

**Calypso NT:
Reliable, Efficient Parallel Processing
on Windows NT Networks¹**

Donald Mclaughlin, Shantanu Sardesai and Partha Dasgupta
Department of Computer Science and Engineering
Arizona State University
Tempe AZ 85287-5406

Contact Address:

Partha Dasgupta
Department of Computer Science and Engineering
Arizona State University
Tempe AZ 85287-5406
ph: (602) 965-5583
fax: (602) 965-2751
email: partha@asu.edu
http: [/cactus/eas.asu.edu/partha](http://cactus/eas.asu.edu/partha)
[/www.eas.asu.edu/~calypso](http://www.eas.asu.edu/~calypso)

¹ This research is partially supported by grants from DARPA/Rome Labs, Intel Corporation and NSF.

Abstract

Calypso NT is a parallel processing system that runs on Windows NT workstations. The system allows a parallel program written in C or C++ to automatically execute on a networked cluster of workstations. Management of parallel execution as well as support for load balancing, fault tolerance and shared memory are provided by the runtime system.

A Calypso program begins execution as a sequential program. At the beginning of the parallel portion of a computation, it becomes a “manager” process and allows “workers” to join the computation dynamically. As more workers join, the speed up of the computation increases. If workers crash, depart or become slow, they are automatically excluded (dynamically). The program terminates gracefully as long as there is at least one active worker. The source program, however, does not reflect any of these intricacies; they are all taken care of at runtime, by the Calypso runtime library. The Calypso NT programming interface consists of a few (two) simple API calls that expose the facilities for parallelism and shared memory. The runtime system is small and efficient and provides good performance on a local area network. The software, documentation and example programs can be obtained from: <http://www.eas.asu.edu/~calypso>.

This paper provides a detailed description of the Calypso NT API, the runtime system, the user interface and its implementation. We also provide test results, which validate our claims concerning its performance.

1. Introduction

Calypso is a runtime software system (or middleware) that provides a parallel-processing environment on a cluster of workstations. In addition to providing efficient support for parallel programs using shared memory, Calypso provides *transparent load balancing and fault-tolerance at no additional cost*. Shared memory forms the basis of the programming model used by Calypso. Also, the Calypso programming model disassociates physical parallelism (determined at runtime) from logical parallelism (as expressed by the program). There are many parallel processing systems for networked workstations, but Calypso is unique in several respects, as discussed later in this paper.

The first prototype of Calypso was implemented in 1994 on a SunOS platform [BDK+94, BDK95]. Since then, we have also implemented the software to run on the Windows NT platform and made significant changes to the programming interface and the internals. The programming interface uses a (new) language independent API and the internals use an NT compatible architecture. These changes make programming simpler, provides an implementation that is tuned for Windows NT and also produces some positive impact on runtime performance.

Calypso NT² has been implemented in C++, using Microsoft Visual C++ 4.2 and was developed on Pentium-based machines running Windows NT 4.0 (it also runs on Windows NT 3.5 and Windows-95 with TCP/IP installed). A graphical user interface enables the user to visually control and monitor the execution of a Calypso program on local as well as remote machines. This implementation of the interface uses Microsoft Visual C++ and Microsoft Foundation Classes (MFC).

This paper presents a detailed description of the Calypso NT system. Section 2 provides a narrative of the basic mechanisms, goals and properties of Calypso. Section 3 is a tutorial on programming with the Calypso NT API. Section 4 explains the implementation of Calypso NT, including a discussion of the port from UNIX to Windows NT. We then present (in section 5) the Calypso NT user interface and show how it works. Section 6 provides detailed performance results and section 7 is devoted to related work.

2. Goals and Features of Calypso NT

In recent years, the focus of parallel processing technology has shifted from specialized multiprocessing hardware to distributed computing. The most important factor in the favor of distributed computing is that it can provide high performance at low cost. The computational power of a workstation cluster can surpass that of a supercomputer, if harnessed³ properly. The advantages of using a network of workstations to implement a parallel processing system are evident from the development of a plethora of parallel processing solutions based on distributed platforms, in recent years.

These “distributed” parallel processing systems enable the programmer to exploit the hidden computational power of the networked workstations, but they do not always address many of the important issues. Most of these systems use their own programming models and/or programming languages that are not always easy to understand and require extensive modifications to existing software. Message passing systems, for instance, add a layer of complexity that facilitates data transfer and process synchronization using messages. Other parallel processing systems do not differentiate between the parallelism inherent in an application and the parallelism available at execution time. In

² Calypso NT 1.0 has been released on the WWW and is available for free from <http://www.eas.asu.edu/~calypso>.

³ Effective and innovative harnessing of the computing power of workstation networks is the primary research goal of the Calypso project.

such cases, the degree of parallelism is an argument to the program. However once the execution begins, the width becomes fixed. Therefore, issues such as failure recovery or appropriate distribution of the workload to account for slow and fast machines cannot be addressed elegantly. The design of Calypso addresses most of these problems in a simple, clean and efficient manner. In particular the Calypso NT has the following salient features:

- **Ease of Programming:** The programmer writes programs in C or C++ and uses a language independent API (application programming interface) to express parallelism. The API is based on a shared-memory programming model which is small, elegant, simple and easy to learn. It provides programmers with single uniform view of memory and hides the details of data partitioning and data distribution.
- **Separation of Logical Parallelism from Physical Parallelism:** The parallelism expressed in an application, written using a high-level programming language, is logical parallelism. Logical parallelism should be kept separate from physical parallelism, which depends upon the number of workstations available at runtime - a number that is often unpredictable and transient. Unlike many other systems, Calypso allows an application to express logical parallelism, which makes programming much simpler. A transparent mapping between logical parallelism and physical parallelism is then provided at runtime.
- **Fault Tolerance:** The execution of parallel Calypso jobs is *resilient to failures*. All but one of the machines participating in a computation can fail and possibly recover at any time without affecting the correctness of that computation. Unlike other fault-tolerant systems, there is *no additional cost*⁴ associated with this feature in the absence of failures.
- **Dynamic Load Balancing:** Calypso automatically distributes the work-load among the available machines such that faster machines do more work compared to slower machines. This actually speeds up the computation as faster machines do not wait for slower machines until they complete the assigned work.
- **High Performance:** Our performance results indicate that the features listed above can be provided with minimal overhead and that a large class of coarse-grained computations can benefit from our system.

The core functionality of Calypso is provided by a unified set of mechanisms, called *eager scheduling*, collating *differential memory* and *Two-phase Idempotent Execution Strategy (TIES)*. Eager scheduling provides the ability to dynamically exploit the computational power of a varying set of networked machines, that includes machines that are slow, loaded or have dynamically changing loading properties. The eager scheduling algorithm works by assigning tasks to free machines in a round robin-fashion until all the tasks are completed. The same task may be assigned to more than one machine (if all tasks have been assigned and some have not yet terminated). Consequently, free or faster machines end up doing more work than the machines that are slower or loaded heavily. This results in an automatically load balanced system. Secondly, if a machine fails (which can also be regarded as an infinitely slow machine), it does not affect the computation at all. Thirdly, computations do not wait or stall as a result of system's asynchrony or failures. Finally, an executing program can utilize any newly available machines at any time.

As it is obvious the memory updates in such a system need careful consideration, the remaining mechanisms ensure correct executions in spite of failures, and other problems related to asynchrony. To ensure that the inherent possibility of a multiplicity of executions due to eager scheduling results in exactly-once execution semantically, the TIES method is used. Further arbitrarily small update

⁴ This claim is well-justified by our performance results. See section 6.

granularities in shared memory and the proper updates of memory are both supported by the collating differential memory mechanism.

