CALYPSO: A Novel Software System for Fault-Tolerant Parallel Processing on Distributed Platforms*†

Arash Baratloo‡ New York University

Partha Dasgupta§ Arizona State University

Zvi M. Kedem¶ New York University

Abstract

The importance of adapting networks of workstations for use as parallel processing platforms is well established. However, current solutions do not always address important issues that exist in real networks. External factors like the sharing of resources, unpredictable behavior of the network, and failures, are present in multiuser networks and must be addressed.

CALYPSO *is a prototype software system for writing and executing parallel programs on non-dedicated platforms, based on COTS networked workstations, operating systems, and compilers. Among notable properties of the system are: (1) simple programming paradigm incorporating shared memory constructs and separating the programming and the execution parallelism, (2) transparent utilization of unreliable shared resources by providing dynamic load balancing and fault tolerance, and (3) effective performance for large classes of coarse-grained computations.*

We present the system and report our initial experiments and performance results in settings that closely resemble the dynamic behavior of a "real" network. Under varying work-load conditions, resource availability and process failures, the efficiency of the test program we present ranged from 84% to 94% benchmarked against a sequential program.

*This research was partially supported by the National Science Foundation under grant numbers CCR-91-03953, CCR-94-11590, and CCR-95-05519.

†For additional publications and information about the CALYPSO project, see the URLs: `ftp://cs.eas.asu.edu/pub/calypso/` and `ftp://cs.nyu.edu/pub/calypso/`

‡251 Mercer St., New York, NY 10012-1185, (212) 998-3350, baratloo@cs.nyu.edu.

§ASU, Tempe, AZ 85287-5406, (602) 965-5583, partha@cs.eas.asu.edu.

¶251 Mercer St., New York, NY 10012-1185, (212) 998-3101, kedem@cs.nyu.edu.

1 Goals and Properties of CALYPSO

Networks of workstations exist in many organizations, and their number is growing rapidly. These machines are mostly idle. Thus, it is very attractive and cost-effective to utilize this frequently wasted resource. Given the previous reasons, then why is it that programs that utilize networks of workstations have not proliferated? A major reason is that the cost to *harness* this power is too high. That is, although networks of workstations are a good value in terms of raw computing power (meaning hardware), the cost to harness this power (meaning software development) still remains high and unattractive.

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

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

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

CALYPSO is a prototype system for writing and executing parallel programs on multiuser networks. It it unique among other systems in that it provides the following features through a unified set of solutions.

- **Ease of Programming:** The programs are written in CALYPSO Source Language (CSL). CSL is C++ with added constructs to express parallelism. It is based on the shared memory model and it is very simple to learn and use. The programmer does not have to partition the data, or to specify how to synchronize or move it among the workstations.
- **Separation of Programming Parallelism from Execution Parallelism:** The parallelism inherent to a problem is independent of the number machines it will run on. So, why should a parallel program be tied to the number of available workstations—a number that in most cases is unpredictable and transient? This is a common weakness in many systems, and adds complexity to an already difficult program development. The mapping between program parallelism and execution parallelism is transparent in CALYPSO.
- **Dynamic Load Balancing and Fault Tolerance:** CALYPSO automatically distributes the work-load depending on the dynamics of the participating machines. Faster machines are not blocked waiting for slower machines to finish their "work assignments"—faster machines overtake the slower ones. In addition, CALYPSO executions are resilient to failures. All processes, other than a specific designated "manager" process, can fail at any point without affecting the correctness of the computation. In contrast with other fault-tolerant systems, there is effectively no additional cost associated with this feature in the absence of failures. This is confirmed by our initial observation that the performance of CALYPSO is comparable to other non-fault-tolerant systems.
- **High Performance:** While providing the features listed above, our initial experiments indicate that the overhead is small, and a large class of coarse-grained computations can benefit from CALYPSO.

The paradigm that CALYPSO embodies in a working system, was first described in [26]. That paper details a methodology for instrumenting general parallel programs to automatically obtain their fault-tolerant counterparts. While the solutions in [26] were formulated in the context of synchronous faults, they were later applied in [24] to a certain variant of asynchronous behavior (as does CALYPSO).

