

Parallel Processing on Networks of Workstations: A Fault-Tolerant, High Performance Approach*

Partha Dasgupta¹
Department of Computer Science
and Engineering.
Arizona State University
Tempe, AZ 85287-5406
Email: partha@asu.edu

Zvi M. Kedem²
Department of Computer Science,
Courant Institute
New York University,
New York, NY 10012.
Email: kedem@cs.nyu.edu

Michael O. Rabin³
Aiken Computation Lab.
Harvard University
33 Oxford Street
Cambridge, MA 02138
Email: rabin@das.harvard.edu

Abstract

One of the most sought after software innovation of this decade is the construction of systems using off-the-shelf workstations that actually deliver, and even surpass, the power and reliability of supercomputers. Using completely novel techniques: eager scheduling, evasive memory layouts and dispersed data management, it is possible to build a execution environment for parallel programs on workstation networks. These techniques were originally developed in a theoretical framework for an abstract machine which models a shared memory asynchronous multiprocessor.

The network of workstations platform presents an inherently asynchronous environment for the execution of our parallel program. This gives rise to substantial problems of correctness of the computation and of proper automatic load balancing of the work amongst the processors, so that a slow processor will not hold up the total computation. A limiting case of asynchrony is when a processor becomes infinitely slow, i.e fails. Our methodology copes with all these problems, as well as with memory failures. An interesting feature of this system is that it is neither a fault-tolerant system extended for parallel processing nor is it parallel processing system extended for fault tolerance. The same novel mechanisms ensure both properties.

1. Introduction

The basic tenet of high-performance computing is to allow multiple concurrent threads of control to perform multiple parts of a computation, while minimizing overheads of operating systems and other software/hardware artifacts. Such computations are typically run on expensive, dedicated hardware, which is often custom-designed. However, in recent years there has been a concerted push to use low-cost distributed hardware to execute parallel computations.

This paper describes a novel approach to harness the power of workstations for parallel computations requiring high performance platforms. Recently, new theoretical results have provided precise techniques that allow a formal

machine, modelled as an asynchronous multiprocessor (with shared memory), to execute large grained parallel programs efficiently (see section 3). Interestingly, these algorithms, with appropriate modifications, are also suitable for running parallel computations on networks of workstations. This paper presents the latter results.

In addition to supporting parallel computations, we do so in a fault-tolerant manner. This is an important consideration, as not only are workstations inherently unreliable, but the transient loading of workstations renders some of them slow (unpredictably). A fault-tolerance mechanism that treats slow computers as failed ones *can actually speed up* computations that would otherwise get bogged down by wrong choices during the initial scheduling process. In addition, in our approach we do not impose significant additional overhead for fault-tolerance or dynamic load balancing.

1.1. The Technology Setting

While there has been considerable pragmatic research on the subject of running parallel computations on a network of workstations, our approach is quite different from the current ones. Systems that support distributed computations generally rely on conventional techniques such as (1) low-level mechanisms like Distributed Shared Memory (DSM), Remote Procedure Calls (RPC), and Message-Based programming to implement applications that run on distributed nodes; (2) language support and runtime packages to write parallel programs and (3) system-level services like specialized servers, distributed operating systems and microkernels to support distributed computations.

In consequence, the following fundamental difficulties are frequently encountered: (1) The programming system necessitates major re-writing and/or restructuring of application programs to fit the specific model of the system. (2) Fault tolerance is either not supported or not well integrated with the rest of the system. (3) The overheads are high due to distributed control and synchronization strategies, or due to extensive replication for fault tolerance.

Our work starts with the premise that two conditions are critical for the utilization of networks of multiprogrammed workstations as virtual parallel computers. We would like programs to run on the network of workstations

* This research has been partially supported by the following grants:

¹ NSF: CCR-95-05519,

² NSF: CCR-91-03953 and NSF CCR-94-11590 and

³ NSF: CCR-93-13775, ONR N00014-91-J-1981.

with minimal rewriting. Also parallelism and fault-tolerance should be two integrated features of the system.

2. Related Research

This research is related to research in parallel processing in distributed environments as well as fault-tolerant processing. Pedagogically, these two areas are considered separate.

