Parallel Processing with Windows NT Networks[1]

Partha Dasgupta
Department of Computer Science and Engineering
Arizona State University
Tempe AZ 85287-5406
Email: partha@asu.edu
http://cactus.eas.asu.edu/partha

Abstract

Workstation-based parallel processing is an area that is still dominated by Unix-based systems. We have been building new methods for shared-memory parallel processing systems on top of Windows NT based networks of machines.

As of present we have been involved in four related systems, called *Calypso NT*, *Chime*, *Malaxis* and *MILAN*. All of these are middleware, that is they are system level libraries and utility programs that allow programmers to utilize a network efficiently for high volume computations. Calypso was first built on Unix [BDK95], and then ported to Windows NT. Chime and Malaxis are NT systems and MILAN is still under the design phase.

This paper describes the systems, the techniques used to implement them on Windows NT and the roadblocks from a Unix programmer's point of view.

1. Introduction

This paper describes the experience of porting to and programming with Windows NT (from a Unix programmer's perspective) while implementing *four* related parallel processing projects on a network of computers. The paper also provides overview details about the parallel processing systems we have built. Our research efforts have produced system-level programs and libraries that run under Windows NT and allow parallel applications to utilize a network of workstations and have access to shared memory and provided fault tolerance. We have been working with Windows NT since January 1996 and have designed, implemented and ported, considerable amount of software. All the people involved with the projects had prior significant system development experience with Unix. Thus we started with our own set of *biases* and "*set-in*" ways. Therefore, switching to Windows NT has been fun at times and frustrating at others.

Two of the four projects under way have been implemented and tested. These are:

- *Calypso NT:* Calypso NT is a modification and port of a preceding Unix implementation. Calypso on Unix supports shared memory parallel programs that execute on a network of Sun machines and provides an efficient execution substrate coupled with low-cost fault tolerance and load balancing. The porting of Calypso to Calypso NT was our first experience with the Windows NT operating system. The Calypso NT system is being distributed for free and is available at *www.eas.asu.edu/~calypso*.
- *Chime:* Chime is an efficient, fault-tolerant implementation of the Compositional C++ parallel programming language on Windows NT. Some of the lessons learned from Calypso have migrated to Chime, but a lot of the inner architecture is different, due to a change in programming style from "Unix-ish" to "NT-ish".

Two newer NT-based projects are in progress. These are:

- *Malaxis:* An implementation of a Distributed Shared Memory (DSM) system for Windows NT. This system uses an innovative distributed locking mechanism that is coupled with barrier synchronization. This technique is expected to provide performance better than release consistent DSM systems. The actual performance has not been tested yet.
- *MILAN:* A metacomputing environment that extends the technologies prototyped in Calypso, Chime, Malaxis, and a Java-based system (Charlotte) [BKKW96], to provide a unified computing environment for robust general purpose computing on a networked platform.

Our current Windows NT platform consists of twelve Pentium-Pro-200 computers, four Pentium-133 computers and five Pentium-90 computers connected by a 100Mb/s Ethernet, running Windows NT 4.0. The development environment consists of Visual C++ 4.2.

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

2. From Unix to Windows NT

Like most academic research groups, we were a heavily Unix (SunOS) oriented group. Our research was hosted on Sun computers and all the participants were well versed in Unix and somewhat skeptical of Windows NT. So making the plunge into NT was venturing into uncharted waters.

As stated earlier, we started by attempting to port the Unix version of Calypso, to Windows NT. Since many of the mechanisms used by Calypso are operating system independent, we 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 we were wrong.

We then discovered the key differences between Unix and NT. These differences cannot be shoved under the rug via user-level libraries, and affects the porting of system-level software. The differences include:

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. Windows NT does not provide a "remote shell" feature.
4. Threads are the mechanism of choice for handling any form of asynchrony - including those tasks normally done through signals in Unix.
5. Windows NT expects the applications to be integrated with the windowing system and preferably such applications should be developed with the MFC (Microsoft Foundation Classes) library.
6. Terminology is different, making things confusing and sometimes exasperating.