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

The importance of idempotence, and the utilization of the eager scheduling to take advantage of it, was discovered in [26] in an abstract context. (The term "eager scheduling" itself was coined recently.) Eager scheduling is a mechanism for assigning concurrently executable tasks to the available machines. Any machine can execute any "enabled" task independent of whether this task is already under execution by another machine. As a consequence, free machines end up doing more work than loaded ones, leading to a balanced system. Secondly, computations *do not stall* while dealing with system's asynchrony and faults. Thirdly, newly available machines can transparently be integrated into an executing computation. And finally, any of the machines that are "helping out" the parallel computation can fail or slow down at any time.

The mechanism of collating differential memory provides logical coherence and synchronization while

avoiding false sharing. It is an adaption and refinement of the *two-phase idempotent execution strategy* [26] among others. Memory updates are collated to assure exactly-once logical execution, and they are transmitted as bitwise differences, preventing false sharing. This supports efficient implementation of idempotence in addition to other performance benefits that we shall see later.

2 Previous and Related Work

CALYPSO has its roots in results by us and by our colleagues addressing fault tolerance, parallel program execution on fault-prone and asynchronous abstract machines, and distributed systems [30, 31, 11, 26, 12, 24, 14, 21, 23, 22, 4, 25, 3, 15, 13]. The research leading to CALYPSO started as formal work which developed provable methods for executing parallel computations, initially on abstract machines with crash-failing processors, and later on abstract machines with asynchronous processors. An outline of a network of workstations-based system for parallel computing based on earlier formal work was presented in [13]. CALYPSO is an evolution of this design, and is the result of considerable redesign and extensive experimentation on progressively more and more sophisticated implementations. Major new developments in CALYPSO include: (1) a programming strategy that allows dynamic thread segment declarations, thus providing programming scalability; (2) dynamically evaluated termination condition to increase efficiency; (3) addition of sequential steps to handle external interactions and side effects; (4) a global management mechanism which uses buffering, collating of updates and transmits updates as differences thus allowing arbitrary logical data granularity; (5) memory locality management. However, it does not implement a fault-tolerant manager as described in [13], which relied on dispersal and evasion.

Among the many systems that directly address parallel computing on workstation networks, some focus on providing message passing mechanisms. Message passing systems closely resemble the underlying hardware in a portable environment. Popular systems include PVM [16], P4 [9] and MPI [18]. The remote procedure call mechanism adds structure to message passing systems and makes programming a little simpler. Concert/C [2] from IBM and DCE-RPC [28] are examples of mature and portable packages. CALYPSO provides a high-level programming model that relieves the programmer from handling the underlying communication layer.

Another class of systems for parallel computing focuses on providing DSM (Distributed Shared Memory) across loosely-coupled workstations. IVY [29] was one of the first implementations of DSM. Midway [8], Munin [7], and now TreadMarks [27] and Quarks provide a weaker, and sometimes multiple consistency semantics in order to improve performance. In contrast, CALYPSO provides a simple, and a unified programming model that separates logical and execution parallelism. In addition, load balancing is not left for the programmer, rather provided by the system.

Linda [10] is a variant of DSM that provides a common global space. Piranha [17] is built on top of Linda and allows workstations to join an ongoing computation as they become idle, and retreat when reclaimed by their owners. In fact, our daemons are modeled after Piranha. But unlike all of the previous systems, CALYPSO can mask process crashes.

There have been several proposals to provide fault tolerance, mostly by augmenting an existing system. They include FT-PVM [32], FT-Linda [5], PLinda [19], and Orca [20]. A notable exception is DOME [1] that incorporated fault tolerance and load balancing form the onset. These systems provide fault tolerance by using well known mechanisms: checkpointing the data, logging messages, and using reliable atomic broadcasts. In contrast, CALYPSO uses a unified set of techniques to tolerate failures and slowdowns. Furthermore, in the absence of failures, there is virtually no overhead.

