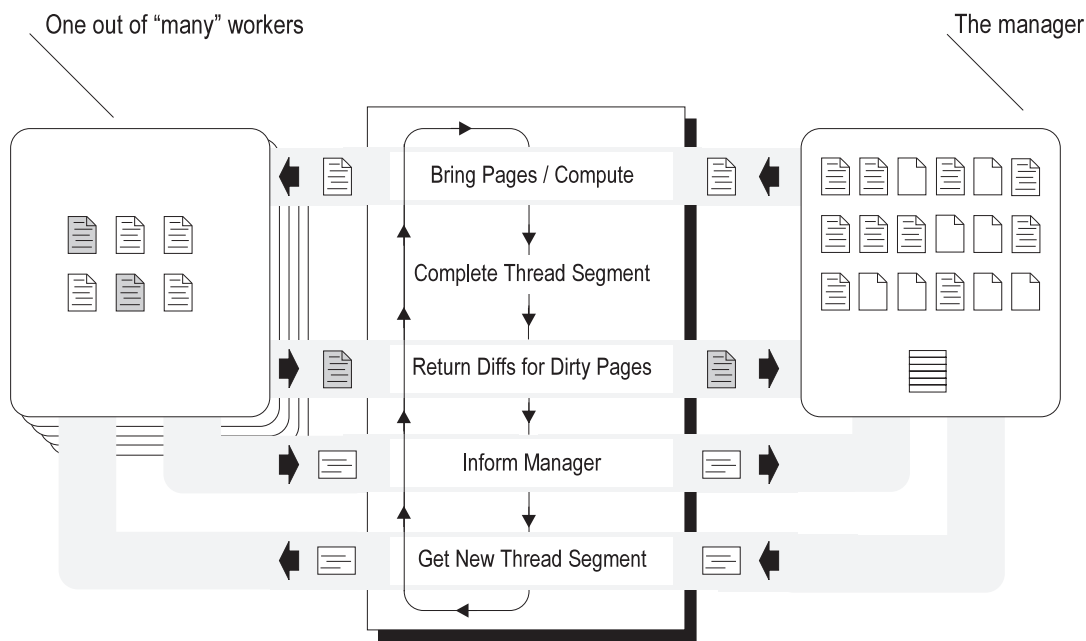


Figure 2: The software architecture of CALYPSO 0.9.

During a computation, the sequential steps are executed by the manager. Parallel steps are executed by the available workers and “managed” by the manager. Any particular parallel step has some number of concurrently executable thread segments. Each of the available workers, contacts

the manager and obtains a work *assignment*: an unfinished thread segment. It then proceeds to execute it. Once finished, the worker reports the results back to the manager and requests another assignment. The manager keeps track of thread segments (which ones have been assigned, which ones have been completed, . . . , etc.) and has the responsibility to assign more work. At first, it prefers to assign thread segments that have not yet been assigned to any worker. But what should it do when the number of available workers exceeds the number of thread segments? Or when all thread segments have been assigned but not finished, and there are idle workers eager to work? Eager scheduling implies that if there is unfinished work, and there is an idle worker, then the work should be assigned. The manager simply assigns work using eager scheduling until all thread segments are complete, at which point that parallel step is considered complete.

Although we have not discussed what it means for a worker to do useful work, it should be clear how eager scheduling leads to both load balancing and fault tolerance in a unified solution. That is, a crash-failed worker is a special case of a slow worker—it is an infinitely slow worker—and a fast worker, a good worker, will never wait for a slower one when it can overtake it and do its job. The “unnecessary” computation is generally small, and in practice it is essentially free. (Note the replicated work is assigned to workers that have “nothing else to do” beyond participating in the current parallel step.) In experimenting with a matrix multiplication program, we were *fully* charged for this extra work, and we observed that the overhead was in fact very low.

For a worker to do useful work, it must not only perform the computation that is specified by its assignment, but it must perform the operations on the manager’s data-set, and return the results back to the manager. CALYPSO provides a software implementation of Distributed Shared Memory. It views the virtual address space of each worker process as partitioned into two disjoint areas: shared and private. Shared virtual memory pages are demand-paged in a manner analogous to that of virtual memory systems. But as stated before, this alone does not ensure idempotence of updates and memory coherence in presence of eager scheduling. The additional techniques required for this are described in Section 6.

5 The Syntax and the Semantics of CALYPSO Programs

In this section we discuss the CALYPSO Source Language (CSL) supported by CALYPSO 0.9.

CSL is standard C++ augmented with 4 keywords: **shared**, **parbegin**, **parend**, and **routine**. Programs written in CSL have access to several new syntactical features and follow certain additional structural constraints, which we describe semi-formally.

Sequential Steps. The syntax is standard C++ with the restriction that the `main()` function is provided by the CSL library. The programmer provides a `main` with the prototype:

```
void calypso_main(int, char *[]);
```

Shared Variables. Globally shared variables are declared using the keyword **shared**. In the program’s main file, the one including `calypso_main`, the following declaration must appear:

```
shared { member-listopt };
```

If the shared variables are used in another program file, they must be declared as external variables:

```
extern shared { member-listopt };
```

Of course, *member-list* must be identical in all such files.

In CALYPSO0.9 all shared variables are declared statically. Dynamic memory allocation of the shared segment has been recently implemented and will be incorporated in a later version. The routines to allocate memory are semantically analogous to `malloc` and `free` functions except they work on the shared memory.

There is an annoying “feature” in the current prototype: shared variables are treated as members of an implied structure *shared* and must be qualified using `shared` keyword. (Thus if *x* is a shared variable, it is referenced as `shared.x`). This is due *only* to the limitations of our current, lex-based, syntactical preprocessor and will not be required in upcoming versions, which will employ a yacc-based preprocessor.

Parallel Steps. A parallel step is a new compound statement with the syntax:

```
parbegin routine-listopt parend;
```

routine-list consists of a sequence of `routine` statements. A typical form of such a statement is:

```
routine [ int-exp ] ( int width , int number ) routine-body
```

routine-body is a standard sequential C++ program fragment. It can access the shared data, the two parameters passed to it (*width* and *number*), and its local data. Other non-shared data that is usable to the sequential step should not be used in the routine. Also, it cannot have external interactions, such as I/O.

Within a parallel step, data item can be read by any number of thread segments *and* written by at most one. Semantically, all thread segments read the value of shared variables at the beginning of the parallel step, and write atomically at the end of a parallel step.