3. The Programming Interface

3.1 Programming Calypso NT

The Calypso programming model is simple, easy to use and is a lot like sequential programming. The basic method of writing a parallel program is to take a sequential program and parallelize some of the compute intensive parts (for example by parallelizing the for-loops). Thus, parallel programs are structured by inserting *parallel* tasks into *sequential* programs. The execution of a parallel task is a *parallel step* while the execution of a sequential fragment between two parallel steps is a *sequential step*. The execution of a parallel step consists of the execution of several threads as illustrated in Figure 1.

An application begins as a sequential program, executing the first sequential step. When a parallel step is reached a set of threads is started. These threads share the same pool of shared memory. After all thread segments complete execution, the parallel step finishes and the next sequential step begins. Unlike CC++ [CK92] and P4 [BBD+87], there is no *distributed memory* and thus, the programmer does not need to deal with two completely different memory models in a single program.

A process called the *manager* executes sequential steps, while processes called workers execute the thread segments of a parallel step. The application semantics determine the number of thread segments in a parallel. However, the number of workers used to execute these thread segments depends upon the runtime environment. The manager at runtime does the assignment of thread segments to workers.

The Calypso NT programming interface is different from previous versions of Calypso [BDK95]. The programmer is not required to learn a new language and uses Microsoft Visual C++ to write Calypso NT programs. A language independent API is provided to express parallelism in the application and allocate the shared memory.

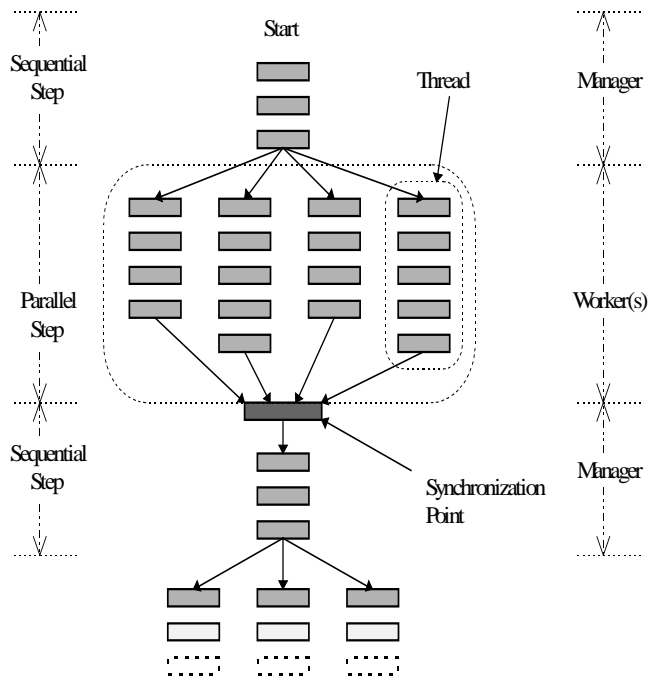


Figure 1: Execution of a Calypso Program Fragment

3.2 Overview of the API

The address space of Calypso programs is partitioned into two disjoint areas designated: *private* and *shared*. The private memory is private to each thread. All C/C++ global variables, as well as local variables fall into the private class. *Note that these private variables are often not very useful in writing parallel programs.*

The shared memory space is a special segment that is visible to all the parts of the distributed application, and must be declared and initialized at the beginning of the program. For example, to declare three arrays in shared memory, the following program fragment is used:

```
typedef struct {int a[50], b[5], c[50]; } shared;
shared *sp;
```

The above lines simply define a pointer to a structure containing the arrays. Now the pointer is initialized by the first call to the Calypso NT API, at the beginning of the program:

```
CalypsoMain(argv, argc)
{
    InitCalypso(sizeof(shared), &sp);
```

Note that the main program is called `CalypsoMain` and *not* `main`. The call to `InitCalypso` initializes the shared memory segment and initializes the shared pointer `sp`. The pointer, `sp`, is also initialized appropriately (and automatically) on all machines that participate in the parallel computation.

The code following the `InitCalypso` call is the first sequential step. Parallel steps are executed by making a call to the `ParallelExec` function as follows:

```
ParallelExec(func1, n1, func2, n2, . . . ., NULL);
```

This function, which takes a variable number of parameters, executes the specified list of functions in parallel. The parameters appear in pairs consisting of a name of a function (e.g. `func1`) followed by an integer specifying the number of parallel instances of the function that will be executed (e.g. `n1`). The programmer can specify an arbitrary number of functions to be executed in parallel. Passing a `NULL` function pointer specifies the end of the parameter list.

The function `ParallelExec` allows the programmer to express *task* as well as *data parallelism* in a application. The programmer can specify different functions to be executed in parallel within a parallel step and thereby exploiting available task parallelism inherent in an application. The data parallelism of an application can be exploited by specifying multiple parallel executions of the same function within a parallel step. The following is an example of task parallelism (i.e. five different functions are executed in parallel, and each function is executed exactly once).

```
ParallelExec(f1, 1, f2, 1, f3, 1, f4, 1, f5, 1, NULL);
```

The following is an example of data parallelism. Here 20 instances of the function `parfunc` are executed in parallel.

```
ParallelExec(parfunc, 20, NULL);
```

The execution of `ParallelExec`, causes the functions names as arguments to start execution. A separate thread that has private and shared memory segments runs each of these functions. Each thread is passes two parameters as it starts running the assigned function. Thus, the prototype of a function that can be started via a `ParallelExec` must be as follows:

```
void parfunc(int instances, int id);
```

The parameter `instances` tells the function how many copies of the function were started while the parameter `id` gives each instance of the function a unique id-number (these values vary from 0 to `instances-1`). This allows each executing instance of the function to adapt and distinguish its behavior from that of the other instances. Consequently, parallel executions of the same function work on different parts of the shared data and exploit the data parallelism inherent in the application.

The API also provides an alternate method of expressing parallelism that can alter its behavior dynamically. The programmer can use the following API function to specify the functions to be executed in parallel and their instances at run time:

```
void ParallelExecList(CalypsoJob[]);
```

This function takes an array specifying the functions to be executed in parallel. Each element in the array is of the type:

```
typedef struct{
    FunctionPtr    function;
```

```

        int                numberOfJobs;
    }CalypsoJob;

```

Therefore, each array element specifies both a pointer to the function to be executed and the number of instances of that function that will be executed during the parallel step. A NULL function pointer indicates the end of list. The programmer can initialize elements of the array at run-time and pass the array to the above API function.

To summarize, the following steps are required to develop a Calypso NT program:

- 1] A sequential program structure or an existing sequential program is re-structured into a sequence of *sequential* and *parallel* steps by identifying the parallel and sequential tasks.
- 2] The parallel tasks are implemented as functions thereby creating a modular program structure.
- 3] The shared memory requirement is estimated and is allocated as the very first step in the program by making call to `InitCalypso()`.
- 4] Whenever there is need for a parallel computation, a call to `ParallelExec()` or `ParallelExecList()` is made.

As it can be seen, the above programming methodology not only produces modular programs but also promotes reusability of code. It enables programmers to parallelize existing sequential programs with just a few minor modifications to exploit logical parallelism inherent to the application.