Parallel processing systems are built on top of a message passing (or RPC) layer as in the very popular PVM [Sun90] system. Other systems along similar lines include ORCA [BT90], GLU [JA91], Amber [CAL+89], Concert/C, and so on. Most of these systems are hard to program and they do not easily support fault-tolerance. Systems based on global address spaces, or distributed shared memory include IVY [LH89], Munin [BCZ90], Clouds [DAM+91, DCM+90, DLAR90], Mether-NFS [MF89]. They allow networked workstations to be treated as a multiprocessor system with the underlying software providing coherent memory. Page shuttling, false sharing, need for distributed locking and lack of fault-tolerance are the disadvantages. The tuple-space concept for sharing and synchronization has been effectively used in the Linda [CG89] system.

Fault Tolerance has been implemented using replication of data and computation or checkpointing. These systems include CIRCUS [Coo85], LOCUS [PWC+81] and Clouds [DLAR90]. Recently encoded techniques such as RAID [CLG+94] is gaining popularity. The process group approach and causal communications have been popularized by ISIS [Bir93].

Combining the parallel processing approaches with fault-tolerance has met with limited success. Mainly due to the fact that it compounds the overheads and these high overheads have to be paid *even when there is no failure*.

3. The Formal Foundations

This section presents some of the formal results which form the basis of our design. These results have been published in [AKPR93, Ked92, KP92, KPRR92, KPRS91, KPRS93, KPS90, Rab83, Rab89]. The formal results are developed in the context of abstract machines modeling some key properties of realistic highly-parallel machines. These results lead to precise provably correct and efficient techniques for execution of parallel computations in a fault-tolerant manner on these abstract machines. While this work is not directly implementable on a workstation network, we have been able to adapt the ideas to make such implementation feasible. In this section we discuss the key

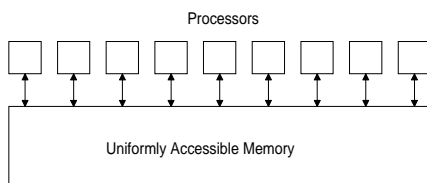


Fig. 1.: The Abstract Machine

points of the formal work in brief. A lot of detail is lost in the brevity, and the interested reader is referred to the original papers.

The formal research uses two abstract machine models: the ideal machine M as presented to the programmer, and the more realistic machine M_r which is used to execute the programs written for M . The formal techniques are used to automatically transform programs written for M , to execute on M_r .

The Ideal Machine M : The ideal machine M presented to the programmer is a synchronous shared memory machine with an unbounded number of perfect *virtual* processors. (See Fig. 1.) The programmer writes an *ideal* parallel program whose execution on M consists of a *sequence of parallel steps*. In each parallel step, some number of *thread segments* execute in parallel, each on a dedicated processor. A parallel step ends when all the thread segments in it have completed their execution. Then the next parallel step starts. (The execution model is shown in Fig 2.)

The Realistic Machine M_r : The realistic machine M_r has a finite number of *completely asynchronous* processors with unspecified memory organization but with all the processors able to access it. Any instruction can take potentially unbounded amounts of physical time to complete. (Processor faults are a special case — an instruction never completes.) The memory is fault-prone too. The *only atomic instructions* are *reads* and *writes*, to the main memory. The model M_r is a stable abstraction of some of the problems of a realistic hardware.

The formal research results provide a *method*, which given an ideal program P written for M , produces an

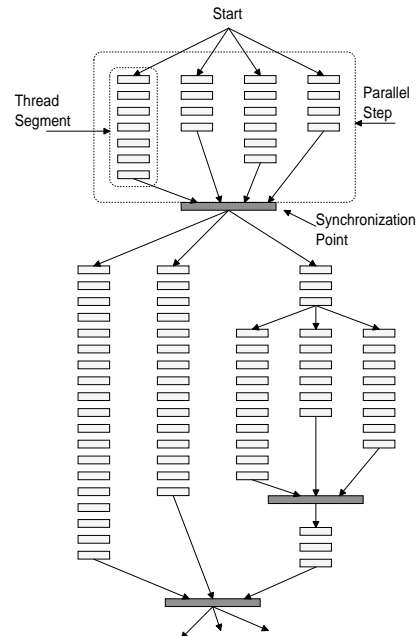


Fig. 2.: The Ideal Program Execution Model

execution of P on M_r , such that the execution is correct and efficient.