In spite of such differences, the similarities between Unix and NT are quite striking. Due to the functional similarities, differences are easy to overlook. Some good articles on porting strategies from Unix to NT exist in the documentation library that is bundled with the development environment (a.k.a. MSDN — Microsoft Developers Network).

3. Parallel Processing on Networks

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 harnessed properly.[2] 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 that facilitates data transfer and process synchronization using messages. Some parallel processing systems do not differentiate between the parallelism inherent in an application and the parallelism available at execution time. In 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.

3.1 Prior Work

Significant numbers of parallel processing systems have been built for use in networked environment. The notable ones are PVM, MPI and Linda.

The parallel processing 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 [S90, GS92], GLU [JA91], Isis and so on. These systems provide a runtime library (and sometimes compiler support) to enable the writing of parallel programs as concurrently executable units.

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

[2] Effective and innovative harnessing of the computing power of workstation networks is one of the primary research goals of the MILAN project in general and Calypso in particular.

around Distributed Shared Memory (DSM) like IVY [Li88] Munin [DCZ90], TreadMarks [ACD+95] and Quarks [K96] and Clouds [DLA+90] 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]. Piranha [GJK93] provides a feature similar to Calypso in that is 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, the HPC++ [MMB+94] project, The Cilk project [BL97] and the Dome [NAB+95] project. All these projects however use approaches that are somewhat conventional (RPC or message based systems with provisions for fault detection, checkpointing, and so on.)

While the majority of the systems run on Unix, there are a few systems that run on Windows NT. These include Win-PVM, Win32-MPI and Brazos [SB97].

4. The Calypso System

The design of Calypso [BDK95, DKR95] addresses efficient, reliable parallel processing in a clean and efficient manner. In particular the Calypso NT [MSD97] 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.
- **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 is be kept separate from physical parallelism, which depends upon the number of workstations available at runtime.
- **Fault Tolerance:** The execution of parallel Calypso jobs is *resilient to failures*. Unlike other fault-tolerant systems, there is *no additional cost* associated with this feature in the absence of failures. [3]
- **Dynamic Load Balancing:** Calypso automatically distributes the workload among the available machines such that faster machines do more work compared to slower machines.
- **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. Furthermore, arbitrarily small update granularities in shared memory and the proper updates of memory are both supported by the collating differential memory mechanism.

The implementation of Calypso was first done on Unix. The Windows NT port preserves the Unix methodology

[3] This claim is well justified by our performance results. See section 8.

and replaces the signal handling and memory-faulting methods with NT specific handlers as described later.

5. The Chime System

Chime is an implementation of the shared memory part Compositional C++ [CK92] language on a network of Windows NT machines. The shared memory part of CC++ is designed for shared memory multiprocessors. It embodies many features that have been considered impossible if not difficult to implement on distributed machines. These include structured memory sharing (via use of cactus stacks), nested parallelism and inter-thread synchronization.

Chime implements these features of CC++, efficiently on a distributed system, making the distributed system look and feel like a real shared memory multiprocessor. In addition, Chime adds fault tolerance and load balancing.

A complete description of how Chime works is beyond the scope of this paper, but we will present some NT specific considerations.

The major difference between the implementation of Calypso and Chime is in the manner threads are used and contexts are migrated. Every site running Chime, runs two threads per process. The threads are called the "*Controlling*" thread and the "*Application*" thread. The controlling thread is responsible for all communication, memory-fault handling, thread context migration and scheduling. The application thread runs the code written by the programmer of the application.