3 Syntax and Semantics of CALYPSO

Programs that run on the CALYPSO system are written in CSL. Programs are structured by inserting *parallel* tasks into *sequential* programs. We refer to an

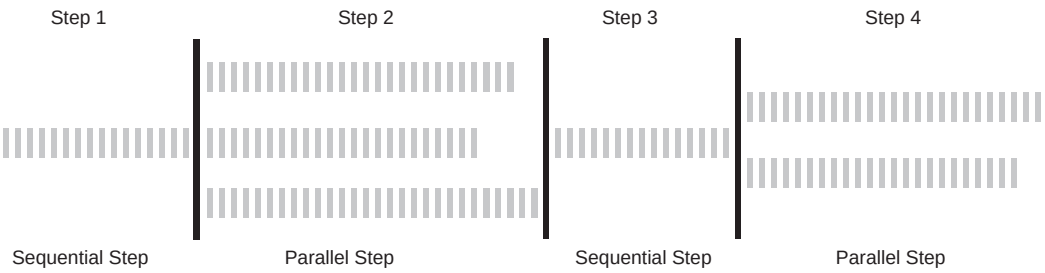


Figure 1: A fragment of an evolving execution.

execution of such a parallel task as a *parallel step*, and we refer to the sequential execution fragment between two consecutive parallel steps as a *sequential step*. The execution of a parallel step consists of several concurrent *thread segments*. Figure 1 illustrates a small fragment of an evolving execution.

CSL is C++ augmented with 4 keywords: `shared`, `parbegin`, `parend`, and `routine`. Informally, the major features and restrictions of CSL are as follows.

The address space of CSL programs is partitioned into two disjoint areas: private and shared. Shared variables are declared by:

`shared` { *member-list* }

A parallel step is a new compound statement to express parallelism. A typical form of one such statement is:

```
parbegin
  routine[int-exp] (int  width , int id )
        { routine-body }
parend
```

(There could be several `routine` statements.)

During the execution of a parallel step, for each `routine` statement several concurrent jobs are logically "spawned." We call these *thread segments*. The number of thread segments is specified by *int-exp*, and this number is passed to the first parameter of the *routine-body*, i.e. the *width* field. The second argument i.e. the *id* field, contains its unique number which ranges from 0 to *int-exp* -1. Thus, each thread segment is able to *adapt* and *distinguish* its behavior from that of its siblings by these two parameters. Once all thread segments are completed, the parallel step ends. (Dynamic termination conditions are allowed too.)

A *routine-body* is a sequential C++ program fragment. It can access the shared data, the two parameters passed to it (*width* and *id*), and its local data. It cannot have external effects (such as I/O).

Within a parallel step CR&EW (concurrent read *and* exclusive write, or multiple readers and a single writer) semantics are supported. In other words, a data item can be read by any number of thread segments *and* written by at most one. (In fact we have implemented the stronger Common CR&CW model.) Semantically, all thread segments read the value of shared variables at the beginning of the parallel step, and write atomically at the end of a parallel step.

The following is the *complete source code* of a CSL program that initializes two 500×500 matrices with pseudo-random numbers, multiplies them in parallel, and stores the result in the third matrix.

```
01 #include <calypso.H>
02 const int  N =  500;
03 shared {
04    float  A[N][N], B[N][N], C[N][N];
05 };
06
07 void rFill(float m[N][N], int size) {
08    for (int i=0; i<size; i++)
09       for (int j=0; j<size; j++)
```