The execution is a sequence of parallel steps, logically numbered 1, 2, ..., etc. The *logical clock* maintains the current step number. A “free” processor of M_r *eagerly schedules* itself by “grabbing” a copy of a thread segment whose execution has not been completed in the current parallel step. It then executes the thread segment: reads variables, computes, and writes variables. A parallel step ends when each of its thread segments has been completed. Then the clock is advanced.

To ensure correct execution, each thread has to be executed logically once (*idempotence*) and multiple asynchronous executions of the threads should not update variables out of order (*clobber protection*). The execution ordering is assured by the *Certified Touch All* technique is used: During each parallel step, a data structure is maintained for keeping track of which thread segments have been executed. When all thread segments have been executed (touched), the clock is advanced. An efficient algorithm (essentially, $O(\log n)$) is used.

A critical difficulty is that an asynchronous processor can “clobber” a value. Consider a slow processor P_1 executing a thread segment of some step T , which will include a write of variable x . Since it is slow, a faster processor P_2 executes this thread segment, and the computation moves further. Then x is updated again in step T' . Subsequently, the now obsolete write of x by P_1 is executed, destroying the current value of x . Clearly the current values need to be protected.

We sketch here how this is done in the more interesting case of *large-grained variables*, say several hundred bytes or more. First, each variable is *dispersed*, that is, stored as a number *pieces* in an error-correcting manner with the property that the variable can be reconstructed from a fraction of the pieces. Second, *evasion* is used. When a processor writes the new values of the pieces of some variable x , it does not write them in the locations in which they have been stored so far, but in a *random* location dependent on the step number. To return to the previous example, pieces of x produced in step T' are written in locations different from the locations of pieces of x produced in step T . Note that a late write’s pieces can clobber pieces of current variables. However, with overwhelming probability, the rate of clobber will be one piece or fewer per variable. Because of dispersal, the correct value of a variable can be still recovered. The *Variable Handle Array* is a data structure that maintains the location of the current values of the variable. Because of the reuse of the storage, evasion can be done with a small space overhead. It is worth commenting that though evasion may appear to be similar to multi-version storage, it is in fact very different.

Evasion also assures idempotence of a thread segment execution, as in fact the input and the output locations are different, even though the names may be the same. (In $x := f(x)$, the new value of x is stored in a location different from where the old value of x was kept.) Note that the memory layout used to cope with asynchrony also provides fault tolerance. The system can tolerate loss of pieces due to reasons that have nothing to do with asynchrony, for example, a disk shutdown. To reiterate, dispersal and evasion together mask out clobbers, provide fault tolerance, and assure idempotence. (We remind the reader that we do not describe here how small-grained variables and control structures, such as the current addresses of the large-grained variables, and the clock are stored. In fact, the system design utilizes techniques different from those developed in the formal research.)

Sources: The first *Idempotent Execution* strategy (not relying on evasion), the *Certified Touch All* technique, and the related *Eager Scheduling* for it, were presented by Kedem, Palem, and Spirakis [KPS90]. Additional improvements were presented by Kedem, Palem, Raghunathan, and Spirakis [KPRS91]. (See also [Ked92, KP92, KPRS93].) The first *asynchronous parallel execution* (including the underlying asynchronous clock construction) was presented by Kedem, Palem, Rabin, and Raghunathan [KPRR92]. An improved construction was presented by Aumann and Rabin [AR93]. *Dispersed variables* and *fingerprinting* were presented by Rabin [Rab89, Rab81]. The *evasive memory* layout was presented by Aumann, Kedem, Palem, and Rabin [AKPR93].

4. System Design

In this section we present a design that allows us to use the ideas developed in the formal research to produce a system that runs on regular networked workstations and provides support for high-performance, fault-tolerant parallel computations.