As an example, consider the case when a application thread, running one of the parallel tasks of a parallel application decides to spawn a subtask, which is a nested parallel computation:

1. The application thread suspends itself and signals an event to the controlling thread.
2. The controlling thread saves the context of the application thread. This context will be used to create the parallel tasks on remote machines.
3. The controlling thread registers with a manager, the context of the application thread, the number of new tasks to be created, the stack of the application thread and the "continuation", i.e. the remainder of the application thread, that should be executed after the parallel tasks are over.
4. The manager then schedules the new threads on available machines.
5. A controlling thread on a worker machine picks up a task from the manager.
6. The controlling thread on the worker now crafts an application thread, with the same stack and context as the parent thread, and starts the thread.
7. Via some compiler tricks, the newly created thread executes the task it was supposed to execute.
8. Then, when the thread terminates, the updates it made to the global data and the stack are returned to the manager and its state is appropriately updated.

The above is just one aspect of the execution behavior implemented in Chime. The complete system supports proper scoping of variables, execution management, inter-thread synchronization and nested parallelism. The system consists of the Chime runtime library and a preprocessor that serve as a front end to the C++ compiler.

6. Malaxis and Milan

We are also building more systems using Windows NT. The two notable ones are (i) Malaxis, a DSM (Distributed Shared Memory) system that provides data locking and barrier synchronization (ii) MILAN, a metacomputing platform. Due to space constraints, the descriptions of these are omitted.

7. Using NT features

In order to implement software such as Calypso, and Chime, we needed some support from the operating system. The support included:

- Support for user level demand paging for implementing page-based distributed shared memory.
- Support for obtaining and setting thread contexts for implementing task migration and task scope preservation.
- Support for resetting the contents and the position of the user stack in order to implement distributed cactus stacks.
- Support for network communication.
- Support for asynchronous notification and exception handling.

It turned out that Windows NT supports all of these features—and in some ways more elegantly than Unix does. It was just necessary to expend considerable time and effort to work out the detail of usage and semantics.

Previously, when we were developing libraries for parallel computing in Unix we used the ubiquitous ASCII interface for all programs. The ASCII interface is *not* so ubiquitous under NT, in fact, it is thought of as arcane. NT programs that use textual interfacing are called "console applications". So far, except for some user

interfaces, most of out programs are console applications, though we intend to change this in the near future.

7.1 Memory Handling

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). These states allow (among other things) a set of threads to allocate address space and then later allocate memory when the need arises.[4]

Windows NT memory management is designed for use with threads and sounds like overkill to Unix programmers who are not heavy users of threads. We will discuss the threads issue in a later section. We found the functionality to be useful for allocating and protecting memory for supporting the dynamic distributed shared memory used by Calypso. The important memory management API functions are `VirtualAlloc` and `VirtualProtect`.

It is possible to protect any range of pages in memory against read or write or execute access. Access violation results in an exception, and the exception handler is provided with a plethora of information about the nature of the exception, something most Unix-based systems do not provide. Due to the tight coupling between the memory protection and exception handling, much of the Calypso memory system had to be reprogrammed, but the end result, we feel, is more elegant and extensible.

7.2 Exception handling

Exception handling was the major surprise. While UNIX uses signals, Windows NT uses *Structured Exception Handling (SEH).* SEH is *not* the same as C++ exception handling, which makes use of C++ keywords `throw`, `try` and `catch`. SEH uses the *`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 specify its activation. Once, the flow of control is out of the `_try` block, any raised exceptions can not be intercepted. While porting Calypso to NT, this restriction forced us to make some structural changes in the implementation.

In retrospect, there is nothing wrong in the NT approach. However for Unix programmers who think and breath `signals` the paradigm shift can be unnerving (it was for us). Also the lack of signals, at first made it look like doing things like process migration would be impossible. It is possible; it necessitates the use of threads.

7.3 Threads

Handling process migration and process context changes had us stumped for a bit. Without signals, a Unix programmer is lost! However, we soon discovered the power of threads under Windows NT.

Threads are one of Windows NT's strongest features. Its thread support is simple, elegant and works well. A thread is started by calling the `CreateThread` routine with a function as argument, the new thread executes the specified function. The threads are kernel scheduled and share all the global memory. Quite simple and intuitive. We have found threads to be quite useful, in many situations, specifically:

- Threads are very useful for process migration and implementing distributed stacks (next section).
- Threads are useful for distributed memory service; i.e. a thread can listen to incoming invalidations while another thread runs the computation.
- Threads are also useful in segregating functionality—even when threads are not really necessary. For example, in Chime, after a page fault, the faulting thread suspends itself, while a service thread acquires the page, installs it and restarts the faulting thread.