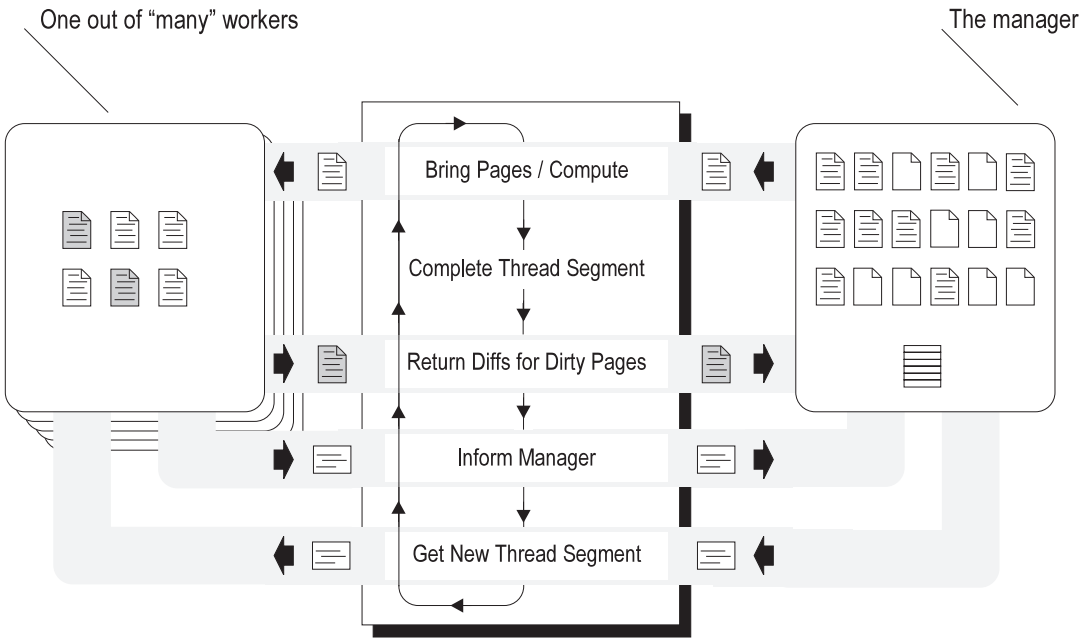


Figure 2: The software architecture of the current CALYPSO prototype

```
10          m[i][j] = rand();
11 }
12
13 void calypso_main(int ac, char *av[]) {
14    rFill(A, N, N);
15    rFill(B, N, N);
16    parbegin
17     routine[NUM] (int width, int id) {
18     int from = id * (N/width);
19     int to = from + (N/width);
20     for (int i=from; i<to; i++)
21        for (int j=0; j<N; j++) {
22           C[i][j] = 0;
23           for (int k=0; k<N; k++)
24              C[i][j] += A[i][k] * B[k][j];
25        }   // for
26     }      // routine
27    parend;
28 }  // calypso_main
```

Lines 3–5 define 3 shared arrays. Lines 14–15 initialize the input arrays `A` and `B`. Line 16 starts a parallel step. This step consists of only one routine, where `NUM` thread segments concurrently perform the multiplication and store the result in `C`.

4 From a Program to its Execution

In general the execution of a CSL program is distributed over a dynamically changing set of interconnected host machines. We do not need a specific network protocol or a shared file system for CALYPSO. In fact, we ran a CSL program on a set of machines, some in New York and some in Phoenix, with the machines at each location connected by an Ethernet, and the two locations connected by the Internet.

A CALYPSO computation is executed by exactly one *manager*, and a dynamically changing set of *worker* processes. In the current prototype the manager must be a completely reliable process. The workers may

come and go, speed up and slow down in an unpredictable manner, depending on the transient availability of resources and *not* on the properties of the computation. In Figure 2, we sketch the software architecture of CALYPSO.

Once a CSL program, say `program.csl` is written, it is preprocessed, compiled, linked, and then executed.

Our preprocessor reads a CSL program (e.g. `program.csl`) and generates a standard C++ program (e.g. `program.c`). Basically, routines are *stripped away* and wrapped in functions. As a very schematic example consider the pseudo-CSL program fragment:

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

The preprocessor replaces the parbegin-parend construct with calls to our library functions that will at run-time: (1) create m instances of functions each executing *SequenceOfStatements*, (2) assign executions of these functions to available workers, and (3) manage and monitor the execution.

Compilation of `program.c` using standard C++ compiler produces `program.o`, which is then linked with our library to produce `a.out`.

As mentioned before, the manager executes sequential steps. Once it reaches a parallel step, it suspends execution and "manages" the execution. Again for simplicity assume the previous CSL program fragment.