For illustrative purposes, we consider the following program fragment declaring a parallel step:

```
parbegin
    routine[m+1](int wid1, int id1) { SequenceOfStatements1 }
    routine[m+3](int wid2, int id2) { SequenceOfStatements2 }
parend;
```

When the fragment is entered, several thread segment “siblings” will be *spawned* by each of the two `routine` statement. Specifically, each routine spawns the number of thread segments equal to its value of *int-exp* field. Thus, if $m = 1$, the first routine spawns 2 thread segments, each executing *SequenceOfStatements1*; and the second routine spawns 4 thread segments, each executing *SequenceOfStatements2*. Each thread segment receives two parameters passed by value: *width* and *number*: the first provides it with the number of siblings and the second with its own sequence number among its siblings. Thus, the 4 siblings executing *SequenceOfStatements2* are passed parameter pairs: (4,0), (4,1), (4,2), and (4,3), respectively. Each thread segment is able to *adapt* and *distinguish* its behavior from its siblings by these two parameters.

Once all thread segments are completed, the parallel step ends.

As an example, we provide a “very friendly” CSL version of the HelloWorld program:

```
#include <iostream.h>
#include <calypso.H>

shared {
    int Array[100];
```

```

};

void calypso_main(int, char *[]) {
    int    NumberOfFriends;

    for (int i=0; i<100; i++)
        shared.Array[i] = -1;
    cout << "How many greetings would you like? ";    // At most 100
    cin >> NumberOfFriends;

    parbegin
        routine[NumberOfFriends](int NumberOfThreads, int ThreadNumber) {
            shared.Array[ThreadNumber] = ThreadNumber;
        }
    parend;

    for (i=0; i<NumberOfFriends; i++)
        cout << "\"Hello World\" from Friend " << shared.Array[i] << endl;
}

```

Thus, each thread segment is given its own serial number in the parameter `ThreadNumber`. It then puts its serial number in the corresponding location of the shared vector. The resulting output is:

```

"Hello World" from Friend 0
"Hello World" from Friend 1
. . .

```

6 From a Program to its Execution

Once a CSL program, say `program.csl` is written, it is preprocessed, compiled, linked, and executed. We now describe the steps involved.

6.1 Preprocessing

A preprocessor takes a CSL program, say `program.csl`, and translates it into a standard C++ program called `program.C`. Basically, parallel steps are *stripped away* and replaced by function calls implementing the thread segments. As a very schematic example consider the pseudo-CSL program fragment:

```

    parbegin
        routine[m](int wid, int id) { SequenceOfStatements }
    parend;

```

The following is a simplified description of the preprocessor actions. The preprocessor replaces the `parbegin-parend` construct with function calls to our library, which will at run-time:

1. Create m instances of a function executing the given *SequenceOfStatements* (with the values of *wid* and *id* passed appropriately).
2. Assign executions of these functions (as thread segments) to the available workers.
3. Manage the execution.

The actual function definition specifying the *SequenceOfStatements* is kept separately (as an assignable unit of execution) under a specific dummy name, say `foo123`. Furthermore, the preprocessor packages all the shared variables on pages on separate virtual pages, referred to as *shared pages*.

6.2 Compilation and Linking

Compilation of `program.C` using standard C++ compiler produces `program.o`, which is then linked with our library to produce `a.out`. We stress that `a.out` contains the code for both the manager and the worker processes. Thus the memory layout of the manager and worker processes are identical. The `a.out` file can be executed as a manager or as a worker by using command line options.

6.3 Execution

As described above, a CSL program is executed on a single manager and a dynamically varying set of workers.

When the computation starts, the manager executes the sequential code until it reaches the first parallel step, step 2. It then suspends its execution and gets ready to “manage” the execution. Again for simplicity assume this is the program fragment

```
parbegin
    routine[m](int wid, int id) { SequenceOfStatements }
parend;
```

Say that the address of `foo123` (see Section 6.1) is `0X0AAA` and the variable m is evaluated to 3. The manager then prepares a table called the *progress table* and initializes it as follows:

Step	Address	Width	Identification	Started	Finished
2	0X0AAA	3	0	0	NO
2	0X0AAA	3	1	0	NO
2	0X0AAA	3	2	0	NO

The last row of the table, for instance, indicates that this is step number 2; that a thread segment defined by the function at the address `0X0AAA` needs to be computed; that there are 3 such thread segments defined by this function; that this row describes the third out of 3 sibling thread segments (numbered 0, 1, 2); that no workers have started working on this thread segment; and that its computation has not yet finished.

Assume that the first thread segment has been assigned to two workers and the second and third thread segments have been assigned to one worker each. Furthermore, the workers assigned to the first and the second thread segments are still working but the third segment has already finished. This is reflected by the following table:

Step	Address	Width	Identification	Started	Finished
2	0X0AAA	3	0	2	NO
2	0X0AAA	3	1	1	NO
2	0X0AAA	3	2	1	YES

The system utilizes a simple version of *eager scheduling* as follows. The manager listens to workers requesting assignments. It assigns to each free worker a thread segment that has not been finished—among all such thread segments it assigns one that has been assigned the least number of times.

We now turn to the description of a worker. The worker knows all the shared pages—pages on which all and only shared variables are located. The worker access-protects the shared pages (using the system call `mprotect()`), and then contacts the manager for work, who sends it an assignment specified by the 3 parameters `address`, `width`, and `identification`. The worker now executes this assignment: it performs a function call to the address `address` with the parameters `width` and `identifier`. During this execution, the first time a worker accesses a protected shared variable a `SIGSEGV` signal is raised. The signal handler fetches the appropriate page from the manager, installs it in the worker's process space, and unprotects the page for future use. Then the computation proceeds. When the task terminates, the worker identifies all of its dirty memory and sends the differences (XORS) between the original page and the updated page to the manager. Then it protects its shared pages again and contacts the manager for another assignment.

The manager accepts the first completed execution for each thread segment and discards subsequent ones. Late updates from workers are easily recognized and ignored (e.g. an update for thread segment assigned in step 2 but arriving in step 4).

The manager buffers the updates until the end of a parallel step, at which time all updates are performed. Different parts of a page can be updated by different workers, as long as the CR&EW condition is met. Note that this condition is an aspect of the logical program design, and is independent of the various workers' assignments. In a parallel step, values read by a worker are those existing at the beginning of the step, and the updated values are readable only at the beginning of the next step.