Windows NT support a variety of thread synchronization and thread control facilities, including the ability of one thread to stop another thread and load or store the other thread's context.

7.4 Process and Stack Migration

Task migration has been used in Calypso to implement pre-emptive scheduling, a topic outside the scope of this paper. Similar mechanisms have been used in Chime for setting the correct scope of tasks.

Suppose a process is executing, and we need to freeze it and restart it on another machine. Under Unix, we would send it a signal and let the signal handler handle the migration.

[4] Allocation of address space is different from allocating memory. A range of addresses can be allocated, without allocating the backing store.

Under Windows NT we use two threads for this purpose [MD97]. One thread listens to incoming messages while the other thread executes. We send a message to the listening thread. This thread suspends the executing thread and extracts its context and then ships the context over to another listening thread on the remote machine. The remote thread sets up the context of a new thread and starts it.

Similar mechanisms are used in Chime. In CC++ it is necessary for two parallel computations to share variable declared in the context of the function that started the parallel computations. This requires a "*distributed cactus stack*". The overview of the implementation steps for this case has been described in section ??. The actual mechanisms used are events to block and restart threads, and the API calls `GetThreadContext` and `SetThreadContext`. We are very happy to see that a thread context saved on one machine, *can be restored on another machine and the thread executes correctly*.

However, in general thread migration is very tricky in Windows NT due to the structure of the system. If a thread is in a DLL and its context is saved, can this context be restored on anther machine? We think not! Some tests reveal that this sometimes works and sometimes does not. However, how to find a "safe" place to save the context of an executing thread (without modifying user-written code) is still an open problem for us.

7.5 Networking

Networking with TCP-IP is almost identical under Unix and Windows NT, via the use *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.

7.6 Remote Execution

Creating distributed computations under Unix is simple due to the remote shell (`rsh`) feature. A process can easily spawn more processes on remote machines. Such a feature is not available under Windows NT, making distributed computations use some form of kludge. The preferred way of distributing the computation is to use RPC. While the RPC model is fine for client-server computations, its does not work well for "push" computations such as parallel processing.

We have used a temporary kludge, where a daemon process is started on the machine that will host the tasks of the parallel computation. This daemon listens to commands on a port and starts a process when instructed to do so. A better solution is to have a Windows NT "service" which starts up at boot time and then starts processes using the user-id of the remote user. We have experimented with such a service but have not tested it thoroughly yet.

7.7 Graphical Interfaces

As stated earlier, we used the console application feature to write most of our applications. That is, the application works in a "command window", which looks like the DOS shell, and in some ways similar to an `xterm`. However, all applications under Windows NT are expected to be integrated with the Windows GUI. For our GUI based interface, we cheated and used a Unix-like solution. A separate process runs the GUI and communicates with the controlling process (or manager) via messages, displaying the status and accepting commands. This works well, but is not a politically correct approach under Windows NT.

We are working on developing an event-driven framework that interfaces with MFC and other features in Windows NT to provide a better solution. We have not yet gained enough experience to make this part work (and we are systems programmers and not GUI experts).

8. Performance

The performance we obtained, both under Calypso and Chime were good, and comparable to performance obtained under Unix. For the speedup tests we took a RayTrace program and compiled a sequential version with all compiler optimization turned on. Then we took a Calypso (parallel) version and compiled it under the same optimizations. Then we ran the program on Pentium 90MHz machines and noted the wall clock times. The results are shown in Figure 1.