3.3 An Example - Matrix Multiply

Following is the complete *source-code* for an example matrix multiplication program in C++, which uses the Calypso NT API. This program is complete and has compiled and run under Calypso NT 1.0. The program initializes two matrices (500×500), with pseudo-random numbers, multiplies them and stores the result in a third matrix. The actual multiplication is done in parallel using a user specified number of threads.

```

#include <calypso.H>
#include <stdlib.h>
#include <iostream.h>
const int N = 500;
typedef struct {
    float    A[N][N], B[N][N], C[N][N];
} Shared;
Shared *sp; //the shared pointer
void RandomFill(float mat[N][N], int size) {
    for (int i=0; i<size; i++)
        for (int j=0; j<size; j++)
            mat[i][j] = rand();
} /* RandomFill */
void MatMult(int instances, in id){
    int from = id * (N/instances);
    int to = from + (N/instances);

    for(int i=from; i<to; i++)
        for(int j=0; j<N; j++){
            sp->C[i][j] = 0;
            for (int k=0; k<N; k++)
                sp->C[i][j] += sp->A[i][k] * sp->B[k][j];
        }
}

```

This is the parallel matrix multiply function. It is run as a set of parallel threads.

```

        } /* for */
    } /* MatMult */
void CalypsoMain(int argc, char *argv[???]){
    int numofThreads;
    InitCalypso(sizeof(shared), &sp);
    cout << "Input number of threads to be used: ";
    cin >> numofThreads;
    RandomFill(sp->A, N);
    RandomFill(sp->B, N);
    ParallelExec(MatMult, numofThreads, NULL);
    //run the threads to multiply the matrices
} /* CalypsoMain */
// main() : there is NO main()

```

The three arrays (matrices) are declared as a part of the `Shared` type and pointer is declared which point to the beginning of the shared structure. The execution of the program begins in `CalypsoMain`. The program begins with a call to the API function `CalypsoInit`, which declares the size of the shared memory segment and initializes the shared memory pointer, `sp`, appropriately. The two arrays (A and B) are initialized with pseudo-random numbers, using the function `RandomFill`, during the sequential step.

Then comes the parallel step. The multiplication of the matrices is performed in parallel by the `ParallelExec` API call which uses the `MatMult` function and a user-specified number of threads (`numofThreads`). Each instance of the `MatMult` function then operates on a different part of the matrices based on its `id`. The result is a set of partial results computed in parallel. Since the data is stored in shared memory there is no need to distribute the data, or to coalesce the results. Also, note that there is no need to page align the data, the distributed data access granularity is a byte.

As stated previously, *the number of threads specified by the user need not be the same as the number of machines available at run time*. Calypso will utilize the available set of machines at the runtime and will perform load balancing dynamically, in response to execution speed and behavior of these machines. As implied before, the application does not concern itself with distribution, scheduling, load balancing or fault handling.

4. The Implementation of Calypso NT

This section describes the implementation of the first version of Calypso for Windows NT machines namely Calypso NT 1.0. This version has its roots in a UNIX implementation completed in 1994 [BDK+94, BDK95]. In this section, we present the details of the mechanism used in Calypso NT and the implementation techniques, including differences in the UNIX and Windows NT versions.

4.1 The core mechanisms

The core mechanisms used by Calypso, *eager scheduling*, *differential collating memory*, *TIES*, *manager/worker coordination* and *object code architecture* are some of the basic mechanisms that make Calypso operate. Many of the techniques used in these components are operating system independent and hence are quite similar for both the UNIX and Windows NT versions. However, converting from UNIX to Windows NT was far more involved than simply recompiling source code. In this section, we first briefly describe the inner workings of Calypso. Then we discuss the implementation details of Calypso NT.

Calypso uses a manager/worker execution strategy and idempotent memory handling to ensure proper execution. Conceptually, a manager process executes all sequential steps of an application while

workers execute the thread segments of the parallel step. When a parallel step is under execution, the manager process stalls and waits for the worker processes to ask for work. When a request of this sort is made, the manager assigns it a task that is selected using the following criteria (in decreasing order of priority):

- 1] The task has not yet been assigned to any worker.
- 2] The task has been assigned to one or more workers, but has not yet been completed and has been assigned the least number of workers.

These two criteria define eager scheduling strategy in its simplest form. It ensures that all tasks finish and that they finish in almost the least amount of time, i.e. a slow worker does not hold up processing.

While a worker is running, it acquires the shared memory it uses, via demand paging, as discussed below. During a parallel step the manager also acts as the shared memory server. When a worker finishes a task, it returns memory updates to the manager i.e. each updated page is XOR-ed with the original copy and the differences are sent to the manager. If the manager has already received these updates (from a faster worker) it discards them. Otherwise, the manager “applies” the updates by XOR-ing the updates with the original copy of the page. Step numbers, worker-ids and such information are used to decide on updating memory.

4.2 How it really works

A Calypso NT program is first compiled (it is regular C/C++) and linked with the Calypso NT 1.0 library generating one executable file. This executable has embedded in it all the software needed for a fault-tolerant parallel execution. The executable is then run “*as managers*” (via command line options) on one machine and more copies of the executable are run on worker processes “*as workers*”. A GUI automates the starting of workers as described in section 5.

As stated earlier, execution of a Calypso NT application starts as manager. After initialization, the manager starts the execution of the “actual” application with a call to `CalypsoMain`. At the beginning of its execution, `CalypsoMain` is required to call `CalypsoInit`. The call specifies the size of shared memory and initializes the shared memory pointer specified by the programmer. The application(`CalypsoMain`) resumes the execution after return from the call. This constitutes the execution of the “sequential step”. At some point, the execution of a parallel step is initiated by a call to `ParallelExec` (or `ParallelExecList`). At this point, the manager builds a “*progress table*”. The progress table contains the information needed to run the workers, and schedule the workers in order to complete the execution of the parallel step. For example, Figure 2 shows a progress table that is build while executing the `MatMult` function if the `numOfThreads` is set to 4.

Each row in the progress table represents a single thread segment of the parallel step. For instance, the last row indicates that the parallel step number is 1 and the thread segment will execute `MatMult` function (this column contains the actual address of the function). The third column contains the number of siblings this thread has (4) and the fourth column has the id-number for this instance (3). The next column says that the thread has not been assigned to any workers and the last column decrees that the thread has not completed execution.

Step	Function	NumOfInstances	Instance Number	Started	Finished
1	MatMult	4	0	0	NO
1	MatMult	4	1	0	NO
1	MatMult	4	2	0	NO
1	MatMult	4	3	0	NO

Figure 2: Progress Table for parallel step in matrix multiply program

After creating the progress table, the manager listens for workers requesting work. When a worker contacts the manager, it assigns to the worker a thread segment that has been started least number of times and has not finished. It sends to the worker the information of the selected row of the progress table. The manager also gets informed when a worker finishes its work assignment, and the manager updates the “Finished” column of the appropriate row upon notification. Finally, after a sequence of sequential and parallel steps, the execution of the application (`CalypsoMain`) terminates and the manager informs all the worker of the termination. In addition to the scheduling effort the manager services all memory requests and applies memory updates as workers terminate tasks.

As mentioned earlier, worker processes are started on other workstations, using the same executable. The main program of the worker initializes the worker and then calls `CalypsoMain`. The `CalypsoMain` is expected to call `InitCalypso`. The `InitCalypso` routine in the worker first protects (makes inaccessible, via a `VirtualProtect` call) the shared memory area. The execution of `InitCalypso` routine in the worker *never returns!* It instead calls another routine, which is an infinite loop that contacts the manager for work.

The worker receives from the manager the address of the function to execute and the parameters. It then calls that function with the appropriate parameters. As this function executes, it accesses the shared pages and generates exceptions (or page faults). The exception handler gets the pages from the manager, installs them, and un-protects them, as they are required. Once the execution is complete, it sends all the updated pages to the manager in the form of *difference* (XOR) between the original and updated pages. The worker protects all the shared memory pages before it starts the execution of next thread segment and repeats the same sequence of events until the execution of the application is complete.

The above description of memory handling ignores caching of memory by workers. Workers actually handle memory quite intelligently, detecting between read and write accesses, caching pages that have not changed as well as caching updated pages that other workers have not changed. The details of the caching strategy are outside the scope of the paper.