The shared memory is always logically coherent as far as the program is concerned. Yet there is no need for expensive mechanisms such as *distributed locking* or *page shuttling*. This is due to the exploitation of the synchronization primitive which is a side effect of the language construct, as well as utilizing differences for transmitting updates. Also, the collating technique (buffering updates, accepting the first update, discarding the other) in fact implements *two-phase idempotent execution strategy*. As a consequence, correctness is assured in spite of the multiplicity of executions.

Memory that has been paged-in by a worker, is kept valid as long as possible. For instance, the manager knows in which step a page has been modified last. So, for instance, if some page was modified last in step 4, it was read by some worker in step 6, and that worker is working on a thread in step 8, then the worker does not fetch the page but accesses its cached copy. This is a low cost (almost free) strategy that gives us a significant performance benefit. Note that, read-only shared pages are fetched by a worker at most once and write-only shared pages are never fetched. Modified shared pages are re-fetched only when necessary. Invalidation requests are piggybacked on the work assignment messages and bear very little additional cost. The programmer *does not* declare the type of coherence or caching technique to use, rather, the system dynamically adapts.

It is clear from the above description that the workers (and their location) and thread segments are

not related in any fundamental way. In fact, the syntax of CSL *does not even allow* the programmer to specify the workers or how to distribute the data. However, we have implemented a library call in our experimental setting which allows us to start workers during an execution based on the available resources.

7 Programmability

Programmability of Calypso, using CSL has been one of the strong points in its acceptance to the user community. Calypso has been used to run a variety of application programs, benchmarks and course projects in Graduate courses, both at ASU and NYU. Programmers pick up the CSL syntax and semantics, very quickly and can start writing non-trivial programs within a day. The feedback we have received has been very positive.

The programmability is attributable to the following:

- The programmer does not need to learn a new language, or learn a large variety of library functions. Just the use of 4 intuitive keywords.
- The semantics of the CSL constructs are simple to comprehend.
- The parallel program is very similar to the sequential program. In fact in most cases the programmers have written a sequential program and then added a few `parbegin` - `parend` constructs around highly computational code, with good results.
- The programmer does not have to conceptualize low level details such as networking and message passing.
- There is no perceived distribution from the programmers point of view – there is no need to partition data, assign work to workers, synchronization and other woes of distributed programming. Not even the overhead of setting up RPC routines.
- Finally, the programmer does not need to know the real number of machines available, the programmed parallelism is converted to physical parallelism by Calypso, at runtime and load balancing and failure tolerance is handled automatically.

To date, the applications ported to Calypso, includes one financial application (computing bond indexes), automatic target recognition, ray tracing and eigenvalue computations.

8 Performance Experiments

We have carefully measured the performance of CALYPSO on a number of applications. Here we present the results for a specific standard problem, matrix multiplication. We wrote a single CSL program, (whose source code is shown later) executed it on sets of workstations under different patterns of slowdowns, failures and recovery.

Results are shown for the following cases, all for the same CSL program:

1. The program running on identically behaving, otherwise unused workstations. We wanted to measure speedups when there are no failures, no slow-downs, and no need for load balancing or fault tolerance (although the execution, of course “does not know this.”)

2. The program running on various combinations of fast and slow machines.
3. The program running on a set of machines, some of which crash-fail during the computation
4. The program running on a set of machines, and additional machines become available during the computation.
5. The program running on a set of machines, and some machines either die, or become available or become slow at various points in the computation.

All experiments were conducted using the following scenario:

- The software used was CALYPSO 0.9. A program (shown below) was preprocessed, compiled, and linked, and the identical `a.out` executable was used for *all* tests.
- The machines used were five identical Sun SPARCStation SLC workstations in a public laboratory, connected by 10 Megabit Ethernet. We tried to make sure the machines and the network were quiescent by running tests during the night. But we had no real control over this. So the test results may understate the potential performance.
- All five machines used are *disk-less* machines, running off one server (hence use network swapping). We expect that the performance will be higher with better-equipped machines.
- All experiments used the matrix multiplication program listed below. It computes the products $A \times B = C$ and $A \times B = D$ in two parallel steps. The two steps were used to also test the cases where worker is “out of step.” The matrices are of size 500 by 500 and filled by pseudo-random floating point numbers. Each parallel step is specified by a single routine that spawns 50 thread segments. This is the *complete* CSL code for `mm.csl`:

```
#include <calypso.H>

const int  SIZE = 500;
const int  NUM  = 50;
shared {
    float  A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE], D[SIZE][SIZE];
};

void randomFill(float m[SIZE][SIZE], int rows, int cols) {
    for (int i=0; i<rows; i++)
        for (int j=0; j<cols; j++)
            m[i][j] = rand();
}

// -----Start of Calypso Main-----
void calypso_main(int , char *[]) {           // Start of Execution
    randomFill(shared.A, SIZE, SIZE);         // Sequential Step
    randomFill(shared.B, SIZE, SIZE);         // (for initialization)
```

```

parbegin                                                    // 1st Parallel Step
  routine[NUM] (int numThreads, int tid) { // Start 50 Thread Segments
    int from = tid * (SIZE/numThreads);
    int to = from + (SIZE/numThreads);
    for (int i=from; i<to; i++) { // Outer Loop
      for (int j=0; j<SIZE; j++) { // Middle Loop
        shared.C[i][j] = 0;
        for (int k=0; k<SIZE; k++) // Inner Loop
          shared.C[i][j] += shared.A[i][k] * shared.B[k][j];
      }
    }
  }
parent;                                                    // End of 1st Parallel Step

parbegin                                                    // 2nd Parallel Step
  routine[NUM] (int numThreads, int tid) {
    int from = tid * (SIZE/numThreads);
    int to = from + (SIZE/numThreads);
    for (int i=from; i<to; i++) {
      for (int j=0; j<SIZE; j++) {
        shared.D[i][j] = 0;
        for (int k=0; k<SIZE; k++)
          shared.D[i][j] += shared.A[i][k] * shared.B[k][j];
      }
    }
  }
parent;                                                    // End of 2nd Parallel Step
}
// -----End of Calypso Main-----

```

To do the experiment, we define 10 machine *profiles*, Machine A, Machine B, ..., Machine J. Each machine profile a-priori determines its behavior. Here are the descriptions of each machine profile, see also Figure 3.