Say that *SequenceOfStatements* has been stripped and wrapped in a function called `foo123`. At the beginning of the parallel step the manager prepares a *progress table* and initializes it as follows:

Step	Function	Width	Id	Started	Finished
2	`foo123`	3	0	0	NO
2	`foo123`	3	1	0	NO
2	`foo123`	3	2	0	NO

The last row of the table, for instance, indicates that the current step is 2; that a thread segment defined by the function named `foo123` needs to computed; that there are 3 such thread segments defined by this function; that this rows describes the third out of 3 sibling thread segments (numbered 0, 1, 2); that 0 workers have started working on this thread segment; and that its computation has not yet finished.

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

Assume that at some point in the execution the first thread segment has been assigned to two workers, the second thread segment has been assigned to one worker, and the third thread segment has been given out to a worker, which has subsequently completed its execution. This is reflected by the following table:

Step	Function	Width	Id	Started	Finished
2	`foo123`	3	0	2	NO
2	`foo123`	3	1	1	NO
2	`foo123`	3	2	1	YES

We now turn to the description of a worker. The worker knows the shared pages—pages on which all and only shared variables are located. The worker access-protects the shared pages (using the system call `mprotect()`), and then contacts the manager for work. The manager then sends it an assignment specified by the 3 parameters `foo123`, `width`, and `id`. The worker now executes this assignment: it runs function `foo123` with parameters `width` and `id`. During this execution, the first time a worker accesses a protected shared variable a `SIGSEGV` signal is raised. The signal handler fetches the appropriate page from the manager, installs it in the worker's process space, and unprotects the page for future use. Then the computation proceeds. When the task terminates, the worker identifies all of its dirty memory and sends the differences (XORs) between the original page and the updated page to the manager. Then it protects its shared pages again and contacts the manager for another job to do.

The manager accepts the first completed execution of each thread segment and discards subsequent ones,

including the updates sent. The manager buffers the updates until the end of a parallel step, at which time all updates are performed. Different parts of a page can be updated by different workers, as long as the CR&EW condition is met. In a parallel step values read by a worker are those existing at the beginning of the step, and the updated values are readable only at the beginning of the next step. As a consequence, correctness is assured in spite of multiplicity of execution. Yet there is no need for expensive mechanisms such as distributed locking, and page shuttling is completely avoided.

The simplicity of CSL program lends itself to certain optimizations. For example, paged-in shared memory segments can be kept valid as long as possible, without paying the overhead associated with an invalidation protocol. So for instance, if a page was last modified in step 4, it was read by some worker in step 6, and that worker is working on a thread in step 8, then the worker does not fetch the page but accesses its cached copy. This is a low cost almost free strategy: read-only shared pages are fetched by a worker at most once; write-only shared pages are never fetched; modified shared pages are re-fetched only if necessary; and invalidation requests are piggybacked on the work assignments. *The programmer does not declare the type of coherence or caching technique to use, rather, the system dynamically adapts.*

Of course, the workers (and their locations) and thread segments are not related in any fundamental way. In fact, the current syntax of CSL *does not even allow* the programmer to specify the workers or how to distribute the data.

5 Experiments

Several applications have been implemented in CSL. These include both toy examples and real applications, such as several sorting algorithms, DFT, computation of eigenvalues and eigenvectors, Option-Adjusted-Spread bond indices [6], and modules of the Automatic Target Recognition software. We have tested the performance of some of these applications.

For a clear, simple, and complete illustration, we present the exact performance results for the specific program listed in Section 3. The program *does not contain any explicit code for load balancing or fault tolerance*, but is run under CALYPSO.

To model machines with different capabilities, machines joining and dropping from the computation, we used machines of three profiles: A, B, and C.

Machine A is "perfect," and available at 100% capacity throughout the computation. *Machine B* is "slow," perhaps timeshared with another independent sequential or parallel computation and therefore available only at 50% capacity. *Machine C* becomes available 60 seconds into the computation, and then disappears 120 seconds later. (See Figure 3.)