The manager accepts only the first completed execution of each thread segment and discards the others. It accumulates all the updates and applies them at the end of the parallel step, when all thread segments are finished. Any two threads can update different parts of the same page as long as *multiple-read and single write* or CR&EW condition is satisfied. The value read by workers is those at the beginning of the parallel step and not the latest values. This ensures correctness in spite of multiple executions of same thread segment. The mechanism also provides an efficient implementation by avoiding *page-shuttling* completely. Moreover, there is no need of complicated mechanisms such as distributed locking.

4.3 Porting from UNIX to Windows NT

Calypso was first implemented under SunOS. Since many of the mechanisms used by Calypso are operating system independent, it was thought that porting would be a matter of replacing some UNIX system calls with Windows NT system calls and recompiling. After a few months of attempts, using several GNU tools and libraries, it turned out that it was a wrong approach. There are several key differences between NT and UNIX that made us change much of the implementation:

- 1] Windows NT does not support signals. There are a variety of mechanisms in NT for asynchronous events including threads, messages, events and so on, but they do not map on cleanly to signals.
- 2] Windows NT uses “Structured Exception Handling” (or SEH) which is quite different from what UNIX programmers are used to.
- 3] The preferred compiler on Windows NT (Visual C++) is an integrated suite and was not amenable to supporting “pre-processing” which is a technique the UNIX-based Calypso system uses to provide to provide parallelism.

So we decide to change the Calypso programmers interface and quite a bit of the internals. The following paragraphs outline some of the changes.

The programming interface was changed from a simple language called CSL (Calypso Source Language) to a language independent API. The API has been presented in Section 3. This makes interfacing with the Visual C++ compiler and all its debug and other facilities quite simple.

Memory handling in Windows NT is different and in many ways superior to UNIX. NT has various states of memory allocation (reserved, allocated, committed, guarded and so on) and uses a set of API functions, notably `VirtualAlloc` and `VirtualProtect` to handle memory. Calypso NT was reprogrammed to use these features. However, notifications of page faults, are quite different. While UNIX uses signals, Windows NT uses *Structured Exception Handling (SEH)* [Rich95]. SEH is a general mechanism, used to inform threads of the occurrence of asynchronous events or exceptions. Win32 SEH is implemented at programming language level rather than at system level and is implementation (compiler) dependent.

SEH is not same as C++ exception handling⁵, which makes use of C++ keywords `throw`, `try` and `catch`. In C++ exception handling, `throw` is used to raise an exception by passing an *exception object* to `catch` block, which processes the *thrown* exception. We found the C++ exception handling to be unusable for our purpose as it does not allow to restart the instruction which raised the exception, after the exception has been handled. However, it does allow the execution to continue from the next C++ statement following the statement that raised the exception. The facility is designed to provide exception handling at programming language level. Whereas, SEH although implemented at programming language level provides operating system level exception handling.

SEH uses *try-except* construct, which allows programmer to specify a guarded scope to catch a hardware or software exception using the `_try` block. The `_except` block specifies the *exception handler* that may be executed based on the value returned by the *exception filter* at the time when an exception is raised. The mechanism is quite different from that of UNIX signals as the lexical structure of the program rather than one-time installation of the handler, in the beginning of the program specify its activation. Once, the flow of control is out of the `_try` block, any raised exceptions can not be intercepted and processed by the application. While porting Calypso to NT, this restriction forced us to

⁵ Microsoft added C++ exception handling in VisualC++ version 2.0 and above. The implementation of C++ exception handling is implemented by taking advantage of the SEH capabilities already present in Windows operating system and the compiler[Rich95].

make some structural changes in the implementation so as to perform appropriate exception handling. The lexical nature of SEH dictates that a Calypso programmer implements `CalypsoMain()` function in place the traditional `main()` function from which all C/C++ applications start their execution. Calypso NT 1.0 calls `CalypsoMain()` from within the scope of a *try-except* block so as to intercept and process all the exceptions raised during the execution of a Calypso application.

Calypso's communication module is based TCP-IP and the UNIX implementation uses *Berkeley sockets* [Stev90] for passing messages between the manager and the workers. It provides higher-level utilities to mask-off the details of network programming. The communication module of Calypso NT 1.0 uses *Microsoft Windows Sockets* that provide a similar interface and functionality as that of Berkeley sockets. The Calypso communication module was compatible with the Microsoft Windows Socket interface with the only exception being the asynchronous mode of communication. The asynchronous mode (FASYNC), on UNIX, enables the SIGIO signal to be sent when I/O is possible. Windows socket implementation has tied the asynchronous mode with the event-driven windows programming. When a socket is in asynchronous mode and I/O is possible, instead of raising an exception, a message is sent to the window specified in the `WSAAsyncSelect()` function call. In effect, the mechanism is synchronous as the thread processing messages has to read the message synchronously inside an *event-loop*. Moreover, a *console application* can not use sockets in this mode. Thus, Calypso NT 1.0 does not support asynchronous manager/worker notifications directly, but it uses a daemon process connected via the user interface to provide such notifications. This will change in the next version, which supports asynchronous notifications via listening threads.

Another feature missing from Windows NT is the ability to do remote shells (`rlogin/rsh`). This is a very useful feature for spawning remote worker processes. There are several reasons why such a feature is not available. Main reasons include the manner in which the Windows GUI is structured and the lack of any ASCII support for applications (all applications are GUI applications). This leads to the lack of a `pty` interface and hence the lack of network logins. While this shortcoming is expected to be fixed in the future with network-aware GUI interfaces, the current method of solving it is to use "NT services" which is an euphemisms of running a daemon process which in turn starts applications. We used that methods to implement automatic worker spawning via the Calypso User Interface. Spawning of workers from the user program is currently not available.

5. Calypso User Interface

Calypso NT 1.0 provides a graphical user interface (GUI) that provides the user with the ability to visually monitor the execution of a Calypso program, on the local machine as well as remote machines. The GUI is a separate process that monitors and controls the distributed execution supported by Calypso. It allows the user to start an application on a local machine as a manager and then specify additional machines on which workers may be started. However, each machine that hosts a worker process must also have a *client-demon* process running at the time the worker is started. This client-demon process is similar to the *rlogin* demon on UNIX and allows the GUI to request remote execution of worker processes on remote machines.

The GUI is initially used to start the execution of a Calypso application. Once started, the application will behave as a manager and communicate with the GUI using messages. The manager will inform the GUI of the change of status of any worker, of job assignments and of termination. The GUI in turn may dynamically start and stop worker processes on any machine on the network that is running a client-demon process.

The client-demon on each machine requires password verification before any request for remote execution is honored. This keeps random processes from sending requests for remote execution to these machines. The password that the client-demon recognizes is specified when it is started. The password that will accompany the GUI's remote execution requests, will be specified when the

manager process is started. The GUI's request. These requests will be honored only if password sent by the GUI matches that of the client-demon. This mechanism provides some measure of security, but does not eliminate security problems entirely. At present, these passwords are not encrypted when they are sent over the network. It is possible, therefore, for someone to snoop on the line and detect these password, making it possible to gain access to remote execution facilities on machines running client-demon processes.

Figure 3 shows the interface window as soon as it is started. The window has three distinct regions (upper, middle, and lower). The upper region is used to specify information required to start a Calypso application as a manager. As described in figure 3, the executable name is specified as well as the password that will accompany requests for remote execution. In addition, a button is provided to start the execution of the manager on the local machine.

The middle region of the window allows a list of machines may be specified (shown in Figure 4). These are machines on which one or more workers may be started. Since Calypso is fault tolerant, workers may be added or deleted at any time during the execution of the an application without affecting correctness of the result. In addition, dynamic load balancing makes sure that the workers on faster machines do more work than those on slower machines.