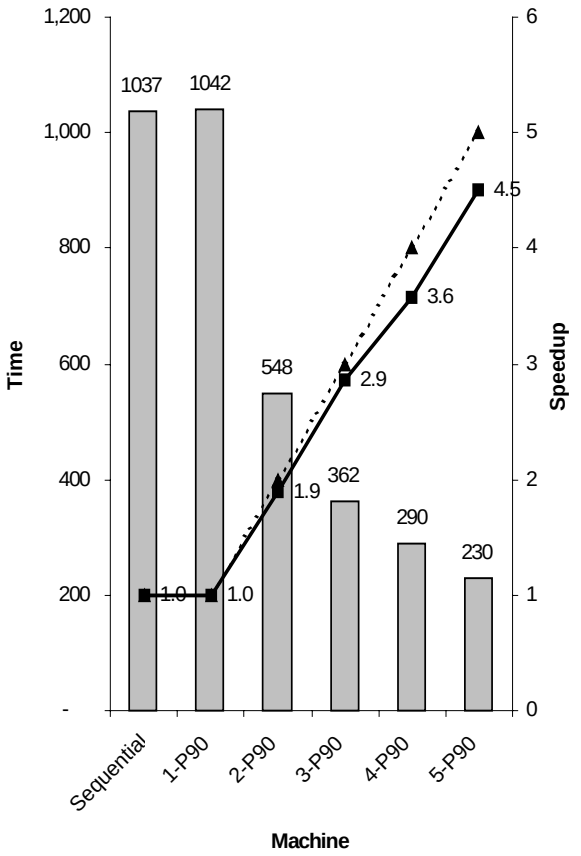

Figure 1: Speedup of Calypso

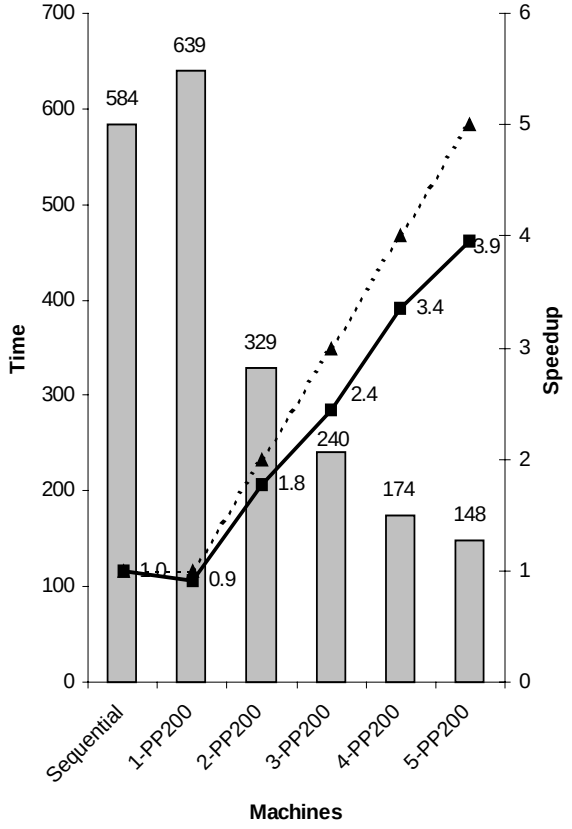

Figure 2: Speedup of Chime

The Calypso program took 1042 seconds to execute on one machine as opposed to 1037 seconds for the sequential program. This shows the low overhead of our mechanisms. We obtained a speedup of 4.5 on 5 machines in spite of providing load balancing and fault tolerance, showing these can be incorporated without additional overhead. This compares very well with Calypso on Unix, which produced the same speedup.

The Chime system is much more complex, and hence produces slightly lower performance. Figure 2 shows the result of running a similar (but larger) RayTrace program under Pentium-Pro 200MHz systems

We performed many other tests, including tests for load balancing (mixture of slow and fast machines) and fault tolerance (transient machines.) Many of these results can be found in [MSD97, SD97] and on the Web site.

A particular test of Chime is interesting. We ran a trivial program under Chime that takes a 1024 element array and initializes each element in parallel. To make the test rigorous, we did thread creation recursively, that is at the top level two tasks are created, which in turn creates two tasks each, until 1024 leaf tasks are created. The total number of tasks created by this program is 3068.