Some of the techniques used in the formal design are either not implementable or not necessary in a real environment. For example, the Certified Touch All algorithm is an $O(\log n)$ implementation of a scanning algorithm, that is best done by a linear scan in a real system. Similarly, the evasion and dispersal algorithms, while initially looking esoteric, are actually quite useful after modifying the manner in which they are used. However the implementation of evasion and dispersal, as developed in theory are too expensive for practical systems. For instance, evasion requires that each write be preceded by a read that checks for existing data before overwriting. This overhead is too much for data access over a network. Similarly, in the theoretical case, the dispersal is done on top of evasion, while we prefer to use (modified) evasion over dispersal. Also, since there is no real global memory, the processor scheduling and variable

```

program EXAMPLE;
var I, J : integer;
procedure declarations;
begin
  cobegin
    begin S111; S112; S113; ... end;
    begin S121; S122; S123; ... end;
    begin S131; S132; S133; ... end;
  coend;
  cobegin
    begin S211; S212; S213; ... end;
    begin S221; S222; S223; ... end;
  coend;
end.

```



```

program EXAMPLE;
var I, J : integer;
procedure declarations;
procedure P11;
  begin S111; S112; S113; ... end;
procedure P12;
  begin S121; S122; S123; ... end;
procedure P13;
  begin S131; S132; S133; ... end;
procedure P21;
  begin S211; S212; S213; ... end;
procedure P22;
  begin S221; S222; S223; ... end;
begin
end.

```

Fig. 3: The source program and its pre-processed version

handling needs to be done differently. The good news is that when the formal algorithms are modified the resulting system architecture is feasible, innovative and quite appealing.

Some features of the design are:

- The execution environment uses a set of regular workstations running a conventional operating system. Some of these workstations are designated as *memory servers* and *progress managers*, the rest are *compute servers*.
- After a computation is started, any number of compute servers may fail, become inaccessible, or become heavily loaded by other computations. As long as there is one responsive compute server, the computation will progress (albeit, possibly slowly). The failures of memory servers and progress managers are also tolerated.
- A parallel program is written using barrier synchronization points, expressed by a **cobegin-coend** or equivalent structure.

In our presentation we make the following assumptions:

- The parallel program is a straightline sequence of parallel steps without nesting.
- If a thread segment writes a variable, then no other thread segment of the same parallel step writes the variable.
- Each variable is page aligned.

4.1. Compilation

A parallel program is written in a language that supports parallelism. This program is “pre-processed” into a sequential program in a standard language. The resulting program is compiled using a standard compiler.

To produce executable object code, we start with a parallel program. A *parallel step* is specified by a **cobegin - coend** block, and a *thread segment* (or thread) by a **begin - end** block. A sample program (written in pseudo-Pascal) is given on the left side of Fig. 3. There are some global declarations and the main body. Within each **begin - end** block, the statements are labeled **Sij1, Sij2, ...**.

This parallel program needs to be compiled into executable object code. The pre-processing stage transforms the source program by stripping away the **cobegin - coend** structure, replacing each **begin - end** block by a global procedure, and removing all statements from the main body. In fact, our program was turned into a set of procedures, each corresponding to a thread (segment). See the right part of Fig. 3. The pre-processor assigned names to these global procedures (e.g. **Pij**) as shown in the figure.

Then the transformed program is compiled into object code by a standard “sequential” compiler. In addition, we extract the starting point of each procedure from the compiler-generated symbol table and create a data structure called the *Progress Table*. The structure of this table is shown below:

Thread Id.	Start Address	Started?	Done?
P11	0x0ppp	F	F
P12	0x0qqq	F	F
P13	0x0rrr	F	F
P21	0x0sss	F	F
...

This table maintains information about where each procedure **Pij** starts in the code. The object code and the Progress Table are placed on a *memory server*, which is a system component that stores the program and its data and sends it over the network to a requesting workstation. For the first-cut design, we assume the memory server is *stable*, that is, it never fails. We will remove this assumption later.

4.2. Memory Service and Execution

As stated before, the stable memory server has the program, data and the Progress Table. In addition to the thread-index (or thread-identifier) and starting location of that thread, the Progress Table also indicates whether a thread has been started and/or possibly completed. All the *compute servers* have a *worker daemon*, which obtains information from the Progress Table of a computation if the

workstation is under-utilized. Again, for simplicity let us assume there is only one parallel computation and hence only one Progress Table. Although in fact, the Progress Tables are maintained and accessed by *progress managers*, for the time being we proceed as if any compute server can access the Progress Table in a mutually exclusive fashion. (Later in the presentation, we modify this.)