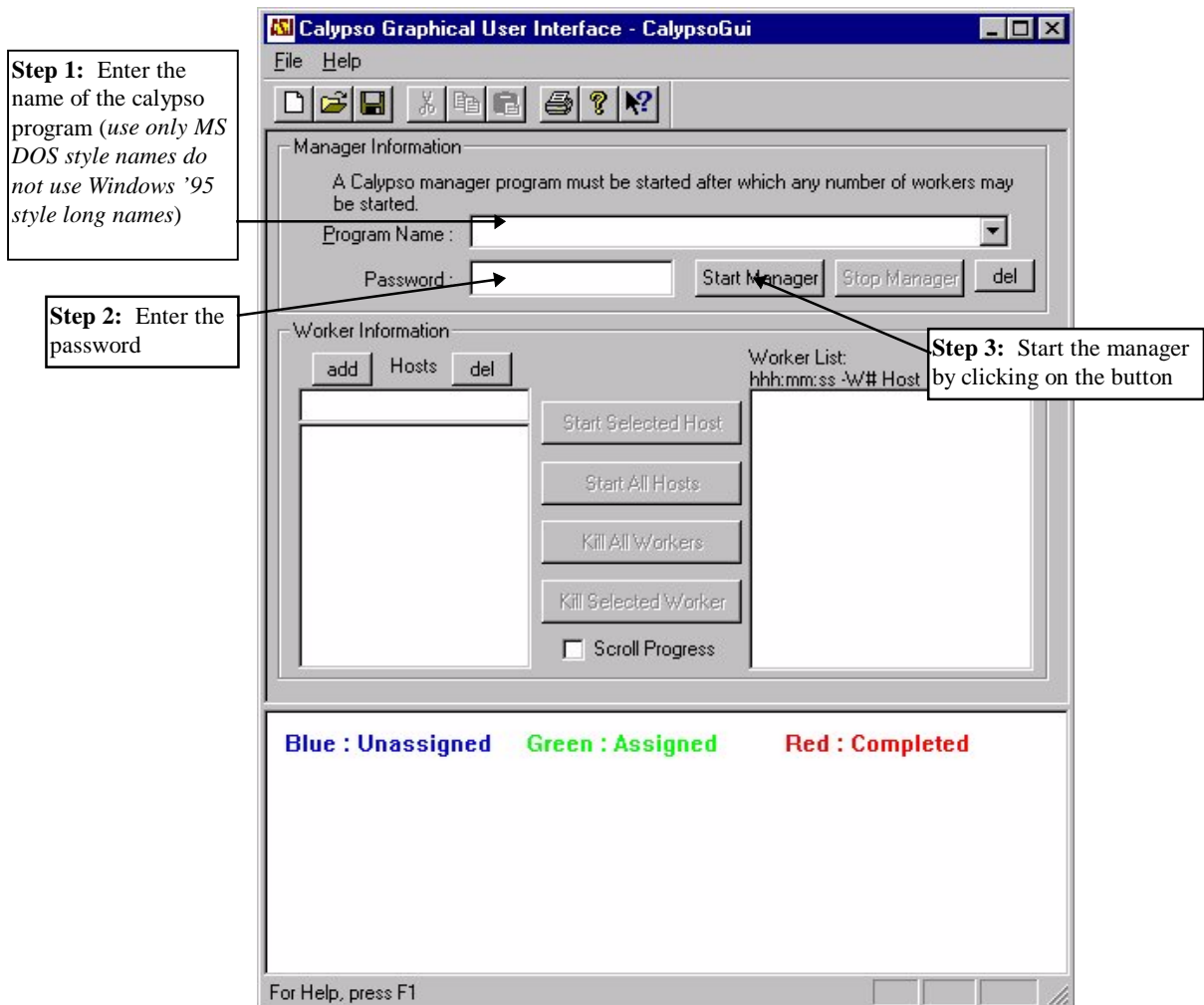


Figure 3: The initial window of the user interface

Finally, the lower region of the window provides a display that allows the progress a Calypso application to be monitored while under execution. As shown in Figure 4, vertical bars represent the thread segments of the parallel step under execution. The bars change color as the status of the thread segments they represent changes from unassigned to assigned, and later from assigned to completed.

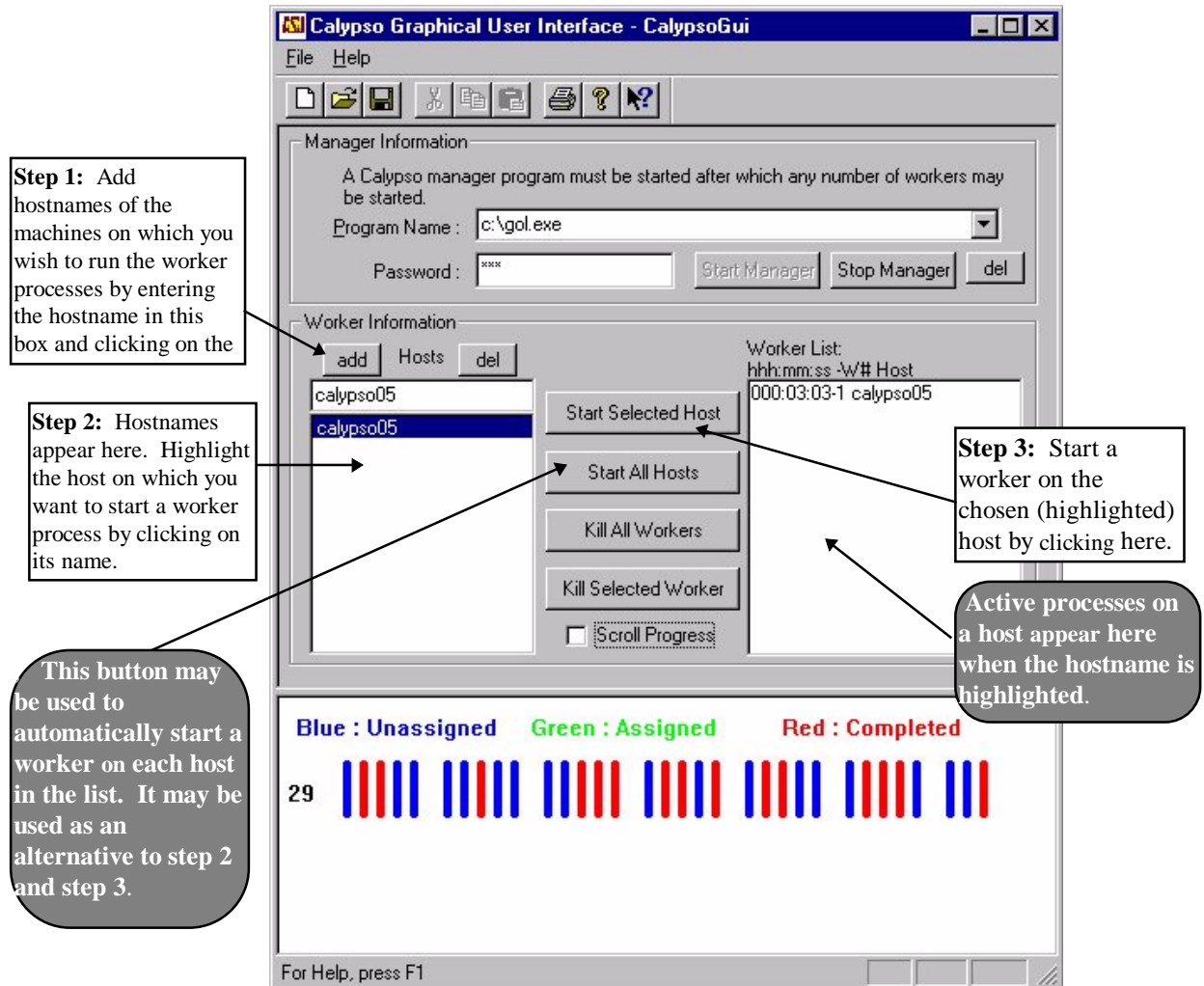


Figure 4: Monitoring the execution of a parallel program.