The resulting execution times on varying numbers of machines are shown in Figure 3.

This test shows that the task creation overhead varied from a high of 133 msec and saturates at about a low of 75 msec. The decrease in the time as machines are added is due to some parallelization of the overhead, while the asymptotic value is when the central manager saturates.

9. Further Down the NT Road

Windows NT has much more to offer than the features we have used. Many of the features are beneficial to application programmers and not quite to middleware service providers like us. But many of the more basic features are quite useful and interesting, although the learning curve is steep.

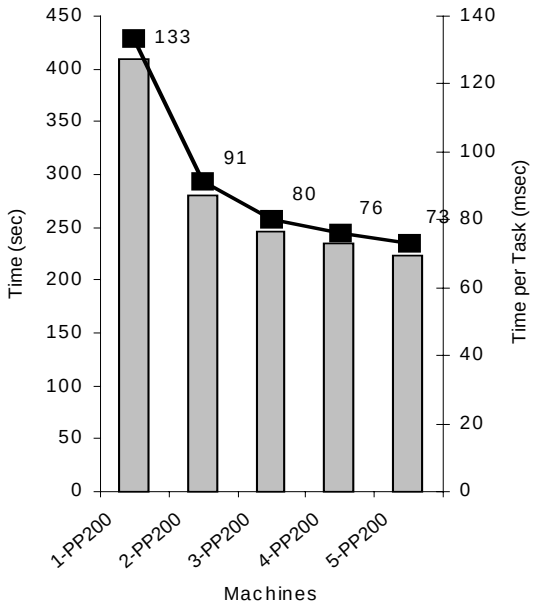

Figure 3: Task Spawning Overhead

Some of the features we would like to explore are:

- Microsoft Foundation Classes
- File system enhancements
- Device Drivers
- Services

The Microsoft Foundation Classes (or MFC) is indeed a powerful, somewhat intuitive (and somewhat confusing) array of prepackaged classes that make programming windows easier.

All our applications were console applications. A windows application has a built in message pump and event handlers. Since the Calypso/Chime systems are essentially libraries, use of a MFC based application is not precluded—however there is need for some modifications. Currently the library provides its own "main" program that set up the memory handling and the memory protection. This is not permissible under MFC programming.

We are looking into how to ensure MFC compatibility. This is not quite straightforward as our managers and workers all have to be automatically generatable from a user program. And the message pumps and the user interfaces need to be created as default, if the user does not specify them, or allow the user to build his/her own interface. This project will be investigated in the future.

Similarly, use of asynchronous I/O, various types of files (mirrored etc.) has advantages that we may be able to exploit. Loadable device drivers will allow us to use custom, lightweight, networking protocols, thus reducing parallel computation overhead.

Some of the features that we did not find a need for include COM, MAPI/TAPI, OLE and other eclectic and fancy support for high end application development.

10. In Retrospect

Windows NT has some very strong points. These include:

- Threads
- Structured Exception handling
- Good memory management
- Excellent program development environment.
- Huge library of online documentation.

And some shortcomings:

- No signals
- No remote execution facility. 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 `ptty` interface and hence the lack of network logins. While this shortcoming is expected to be fixed in the future with network-aware GUI interfaces.
- Confusing terminology that steepens the learning curve.

Overall, we were happy and impressed. The learning curve was sometimes steep and mostly curvy. Terminology differences were exasperating.[5]

We had to change a lot of programming strategies to suit the Windows NT way of doing things. But, in retrospect the changes we made were for the better.

- Using threads instead of signals for asynchronous event handling is better programming practice.
- Structured Event Handling - very weird when we first saw it, is a nice method of handling exceptions.

The integrated compiler/debugger/makefile system provided by Visual C++ was a wonderful tool to use. The debugging support for multi-threaded programs is fascinating, and without it, we would not ever had process migration to work.