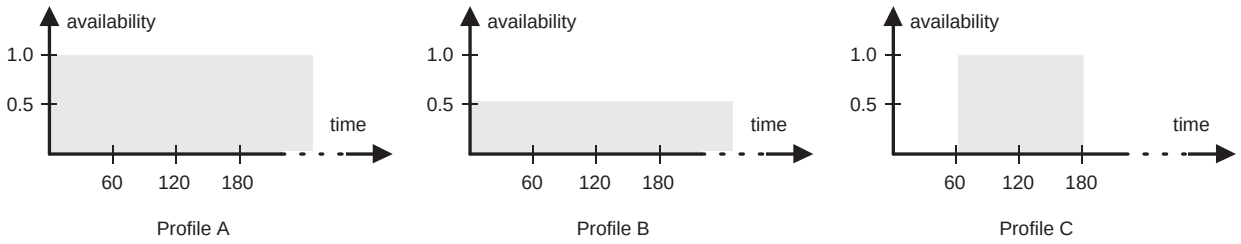
Each machine profile P, is defined by the function $\mathsf{availability}_P$ mapping time (in seconds) into the interval $[0,1]$. Then if the computation lasted for time T, the *work* that was made available to us is $\int_{t=0}^{T} \mathsf{availability}_P \, dt$. This is the area of the shaded region (for time interval $[0,T]$) in the top graphs of Figure 3. For a computation of time T, the *total work* is defined by $W = \sum_{i=1}^{n} \int_{t=0}^{T} \mathsf{availability}_{P_i} \, dt$.

The work W is the "charge" we incur for having the machines available to us during the computation, *whether we use it effectively or not.* Thus the overhead includes the network time, the time wasted by redoing computations, the time taken to move data between workers and the manager, the time spent by the operating system and other system activities.

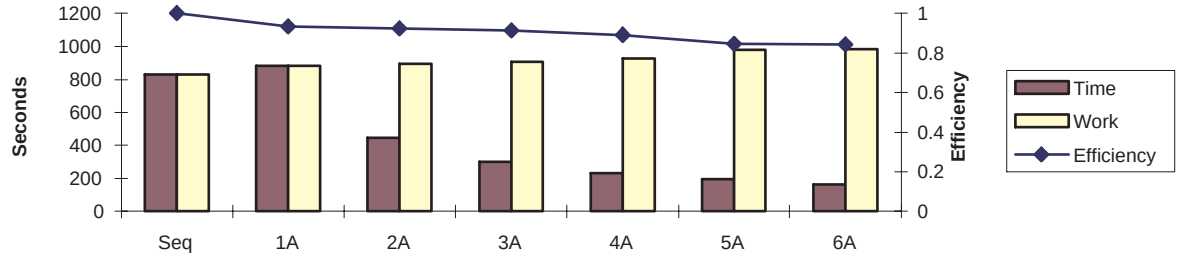
We used a *purely sequential* C++ matrix multiplication program as our base case, and compared the performance of the CSL program given earlier to it. The sequential program took 828 seconds, while running on one 100% available machine. Hence, $T_{\mathsf{base}} = 828$ seconds and $W_{\mathsf{base}} = 828$ machine-seconds.

Our performance metric is *efficiency*, defined by: $\mathsf{efficiency} = W_{\mathsf{base}}/W$. Efficiency is the important metric, since it measures how well we use the resources *provided* to us, bench-marked against the purely sequential case.

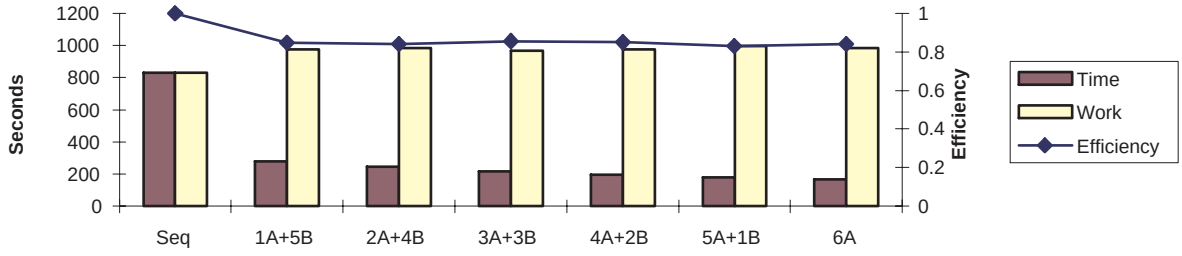
For our experiments we used Sun SLC workstations connected by a 10MBit Ethernet. Here we report three sets of experiments, where each experiment is labeled