- *Machine A* is available to us 100% for the duration of the computation. It is a regular machine, that does not fail or slow down during the execution. We call it the *fast machine*.
- *Machine B* is available to us 50% for the duration of the computation. That is, it executes for us at half its regular speed. This is achieved by running a background process (called `hog`), at high priority, that runs for a second and sleeps for a second. We call this the *slow machine*.
- *Machines C, D, and E* are respectively taken away after 100, 200, and 300 seconds from the start of the computation, respectively. This is done by killing the worker manually at the corresponding time.
- *Machines F, G, and H* are given to us after 300, 200, and 100 seconds from the start of the computation, respectively. This is done by starting a worker manually at the corresponding time.

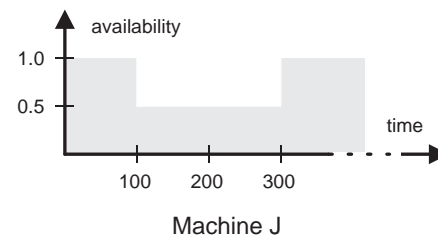
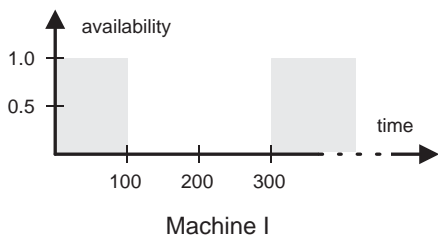
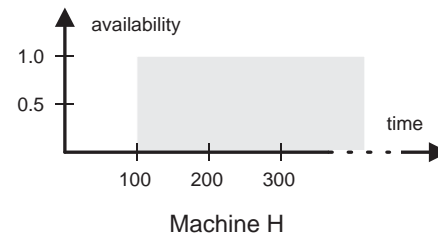
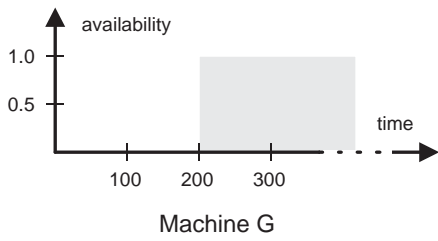
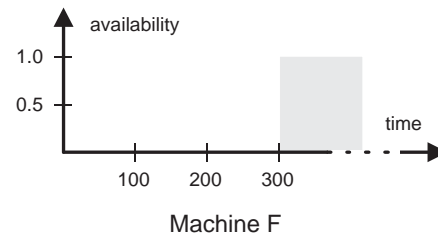
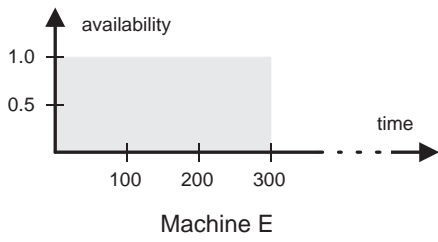
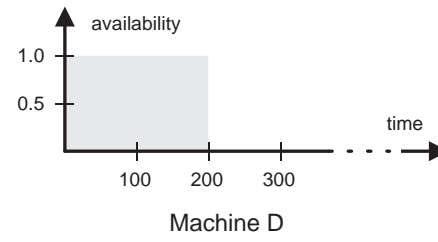
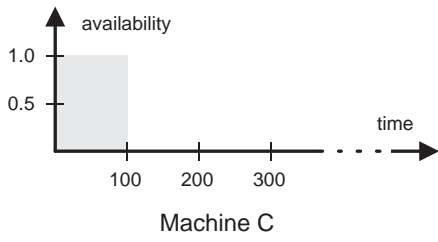
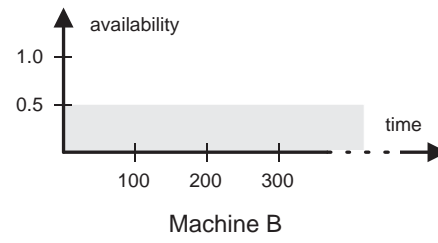
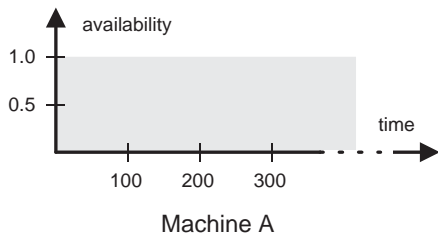


Figure 3: Profiles of machines available to the CALYPSO computation.

- *Machine I* is available to us 100% for the first 100 seconds, then 0% for the next 200 seconds, and then 100% for the rest of the computation. This models a transient network or machine failure.
- *Machine J* is available to us 100% for the first 100 seconds, then 50% for the next 200 seconds, and then 100% for the rest of the computation. This models a transient machine availability.

All times reported are “wall clock” or elapsed times, *not* CPU or virtual times. As the pseudo-random filling of the input matrices, A and B is an artifice, the times were measured from the start of the first parallel step (computing $A \times B = C$), until the end of the second parallel step (computing $A \times B = D$).

In each experiment, we use up to 5 machines, from the 10 profiles. Whenever a machine is “available” to us, we are charged for its use, regardless of whether we are in fact able to benefit from its work or not.

In addition, the calculations are done in a manner (described below) so that all overhead (networking, file access, swapping, updating memory, etc.) are included in our charges, regardless of whether we have control over it or not. Hence, our results are based on quite conservative assumptions.

Following our model, an execution of the program consists of five steps:

1. A sequential step. This step initializes the input. At the end of this step, all the shared data resides in the manager’s space only—the workers do not have any of the input data.
2. A parallel step. This step executes the first matrix multiplication. During the execution, data is moved to the workers as necessary and the results are returned by the workers to the managers, who combines them to obtain the value of the product.
3. A sequential step. It is empty.
4. A parallel step. This step executes the second matrix multiplication. The manager and the workers function as in step 2.
5. A sequential step. It is empty.

In each of our experiments that will be at least one perfect machine, of profile A. Presumably, we have one machine that we completely control. The manager will run on that machine. A single worker will run on each participating machine, including the machine on which the manager is running.