The above figure shows the interface while it is running a parallel program on one worker machine. The instructions show how to add and delete worker machines and how to start worker processes. Also, the figure shows the status display on the lower region of the window.

6. Performance Results

We now present the results of careful performance study of Calypso NT. For this purpose we used the following:

- 1] Five Pentium 90MHz (P-90) machines, with 32Mbytes of memory, running Windows NT 4.0 (Beta2).
- 2] Four Pentium 133MHz (P-133) machines configured as the P-90's.
- 3] A 100Mbit/Sec Ethernet.

- 4] The Calypso NT 1.0 software (available at <http://www.eas.asu.edu/~calypso>)
- 5] The Visual C++ compiler, with optimizations turned on.
- 6] A Ray-Tracing program with two parallel steps and 40 threads in each parallel step.

We used a Ray Tracing program for a variety of reasons. For one, a tunable, long running computation can be made to run for long periods by varying the detail of the output desired. Secondly, it provides a very good method of judging the accuracy of the computations. Since Calypso NT does many low-level memory exchanges, we were concerned if the output of a program was indeed, what it should be. With Ray Tracing we could look at the picture and visually detect errors, even if a few pixels were corrupted (and during debugging phases, such problems did occur). Thirdly, it produced pretty pictures during an otherwise boring session.

The Ray Tracing program was first written as a sequential version, that is a real proper sequential program with no Calypso embellishments. This program was then compiled with all possible compiler optimizations turned on. It ran on a Pentium-90 in 1037 seconds and ran on a Pentium-133 in 700 seconds. This ratio implies that a P-90/P-133 = $1037/700 = 1.481$ which is about the same as the ratio of their clock speeds: $133/90=1.477$. All timings in these tests were done with a real, physical stopwatch and not system times, or user times or any such measures. Thus they are real “*wallclock*” times.

Then we wrote the RayTracing program as a Calypso NT program (with two parallel steps and 40 threads per parallel step). The first test we ran was the “*speedup test*”. In this case we used the sequential execution as the base case. We then ran the Calypso program using one worker (a P-90) and the execution took 1042 seconds. The extra five seconds accounted for all overheads (networking, memory copying, scheduling and so on.). Figure 5 shows the performance results for a maximum of five P-90 machines.

Note that the speedups obtained are very close to ideal and is competitive (we believe) to other parallel processing systems such as PVM. This is in spite of Calypso NT providing support for shared memory and fault tolerance and load balancing. Thus, we claim we provide very useful lightweight methods for managing distributed computing.

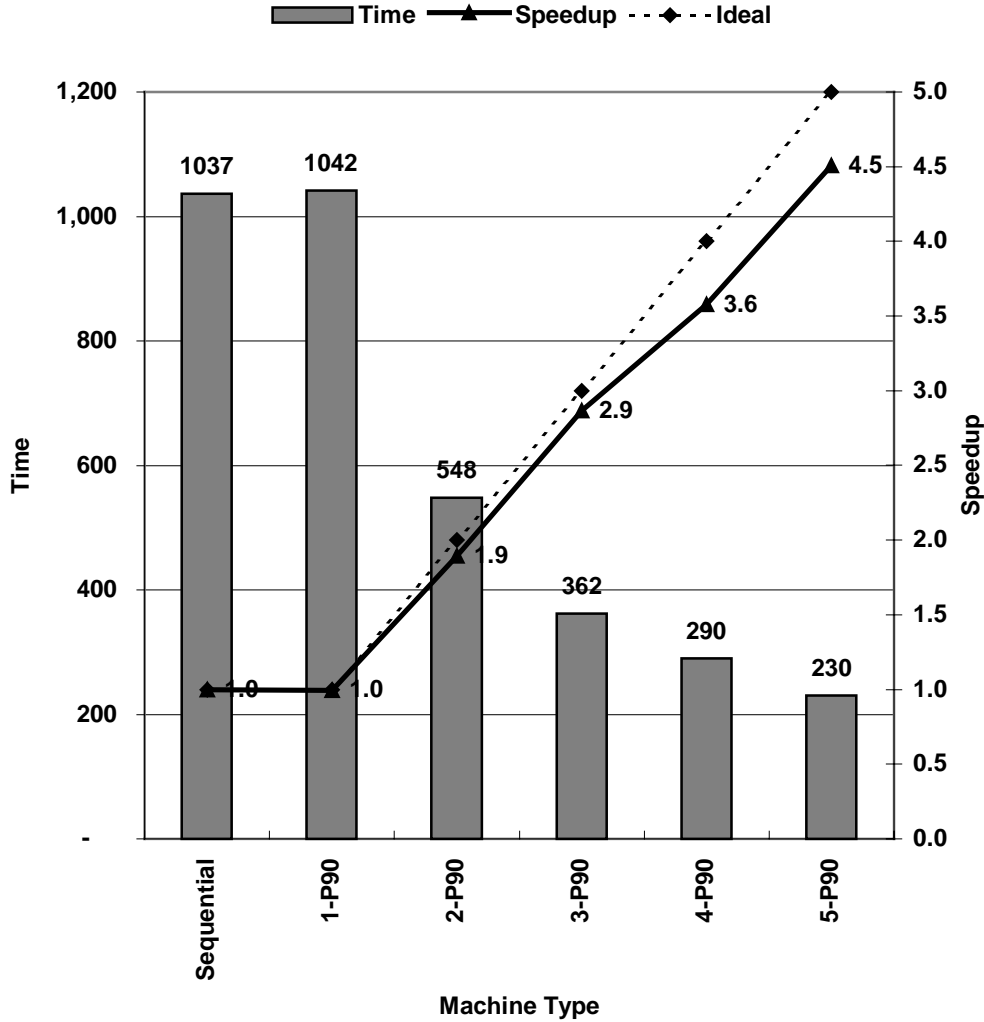


Figure 5: Speedup test, with up to five P-90 machines

The next test involves proper balancing of loads between machines of differing speeds. For this test, we intermixed the P-90 and the P-133 machines. In many parallel processing systems, the slowest machine(s) dictate performance; that is fast machines are held up for the slow machines to finish the work allocated to them. Calypso does it differently.

In tests with machines of different performance characteristics, we use a formula for calculating ideal machines. The number of ideal machines determines the maximum possible speedup. This formula is applicable for cases with varying machine speeds as well as cases where machines fail or join a computation dynamically. We use a very pessimistic method of calculating ideal speedup.

Suppose a computation runs for T seconds. It uses n machines, M_1, M_2, \dots, M_n . Machine M_i has a performance index of p_i and is available for the computation for t_i seconds. Performance index is the relative speed of a machine normalized to some reference, which has a performance index of 1. Then the maximum theoretical speedup that may be achieved is:

$$\text{Ideal speedup} = \text{number of equivalent machines} = \sum_{i=1:n} (p_i * t_i) / T$$

Note that in the above computation we add up all possible machine types and times that may participate in the computation. We even account for machine time that is available but not useful in our

computation. Thus, it is an extremely pessimistic measure and really represents the ultimate limit of possible speedup.