[5] We received a CD-ROM entitled "Microsoft Developer's Library". We thought it contained some library routines that some developers would like to re-use. "*Not for us*", we thought. Turned out, it was chock full of books and articles—some really very useful. A real library!

In addition to the program development environment, of course NT opens up the world of PC computing applications. Office productivity tools, web development tools, personal productivity tools, shareware and freeware are readily available and of great quality. This was an added bonus.

So the final word is that all of the people working on the project unanimously state that is was a nice refreshing move from Unix to Windows NT. NT is a lot nicer system than what we had heard when we fist entered its maze of twisty little passages

11. Acknowledgements

The author wishes to acknowledge the members of the team who worked hard on making everything work on Windows NT. They include Donald McClaughlin, Shantanu Sardesai, Rahul Thombre, Alan Skousen, Siva Vaddepuri and Mahesh Gundelly.

12. Sponsor Acknowledgement/Disclaimer

This research is partially sponsored by the following:

- Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0320.
- The National Science Foundation under grant number CCR-9505519.
- Intel Corporation.
- Microsoft Corporation (software donations).

The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon.

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.

13. Availability

The Calypso NT system is available, for free, complete with documentation, user manual, sample programs, instructions, user interface and remote execution daemon at http://www.eas.asu.edu/~calypso. The Chime system will be available at the same site at a later date.

14. References

[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.

[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.

[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.

[BKKW96] A. Baratloo, M. Karaul, Z. Kedem and P. Wyckoff. Charlotte: Metacomputing on the Web. In *Proceedings of the 9th Intl. Conf. on Parallel and Distributed Computing Systems*, 1996.

[BL97] R. D. Blumofe, and P. A. Lisiecki Adaptive and Reliable Parallel Computing on Networks of Workstations, *USENIX 1997 Annual Technical Symposium*, 1997

[BS93] D. Bakken and R. Schlichting. Supporting Fault-Tolerant Parallel Programming in Linda. *Technical Report TR93-18*, The University of Arizona, 1993.

[CG89] N. Carriero and D.Gelernter. Linda in Context. *Comm. of ACM*, 32, 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.

[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.

[GBD+94] Al. Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Mancheck, and Vaidy Sunderam. *PVM: Parallel Virtual Machine*. The MIT Press, 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.

[HPF93] High Performance Fortran Forum. High Performance Fortran Language Specification Version 1.0, May 1993.

Also in Scientific Programming, Vol. 2, No. 1 and 2, Spring and Summer 1993; also Tech Report CRPC-TR92225, Rice University.

[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.

[JF92] R. Jagannathan and A. A. Faustini. GLU: A Hybrid Language for Parallel Applications Programming. Technical Report SRI-CSL-92-13, SRI International. 1992.

[K96] Dilip R. Khandekar. Quarks: Distributed Shared Memory as a Basic Building Block for Complex Parallel and Distributed Systems. Master's Thesis. University of Utah. March 1996.

[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.

[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.

[MD97] D. Mclaughlin and P. Dasgupta, Distributed Context Switching: A Technique to Speed up Parallel Computations. Available via www.eas.asu.edu/~calypso

[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.

[MSD97] D. Mclaughlin, S. Sardesai, and P. Dasgupta. Calypso NT: Reliable, Efficient Parallel Processing on Windows NT Networks, *Technical Report, TR-97-001, Department of Computer Science and Engineering, Arizona State University, 1997*. Also available via www.eas.asu.edu/~calypso

[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.

[R95] Jeffery Richter, Advanced Windows: The Developers Guide to the Win32 API for Widows NT 3.5 and Windows 95 , Microsoft Press, Redmond, WA, 1995.

[S90] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315-339, 1990.

[SB97] E. Speight and J. K. Bennet, Brazos: A Third Generation DSM System, USENIX Windows NT Workshop, 1997.

[SD97] S. Sardesai and P. Dasgupta, Chime: A Versatile Distributed Parallel Processing Environmen, *Technical Report, TR-97-002, Department of Computer Science and Engineering, Arizona State University, 1997*. Also available via www.eas.asu.edu/~calypso