As we cannot improve the performance of sequential tasks, we will only examine the performance of the parallel tasks. In our case, this simply means that we examine the performance from the beginning of step 2 (the first parallel step) through the end of the computation. For simplicity therefore, we start our “wall clock” at the beginning of step 2. We stress again that at time $t = 0$, the workers have no shared data and at the end of the computation, $t = T$, the manager has received and processed all the outputs from the workers. Thus, we will account for all the costs associated with the execution of the parallel steps.

8.1 The Cost Model and the Performance Metrics

We now describe our cost model. Each machine profile P , is defined by the function availability_P . Availability is a function of time, and depicts the fraction of the CPU resources available to us from

that machine. Thus, the availability of 1 denotes the machine is free and completely available, and the availability of 0 denotes that the machine is unavailable. Then, if the computation lasted for time T , the *work* that the machine gave us is $\int_{t=0}^T \text{availability}_P dt$. This is the area of the shaded region for the time interval $[0, T]$ in the graphs of Figure 3.

In general we will have several machines in the computation, say n machines with profiles, P_1, \dots, P_n , respectively. If the computation lasted for time T , then the *total work* was:

$$W = \sum_{i=1}^n \int_{t=0}^T \text{availability}_{P_i} dt$$

The work W is the “charge” we incur for having the machines available to us during the computation. Since machine availability is an external function, we are charged whenever a machine is available, whether we use it effectively or not. Also given the charging method, it is obvious that the overhead includes the network time, the time wasted by redoing computations, the time taken to move data between workers and the manager, the time spent by the operating system and other system activities.

For the user the interesting metric is generally the *speedup* with respect to the conventional sequential execution. The achieved speedup must be compared with the highest possible theoretical speedup given some set of machines.

Given W , we can compute the number of equivalent perfect machines available to us. We trivially obtain:

$$\text{Number of Equivalent Perfect Machines} = \frac{W}{T}$$

This is the upper bound on *any* obtainable speedup. We now turn to computing the speedup an execution in fact achieves.

We wrote a standard sequential C++ program for the same matrix multiplications, using the trivial cubic time method. This program was tested on machines with various profiles. On a machine with profile A, (a fast machine) it executes in 1610 seconds. We call this value $T_{\text{sequential}}$. On a machine with profile B, (a slow machine,) which is 50% available, it executes in 3212 seconds.

Using the work metric defined above, in the base sequential case the program runs on one 100% available machine, hence $W_{\text{sequential}} = T_{\text{sequential}} = 1610$. (More precisely, $W_{\text{sequential}} = 1610$ machine-seconds, but we will measure work in seconds also.)

If a CALYPSO execution takes total time T , then for that execution the speedup is given by:

$$\text{Speedup} = \frac{T_{\text{sequential}}}{T}.$$

Of course, as stated above:

$$\frac{T_{\text{sequential}}}{T} \leq \frac{W}{T}.$$

The closer the speedup is to the number of equivalent perfect machines, the better is the performance of the system. To normalize this, we use another measure, that of *efficiency*. It is defined by:

$$\text{Efficiency} = \frac{W_{\text{sequential}}}{W}$$

and ranges between 0 and 100%. Efficiency of 100% means that the execution achieved optimal speedup with respect to the best achievable. It measures how well we use the resources that “happened” to be provided to us.

8.2 Results of Performance Experiments

We conducted 5 families of experiments and we describe them in turn. The results are graphed showing the total time, the achieved speedup, and the number of equivalent perfect machines (which upper-bounds the speedup).

For each experiment we utilize up to 5 machines of various profiles. On one machines we ran the manager and a single worker, on other machines we ran a single worker.

We label each experiment with a label such as 3A+2D. Here, 3A+2D indicates that a particular execution was charged for the work of 3 machines with profile A and 2 machines with profile D. Given an execution time, the rest is easily computable.

1. See Figure 4. Here we examine how the computation performs using from 1 to 5 machines of profile A, which devote all their resources to the computation. The label “Seq” indicates the sequential C++ program execution.

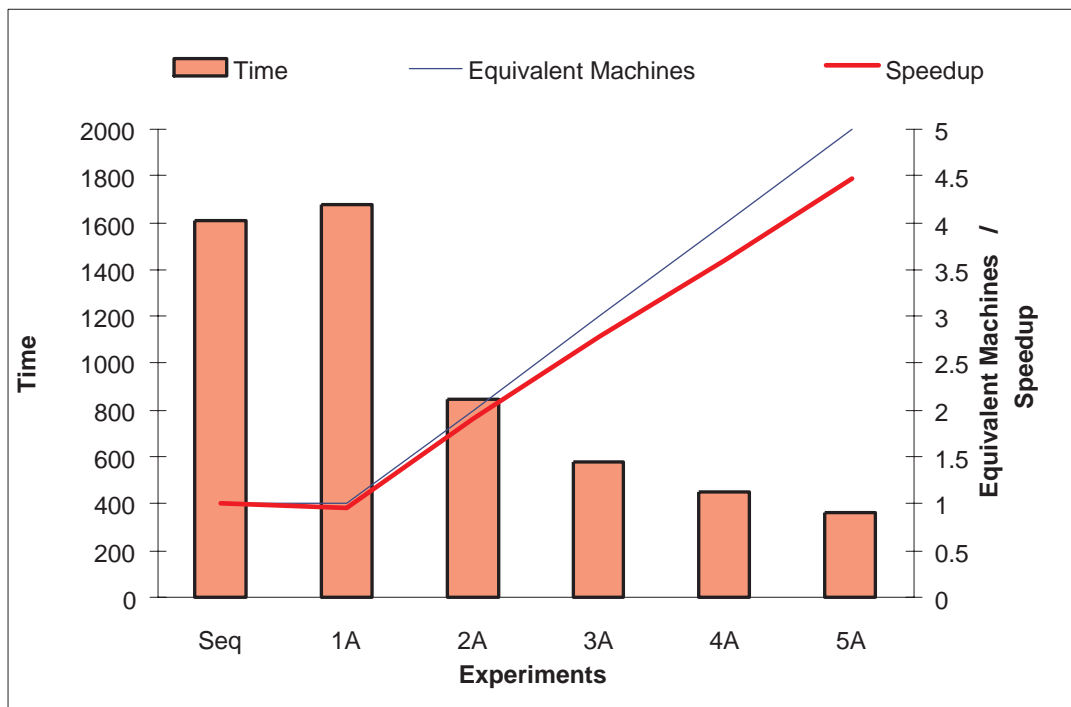


Figure 4: Family of experiments number 1.