The next test is the “*load balancing*” test in which we run the Ray Tracing application using three machines, but vary the types of machines used. We range from using three P-90 to three P-133 machines. We assign the performance index of 1 to the P-90 machines and the performance index of 1.48 to the P-133 machines. In Figure 7, we show the results, consisting of the actual wallclock times, the actual speedup and the ideal speedup. As can be seen, adding the faster P-133 machines to the computation actually speeds the execution, and the actual speedup is quite close to the ideal speedup. Note that the actual speedup is computed as (execution time/sequential time) where sequential time is the time taken by an optimized sequential program.

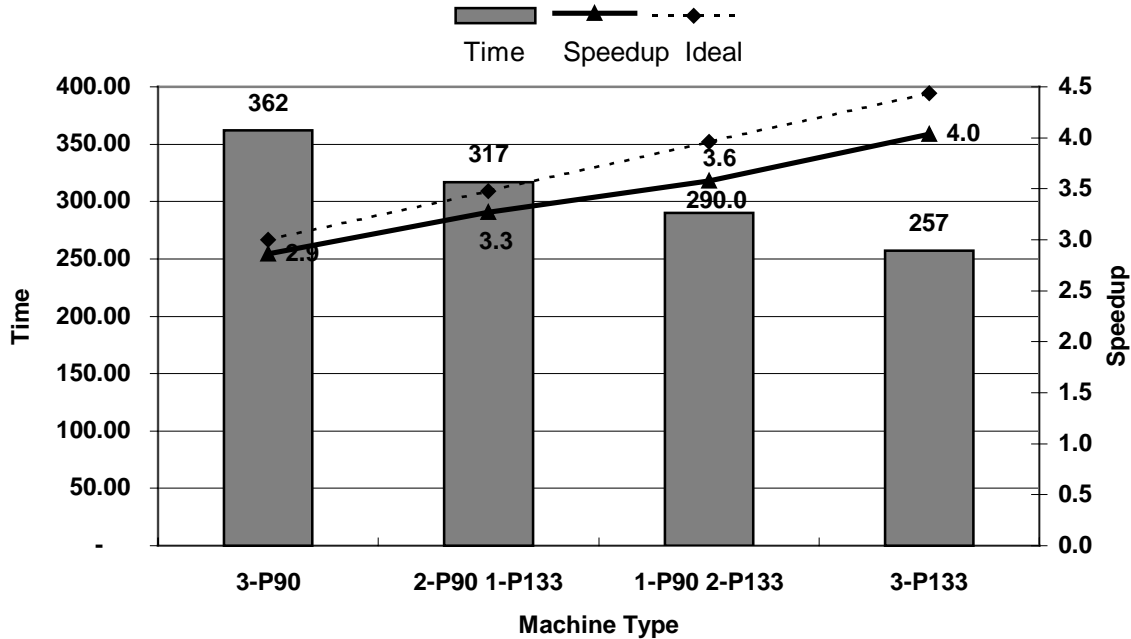


Figure 6: Load balancing test with slow and fast machines.

The final test is the “*fault tolerance*” test. It shows the ability of Calypso NT to dynamically handle failures as well as allowing new machines to join the computation. For this test we used up to four P-90 machines. Of these machines, one was a stable machine, and the rest were *transient* machines. The transient machines worked as follows:

Transient machine: After 180 seconds into the computation, the transient machine would join the computation and then, after another 180 seconds, fail (without warning).

Figure 8 shows the effect of transient machines, along with actual speedups and ideal speedups computed according to our formula. Note that the ideal speedup measure takes into account the full power of the transient machines whether they are useful to the computation or not. The experimental results show that the transient machines do contribute to the computation.

In fact this experiment shows the real power of Calypso. The system really handles load balancing and fault tolerance, with no additional overhead. The real speedups are very close to the ideal speedups. In cases where machines come and go, the failure tolerance features of Calypso actually provides more performance than an equivalent, non-fault-tolerant system.

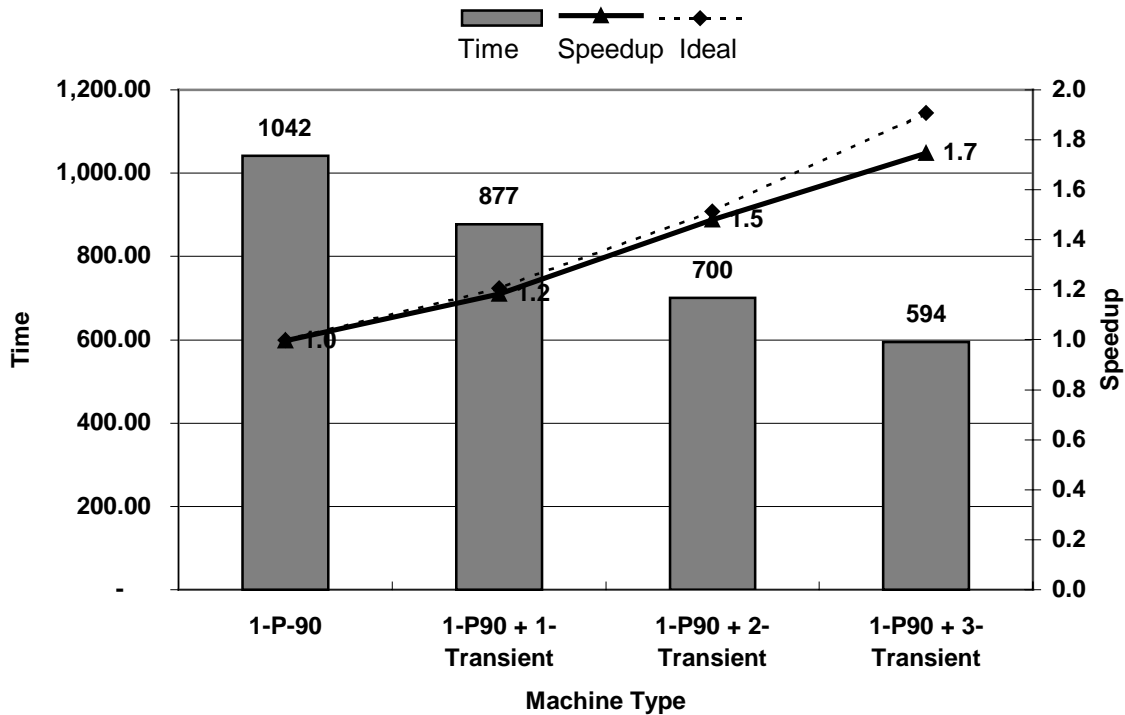


Figure 7: The Fault Tolerance Performance Test

Thus to conclude, the performance tests of Calypso NT shows that we provide an efficient environment for running shared memory parallel computations on a set of networked computers, with support for shared memory, load balancing and fault-tolerance.

7. Related Work

A large body of experimental results exists in the attempt to make parallel programs run on distributed hardware. 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 [Sun90, GS92], Orca [BT90], GLU [JA91], Isis and Horus [RBM96], and Concert/C [ABG+92, AGG+94] and so on. These systems provide a runtime library (and sometimes compiler support) to enable the writing of parallel programs as concurrently executable units. A control site then spawns these units on different machines on the network, typically. 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 [Li88] Munin [DCZ90], Clouds [DLA+90], Mether-NFS [MF89], 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 [CG89]. Instead of having a global address space, Linda employs a database type environment, the *tuple-space*. The tuple space (managed by tuple-servers) provides 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 generates result tuples. The system is interesting, and receptive to additional enhancements. However, programs still have to be re-written to use Linda.

Pirhana [GJK93] provides a feature 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 [Coo85], LOCUS [PWC+81], and Clouds [DLA+90], Isis [BJ87], Fail-safe PVM [LFS93], FT-Linda [BS93], and Plinda [AS91] 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 NOW[Pat+95] project at Berkeley, the HPC++[MMB+94] project and the Dome[NAB+95] project at CMU. All these projects however use approaches that are somewhat conventional (RPC or message based systems with provisions for fault detection, checkpointing, and so on.).