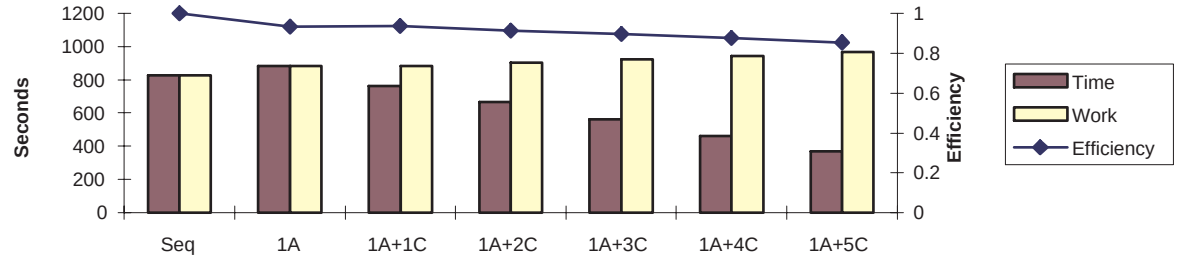
Profiles of the machines used in the performance experiments



Experiment 1: Variable Number of Fast Machines



Experiment Set 2: Fast and Slow Machines



Experiment 3: One Fast Machine and Variable "Temporary" Machines

Figure 3: Performance results for the matrix multiplication CSL program

with the profiles of the machines used, so 2A+4B indicates that there were 2 machines with profile A and 4 machines with profile B. "Seq," labels the execution time of the sequential matrix multiplication program.

We have accounted for the overhead of fault masking, load balancing, networking, swapping, updating memory, etc. The reported times were "wall clock," or elapsed times, *not* CPU or virtual times. As the pseudo-random filling of the input matrices is "phony," the times were measured for the execution starting with line 16. The efficiency ranged from 84% to 94%. (The latter for a CALYPSO execution with a single machine. Thus if we were to utilize this as the base case, the efficiency would be more than 89% for all cases.)

CALYPSO worked efficiently and transparently in the case of dynamic and unpredictable (to the programmer) availability patterns. *It utilized slow machines, integrated newly arrived machines, and bypassed machines that disappeared.*

Acknowledgments

We thank the following for their contributions to the CALYPSO project: Robert Buff, Churngwei Chu, Deepak Goyal, Shi-Chen Huang, Mehmet Karaul, Dimitri Krakovsky, Fabian Monrose, Naftali Schwartz, and David Stark.

References

[1] J. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan. DOME: Parallel programming in a heterogeneous multi-user environment. *Submitted to Supercomputing*, 1995.

[2] J. Auerbach, A. Goldberg, G. Goldszmidt, A. Gopal, M. Kennedy, J. Rao, and J. Russell. Concert/C: A language for distributed programming. In *Proc. of the Winter 1994 USENIX Conf.*, 1994.

[3] Y. Aumann, Z. Kedem, K. Palem, and M. Rabin. Highly efficient asynchronous execution of large-grained parallel programs. In *Proc. 34th IEEE Ann. Symp. on Foundations of Computer Science*, 1993.

[4] Y. Aumann and M. Rabin. Clock construction in fully asynchronous parallel systems and PRAM simulation. In *Proc. 33rd IEEE Ann. Symp. on Foundations of Computer Science*, 1992.

[5] D. Bakken and R. Schlichting. Supporting fault-tolerant parallel programming in Linda. Technical Report TR93-18, The University of Arizona, 1993.