The execution time for 1, 2, 3, 4, and 5 machines were 1677, 845, 578, 448, and 360 seconds respectively. This produces speedups of 0.96, 1.91, 2.76, 3.59, and 4.47, and efficiencies of 96%, 95%, 92%, 90%, and 89% respectively.

(In case 1A, the manager and the worker were on one machine and the 4% degradation is due to CALYPSO operations, message passing between these two processes, as well as the operating system overhead.)

This family of experiments show that when there are no failures or slow-downs, CALYPSO bears little overhead, even though it is “prepared” to handle such adverse cases.

- See Figure 5. Here we examine how the computation performs using 5 machines, with different number of machines with profile A (fast) and profile B (slow). The label 5B denotes 5 slow machines were used, the label 2A+3B denotes 2 fast and 3 slow machines were used.

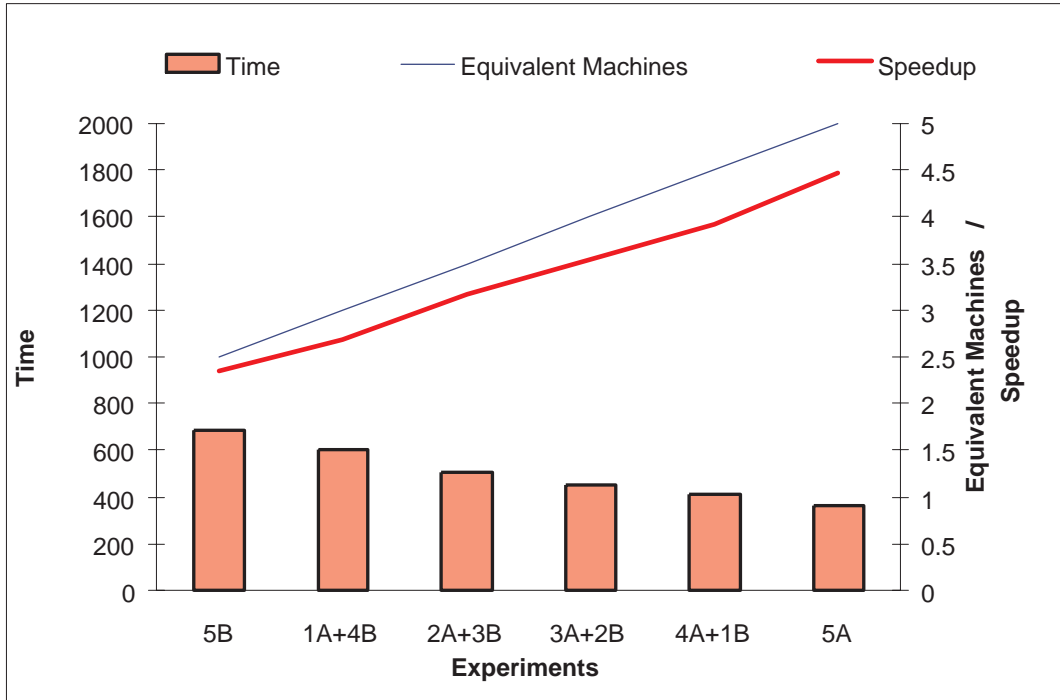


Figure 5: Family of experiments number 2.

The speedups ranged from 2.34 for 5B (in this case the best possible speedup would be $5 \cdot 0.5 = 2.5$), to 3.92 for 4A+1B. Note that the speedup in case 4A+1B (3.92) is better than in case 4A (3.59) and poorer than in case 5A (4.47). The efficiency ranged from the high of 94% to the low of 87%. This family of experiments show that utilization of workstations with heterogeneous speeds is high.

One case is worth discussing. By comparing the results of this and the previous experiments, we obtained the speedups of 3.59, 3.93, and 4.47; for the cases 4A, 4A+1B, and 5A, respectively. So we note that one slow machine helped 4 fast machines.

- See Figure 6. Here we examine how the computation performs in the presence of failures. Every computation here started with 5 machines, and 2 were crashed after 100 (3A+2C), 200 (3A+2D), and 300 (3A+2E) seconds. In our graph we also include the case of the 5 machines with profile A (fast machines) for comparison.

In this experiment, we achieved speedups of 3.06 for 3A+2C, 3.51 for 3A+2D, and 4.11 for 3A+2E. Again, note the effective use of the C, D, and E machines: as the speedup for 3A+2E

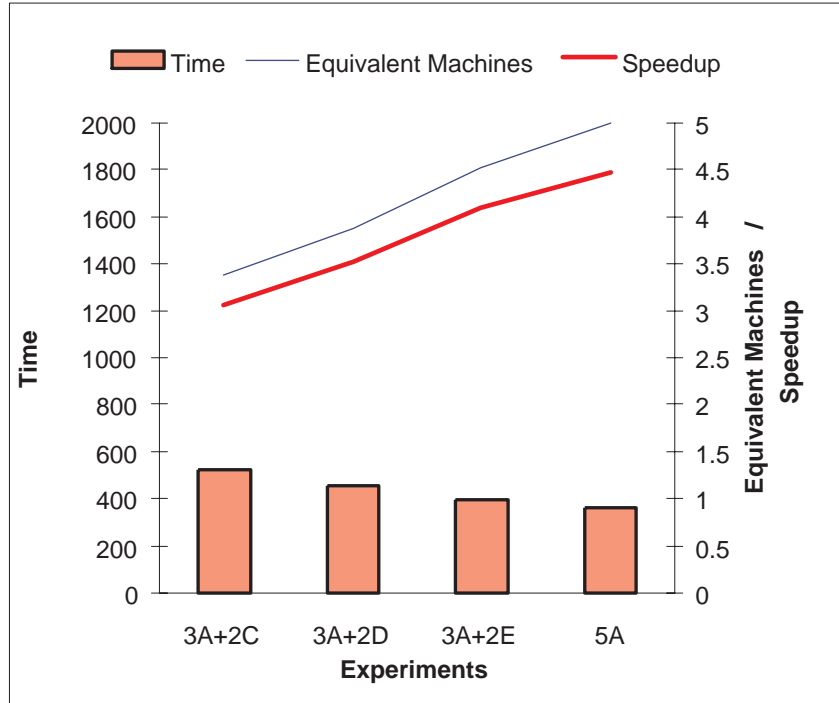


Figure 6: Family of experiments number 3.

is better than 3A+2D, which is better than 3A+2E, and all three are better than just 3A (2.76). Thus the speedup increased monotonically with additional machine availability. The overhead remains quite low, as shown by the efficiency, which ranges from the high of 91% to the low of 90%.