8. Acknowledgments

The Calypso project was started in 1994 at New York University by A. Baratloo, P. Dasgupta and Z. M. Kedem. Currently it is a joint effort of two groups of people at Arizona State University and New York University. The authors wish to acknowledge the contributions of Mehmet Karaul, Fabian Monroe and Rahul Tombre. In addition, the Calypso NT user interface was implemented by Maheshwar Gundelly and Alan Skousen. Some of the internal mechanisms were implemented by Sivamohan Vaddepuri.

Sponsor Acknowledgment: Effort sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0320. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Other sponsors include NSF, Intel and Microsoft.

Sponsor Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

9. References

- [ABG+92] J. Auerbach, D. Bacon, A. Goldberg, G. Goldszmidt, M. Kennedy, A. Lowry, J. Russell, W. Silverman, R. Strom, D. Yellin, and S. Yemini. High-Level Language Support for Programming Distributed Systems. *Proceedings of the 1992 International Conference on Computer Languages*, Oakland, California, April 1992.
- [ACD+95] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations, *IEEE Computer*, December 1995..
- [AGG+] J. Auerbach, A. Goldberg, G. Goldszmidt, A. Gopal, M. Kennedy, J. Rao, and J. Russell. Concert/C: A Language for Distributed Programming. *Usenix Winter Conference Proceedings*, San Francisco CA, 1994.

- [AS91] Brian Anderson and Dennis Shasha. Persistent Linda: Linda + Transactions + Query Processing. *Workshop on Research Directions in High-Level Parallel Programming Languages*, Mont Saint-Michel, France June 1991.
- [BCZ90] 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.
- [BBD+87] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, and R. Stevens. Portable Programs for Parallel Processors. *Holt, Rinehart and Winston, Inc.*, 1987.
- [BDK+94] A. Baratloo, P. Dasgupta, M. Karaul, Z. M. Kedem, and F. Monrose. *Calypso 0.8 Manual*, New York University, 1994. <http://www.cs.nyu.edu/calypso>
- [BDK95] A. Baratloo, P. Dasgupta, and Z. M. Kedem. A Novel Software System for Fault Tolerant Parallel Processing on Distributed Platforms. In *Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing, 1995*.
- [BJ87] K. P. Birman, and T. A. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions of Computer Systems*, Vol. 5, no. 1, pp. 47-76.
- [BLL88] B. Bershad, E. Lazowska, and H. Levy. PRESTO: A System for Object-Oriented Parallel Programming. *Software-Practice and Experience*, 18(8):713-732, 1988.
- [BS93] D. Bakken and R. Schlichting. Supporting Fault-Tolerant Parallel Programming in Linda. *Technical Report TR93-18*, The University of Arizona, 1993.
- [BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272-314, 1991.
- [BT88] H. Bal and A. Tanenbaum. Distributed Programming with Shared Data. *Proceedings of ICCL*, pages 82-91, Miami, FL, October 1988. IEEE, Computer Society Press.
- [BT90] H.E. Bal and A.S. Tanenbaum. Orca: A Language for Distributed Object-Based Programming. *SIGPLAN Notices*, 25(5):17-24, may 1990.
- [CAL+89] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy and R. J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147-158. ACM, 1989..
- [CAL96] The Calypso Home Page. <http://www.eas.asu.edu/~calypso>
- [CCK+95] J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MPVM: A Migration Transparent Version of PVM, *USENIX*, 8(2): pages 171-616, Spring 1995.
- [CG89] N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444-458, April 1989.
- [CK92] K. M. Chandy and C. Kesselman, *CC++: A Declarative Concurrent, Object Oriented Programming Notation*, Technical Report, CS-92-01, California Institute of Technology, 1992.
- [Coo85] E. Cooper. Replicated Distributed Programs, *Operating Systems Review*, 19(5), pp. 63-78, Dec 1985.
- [DKR95] 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 IEEE International Conference on Distributed Computing Systems, 1995*.
- [DLA+90] P. Dasgupta, R. J. LeBlanc Jr., M. Ahamad, and U. Ramachandran. The Clouds Distributed Operating System. *IEEE Computer*, 1990.
- [DLV+94] L. Dikken, F. van der Linden, J. Vesseur, and P. Sloot, Dynamic PVM -- Dynamic Load Balancing on Parallel Systems, *Proceedings Volume II: Networking and Tools*, pages 273-277. Springer-Verlag, Munich, Germany, 1994.
- [GJK93] David Gelernter, Marc Jourdenais, and David Kaminsky. Piranha Scheduling: Strategies and Their Implementation. *Technical Report 983*, Yale University Department of Computer Science, Sept. 1993.
- [GLS94] W. Gropp, E. Lusk, A. Skjellum. Using MPI Portable Parallel Programming with the Message Passing Interface. *MIT Press, 1994*, ISBN 0-262-57104-8.
- [GR88] N. Gehani and W. Roome. Concurrent C++: Concurrent Programming with Class(es), *Software -- Practice and Experience*, 18, 1157-1177, 1988.

- [GS92] G. A. Geist and V. S. Sunderam. Network-Based Concurrent Computing on the PVM System. *Concurrency: Practice and experience*, 4(4):293-311, 1992.
- [JA91] 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.
- [LFS93] 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.
- [LH89] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321-359, November 1989.
- [Li88] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the 1988 International Conference on Parallel Processing*, Volume II, pages 94-101, August 1988.
- [LLM88] M. Litzkow, M. Livny, and M. Mutka. Condor -- A Hunter of idle Workstations. *Proceedings of the 8th IEEE International Conference on Distributed Computing Systems*, pages 104-111, 1988.
- [LV90] S. Leutenegger and M. Vernon. The Performance of Multiprogrammed Multiprocessor Scheduling Policies. *Proceedings of the 1990 ACM SIGMET-RICS Conference on Measurement and Modeling of Computer Systems*, May 1990.
- [MF89] 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.
- [MMB+94] A. Malony, B. Mohr, P. Beckman, S. Yang, F. Bodin. Performance Analysis of pC++: A Portable Data-parallel Programming System Scalable Parallel Computers. In *Proceedings of the Eighth International Parallel Processing Symposium*, pp. 75-85, 1994.
- [NAB+95] J. Nagib, C. Árabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, P. Stephan. Dome: Parallel Programming in a Heterogeneous Multi-user Environment. *Technical Report CMU-CS-95-137*, Carnegie Mellon University Department of Computer Science, 1995.
- [Pat+94] D. Patterson et.al. A Case for Networks of Workstations: NOW, *IEEE Micro*, April 1996.
- [PWC+81] G. Popek and B. Walker and J. Chow and D. Edwards and C. Kline and G. Rudisin and G. Thiel, LOCUS: A Network Transparent, High Reliability Distributed System, *Operating Systems Review*, 15(5), pp. 169-177, Dec 1981.
- [RBM96] Robbert van Renesse, Kenneth P. Birman and Silvano Maffei. Horus, A Flexible Group Communication System, *Communications of the ACM*, April 1996.
- [Rich95] Jeffery Richter, *Advanced Windows: The Developers Guide to the Win32 API for Windows NT 3.5 and Windows 95*, Microsoft Press, Redmond, WA, 1995.
- [SB94] E. Seligman and A. Beguelin, High-Level Fault Tolerance in Distributed Programs, *Technical Report CMU-CS-94-223*, School of Computer, Science, Carnegie Mellon University, December 1994.
- [Stev90] W. Richard Stevens, UNIX Network Programming, *PTR Prentice Hall*, Englewood Cliffs, NJ, 1990.
- [Sun90] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315-339, 1990.