[6] A. Baratloo, P. Dasgupta, Z. Kedem, and D. Krakovsky. CALYPSO goes to Wall Street: A case study. In *Proc. of Third Intl. Conf. on Artificial Intelligence Applications on Wall Street*, 1995.

[7] J. Bennett, J. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proc. 2nd Ann. Symp. on Principles and Practice of Parallel Programming*, 1990.

[8] B. Bershad, M. Zekauskas, and W. Sawdon. The Midway distributed shared memory system. In *Proc. COMPCON*, 1993.

[9] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston, Inc., 1987.

[10] N. Carriero and D. Gelernter. Linda in context. *C. ACM*, 32, 1989.

[11] P. Dasgupta, R. Ananthanarayanan, S. Menon, A. Mohindra, and R. Chen. Distributed programming with objects and threads in the Clouds system. *Computing Sytems*, 4, 1991.

[12] P. Dasgupta and R. C. Chen. Memory semantics for large grained persistent objects. In A. Dearle, G. Shaw, and S. Zdonick, editors, *Implementation of Persistent Object Systems.* Morgan Kaufman, 1990.

[13] P. Dasgupta, Z. Kedem, and M. Rabin. Parallel processing on networks of workstations: A fault-tolerant, high performance approach. In *Proc. of the 15th Intl. Conf. on Distributed Computing Systems*, 1995.

[14] P. Dasgupta, R. J. LeBlanc, M. Ahamad, and U. Ramachandran. The Clouds distributed operating system. *IEEE Computer*, 24, 1991.

[15] M. Fu and P. Dasgupta. Programming support for memory mapped persistent objects. In *Proc. COMPSAC*, 1993.

[16] G. Geist and V. Sunderam. Network-based concurrent computing on the PVM system. *Concurrency: Practice and experience*, 4, 1992.

[17] D. Gelernter, M. Jourdenais, and D. Kaminsky. Piranha scheduling: Strategies and their implementation. Technical Report TR-983, Yale University Department of Computer Science, 1993.

[18] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing-Interface*. MIT Press, 1994.

[19] K. Jeong and D. Shasha. Plinda 2.0: A transactional/checkpointing approach to fault tolerant linda. In *Proc. of the 13th Symp. on Reliable Distributed Systems*, 1994.

[20] M. Kaashoek, R. Michiels, H. Bal, and A. Tanenbaum. Transparent fault-tolerance in parallel Orca programs. *Symp. on Experiences with Distributed and Multiprocessor Systems*, 1992.

[21] Z. Kedem. Methods for handling faults and asynchrony in parallel computations. In *Proc. DARPA Software Technology Conf.*, 1992.

[22] Z. Kedem and K. Palem. Transformations for the automatic derivation of resilient parallel programs. In *Proc. IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, 1992.

[23] Z. Kedem, K. Palem, M. Rabin, and A. Raghunathan. Efficient program transformations for resilient parallel computation via randomization. In *Proc. 24th ACM Symp. on Theory of Computing*, 1992.

[24] Z. Kedem, K. Palem, A. Raghunathan, and P. Spirakis. Combining tentative and definite algorithms for very fast dependable parallel computing. In *Proc. 23rd ACM Symp. on Theory of Computing*, 1991.

[25] Z. Kedem, K. Palem, A. Raghunathan, and P. Spirakis. Resilient parallel computing on unreliable parallel machines. In A. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation*. Cambridge University Press, 1993.

[26] Z. Kedem, K. Palem, and P. Spirakis. Efficient robust parallel computations. In *Proc. 22nd ACM Symp. on Theory of Computing*, 1990.

[27] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations anperating systemsd. In *Proc. of the Winter 94 Usenix Conf.*, 1994.

[28] N. Leser. The Distributed Computing Environment naming architecture. *OpenForum*, 1992.

[29] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7, 1989.

[30] M. Rabin. Fingerprinting by random polynomials. Technical report, Harvard University, 1981.

[31] M. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *J. ACM*, 36, 1989.

[32] V. Sunderam, G. Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, 20, 1994.