- See Figure 7. Here we examine how the computation performs using 3 machines (3A) initially, with 2 additional machines joining the computation at times 100 (2C), 200 (2D), and 300 (2E) seconds. Again, we include the case of 5 machines machines with profile A for comparison.

The speedups are: 3.38 for 3A+2F, 3.68 for 3A+2G, and 4.15 for 3A+2H. Here, the efficiency ranged from the high of 92% to the low of 90%. This experiments show that the system is able to effectively utilize machines that dynamically (and unpredictably) join a computation in progress.

- See Figure 8. Here we examine how the computation performs when 2 out of 5 machines (a) slow down to 0% and then recovered (3A+2J), and (b) slow down to 50% and then recovered (3A+2I).

Here the efficiency ranges from the high of 89% to the low of 88%. These experiments demonstrate the ability of our system to utilize resources as they are available, even if availability changes over time. The system does not “give up” on machines that can still help the computation, even if they are slow, as they will not slow down the computation.

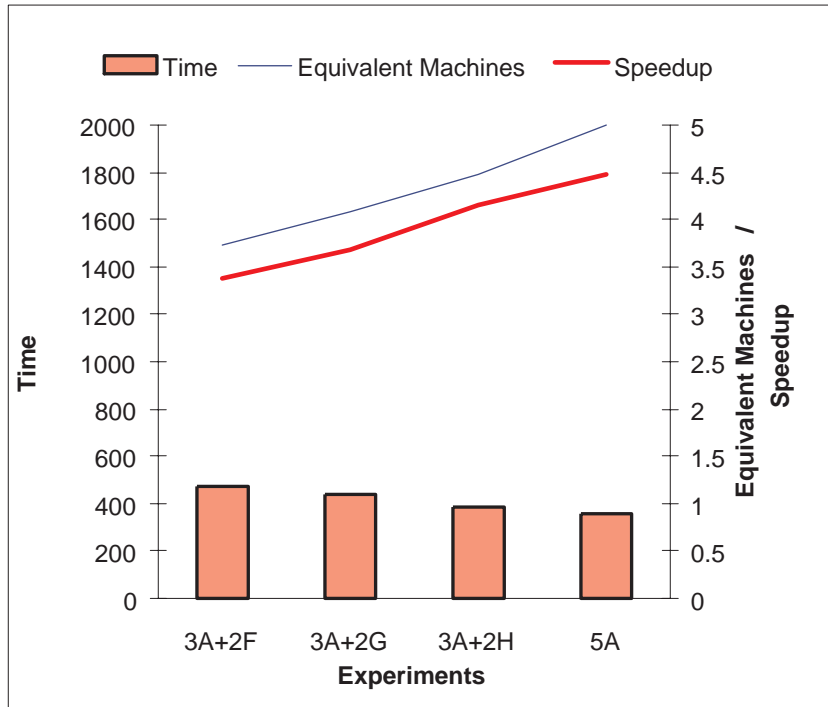


Figure 7: Family of experiments number 4.

9 Conclusions

CALYPSO is an ongoing research project. It has already gone through several phases of experimentation and enhancements in design, implementation, and performance. We are working on further enhancements and extensions to the system. We summarize and list some of the important properties of our system:

- The measured overhead is low. This is of special significance, as CALYPSO is able to execute programs in situations where other systems are inefficient or simply fail to execute. These are not isolated situations either. We are targeting network of workstations with fluctuating processor loads, network traffic, and even crash-failed processes; if anything, this is a conservative view of the real world.
- The system smoothly and efficiently adapts to fluctuating availabilities of workers.
- The two mechanisms we use, eager scheduling and differential collating memory, work well together.
- The programming interface is simple and easy to learn. The ease of programming is enhanced by the separation of logical and physical parallelism, shared memory model, transparent data movement, and the simple style of expressing parallelism.

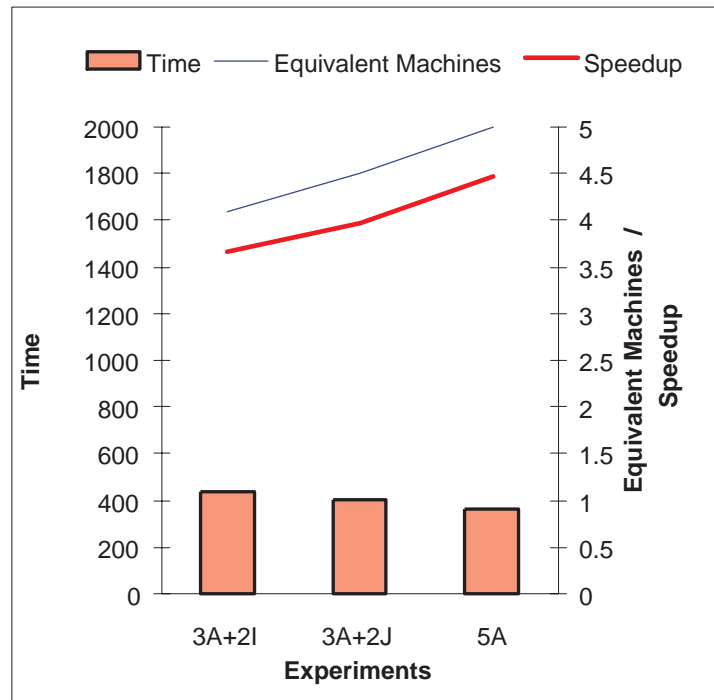


Figure 8: Family of experiments number 5.

10 Acknowledgments

The authors acknowledge the following for their contributions to the CALYPSO project: Churngwei Chu, Mehmet Karaul, Dimitri Krakovsky, Fabian Monrose, and David Stark.

References

- [1] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, 1986.
- [2] B. Anderson and D. Shasha. Persistent Linda: Linda + transactions + query processing. In J.P. Banatre and D. Le Metayer, editors, *Research Directions in High-Level Parallel Programming Languages*, number 57 in LNCS, pages 93–109. Springer, 1991.
- [3] Y. Aumann, Z. Kedem, K. Palem, and M. Rabin. Highly efficient asynchronous execution of large-grained parallel programs. In *34th IEEE Ann. Symp. on Foundations of Computer Science*, pages 271–280, 1993.
- [4] Y. Aumann and M. Rabin. Clock construction in fully asynchronous parallel systems and PRAM simulation. In *33rd IEEE Ann. Symp. on Foundations of Computer Science*, pages 147–156, 1992.

- [5] D. Bakken and R. Schlichting. Supporting fault-tolerant parallel programming in Linda. Technical Report TR93-18, The University of Arizona, 1993.
- [6] H. Bal and A. Tanenbaum. Distributed programming with shared data. In *Proceedings of ICCL*, pages 82–91, Miami, FL, October 1988. IEEE, Computer Society Press.
- [7] H. E. Bal and A. S. Tanenbaum. Orca: A language for distributed object-based programming. *SIGPLAN Notices*, 25(5):17–24, may 1990.
- [8] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, P. Le Dot, H. Nguyen Van, E. Paire, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville, and Vandôme. Architecture and implementation of Guide, an object-oriented distributed system. *Computing Systems*, 4(1):31–67, 1991.
- [9] J. Bennett, J. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proc. 2nd Annual Symp. on Principles and Practice of Parallel Programming*, Seattle, WA (USA), 1990. ACM SIGPLAN.
- [10] B. Bershad, E. Lazowska, and H. Levy. PRESTO: A system for object-oriented parallel programming. *Software—Practice and Experience*, 18(8):713–732, 1988.
- [11] K. Birman. The process group approach to reliable distributed computing. Technical report, Cornell University, 1993.
- [12] K. Birman, T. Joseph, T. Raeuchle, and A. Abbadì. Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering*, SE-11(6):502–508, 1985.
- [13] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, 1991.
- [14] N. Carriero and D. Gelernter. Linda in context. *Communication of ACM*, 32(4):444–458, 1989.
- [15] E. Cooper. Replicated distributed programs. *Operating Systems Review*, 19(5):63–78, December 1985.
- [16] P. Dasgupta, R. Ananthanarayanan, S. Menon, A. Mohindra, and R. Chen. Distributed programming with objects and threads in the Clouds system. *Computing Systems*, 4(3):243–276, 1991.
- [17] P. Dasgupta and R. C. Chen. Memory semantics for large grained persistent objects. In G. Shaw A. Dearle and S. Zdonick, editors, *Implementation of Persistent Object Systems*. Morgan Kaufman, 1990.
- [18] P. Dasgupta, R. C. Chen, S. Menon, M. P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. J. LeBlanc, W. F. Appelbe, J. M. Bernabéu-Aubán, P. W. Hutto, M. Y. A. Khalidi, and C. J. Wilkenloh. The design and implementation of the Clouds distributed operating system. *Computing Systems*, 3, 1990.
- [19] P. Dasgupta, Z. M. Kedem, and M. O. Rabin. Parallel processing on networks of workstations: A fault-tolerant, high performance approach. In *Proceedings of the 15th Intl. Conf on Distributed Computing Systems*, to appear, June 1995.

- [20] P. Dasgupta, R. J. LeBlanc, M. Ahamad, and U. Ramachandran. The Clouds distributed operating system. *IEEE Computer*, 24, 1991.
- [21] M. Fu and P. Dasgupta. Programming support for memory mapped persistent objects. In *COMPSAC*, 1993.
- [22] G.A. Geist and V.S. Sunderam. Network-based concurrent computing on the PVM system. *Concurrency: Practice and experience*, 4(4):293–311, 1992.
- [23] David Gelernter, Marc Jourdenais, and David Kaminsky. Piranha scheduling: Strategies and their implementation. Technical report, Department of Computer Science, Yale University, 1993.
- [24] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing-Interface*. MIT Press, 1994.
- [25] R. Jagannathan and E. A. Ashcroft. Fault tolerance in parallel implementations of functional languages. In *The Twenty First International Symposium on Fault-Tolerant Computing*, 1991.
- [26] K. Jeong and D. Shasha. Plinda 2.0: A transactional/checkpointing approach to fault tolerant linda. In *Proceedings of the 13th Symposium on Reliable Distributed Systems*, 1994.
- [27] Z. Kedem. Methods for handling faults and asynchrony in parallel computations. In *1992 DARPA Software Technology Conference*, pages 189–193, May 1992.
- [28] Z. Kedem and K. Palem. Transformations for the automatic derivation of resilient parallel programs. In *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 15–25, 1992.
- [29] Z. Kedem, K. Palem, M. Rabin, and A. Raghunathan. Efficient program transformations for resilient parallel computation via randomization. In *24th ACM Symp. on Theory of Computing*, pages 306–317, May 1992.
- [30] Z. Kedem, K. Palem, A. Raghunathan, and P. Spirakis. Combining tentative and definite algorithms for very fast dependable parallel computing. In *23rd ACM Symp. on Theory of Computing*, pages 381–390, May 1991.
- [31] Z. Kedem, K. Palem, A. Raghunathan, and P. Spirakis. Resilient parallel computing on unreliable parallel machines. In A. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation*. Cambridge University Press, 1993.
- [32] Z. Kedem, K. Palem, and P. Spirakis. Efficient robust parallel computations. In *22nd ACM Symp. on Theory of Computing*, pages 138–148, May 1990.
- [33] J. Leon, A. Fisher, and P. Steenkiste. Fail-safe PVM: A portable package for distributed programming with transparent recovery. Technical Report CMU-CS-93-124, CMU, 1993.
- [34] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.
- [35] R. Minnich and D. Farber. The Mether system: Distributed shared memory for SunOS 4.0. In *USENIX-Summer*, pages 51–60, Baltimore, Maryland (USA), 1989.

- [36] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. Locus: A network transparent, high reliability distributed system. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 169–177. ACM, 1981.
- [37] M. Rabin. Fingerprinting by random polynomials. Technical report, Harvard University, 1981.
- [38] M. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *J. ACM*, 36:335–348, 1989.
- [39] A. Ricciardi and K. Birman. Using process groups to implement failure detection in asynchronous environments. In ACM, editor, *Proceeding of Distributed Computing*, pages 341–353. ACM, 1991.
- [40] V.S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, 1990.