When a compute server examines the Progress Table, it follows the algorithm:

1. Find a thread segment $P_{i,j}$ such that: all the threads $P_{i-1,x}$ (for all x) have been completed, and $P_{i,j}$ has not been started; else if no such thread is found, then find a thread $P_{i,j}$ such that it has been started but not completed; else stop (as the computation has completed).
2. Mark the “started?” value corresponding to $P_{i,j}$ as true.
3. Get the page containing the memory location $start_{i,j}$ from the memory server
4. Start executing $P_{i,j}$.

While the thread segment is executing, it needs *code* and *data pages*. As the computation touches each page, this page is demand-paged into the compute server. The computation works on the downloaded copy of the page for *as long as the thread segment runs. No attempt is made to keep the pages coherent as is done in a DSM system.* After the execution of the thread segment completes, the compute server returns all the modified (“dirty”) pages. It does this by first identifying the dirty pages, using the modified bits in the paging system, and then writing the pages to the memory server. Finally, the compute server sets the variable “done” corresponding to $P_{i,j}$ to true in the Progress Table.

Thus, every under-utilized workstation picks up a thread segment that can be executed (i.e., this thread segment’s execution has not been completed, but all the thread segments in the previous parallel step have been completed). This is the crux of providing eager scheduling of a somewhat wait-free flavor, which also provides fault tolerance. Notable properties on the system are:

- No active compute server waits for any other compute server to finish. If there is a thread that can be executed, a compute server may work on it, even if other compute servers are executing it.
- If a compute server becomes slow or fails, other compute servers pick up the slack, without any global coordination. Thus load balancing is automatic and dynamic.

However, the design needs several additional enhancements. The major problem with this design is that memory gets “clobbered.” To reiterate the situation described in sec. 3, suppose a compute server W_a is slow, and W_b picks up the thread executing on W_a and finishes executing it. The parallel step then completes and the computation now progresses to the next parallel step. Much later, W_a completes and writes back the results. This would destroy the consistency of the computation. Finally, we cannot assume the

existence of a reliable memory server. The reliable memory server can be constructed from a set of replicated memory servers, but the overhead in synchronization (or atomic broadcasting) at every read and write is very high.

4.3. Dispersal and Evasion

A possible solution to this problem is as follows. Use a centralized memory server, with built-in intelligence (that is, a transaction system) that makes every set of writes from a thread atomic as well as rejecting writes from late threads. Then we can replicate this facility to provide fault tolerance. This conventional solution is too expensive, complicated to implement, and infeasible for regular programming systems. Motivated by the techniques described in sec. 3, we do it differently, employing variants of evasion and dispersal.

The basic idea behind evasion is the separation of variable name from variable address. While in most programming environments, a variable x is allocated an address A_x and this address remains constant throughout the execution. In our case, a variable’s address changes every time it is written.

We explain the evasion scheme first. Assume the memory server is reliable, but incapable of determining which write is a valid write and which is too late. This incapacity is necessary as we do not want to build global control-state information (i.e. step number) into the memory service. The address space of a program occupies some N pages, P_1 to P_N . Initially, these N pages are stored in blocks 1 to N of the memory server. Similarly to the Progress Table, we keep a *Page Allocation Table*. Again, we assume all compute servers have efficient and exclusive access to the page allocation table. We will describe how this is actually done in a later section. The page allocation table has the format:

Page No.	Step No.	Writer-id	Block No.
1	0	0	1
2	0	0	2
3	0	0	3
4	0	0	4
...

When a thread segment $P_{i,j}$ running on compute server W reads a page p , and p has not been modified by any other thread segment, $P_{i,j}$ reads from the initial location (or block number) p . If $P_{i,j}$ updates the page, instead of writing it back to the block p of the memory server, it writes it to any free block on the memory server storage, say block q . After all the pages are written back, the processor then updates the Page Allocation Table by adding entries containing the Page Number p , the Writer-ID W , the Step Number i and the new location of this page q . It does this for all the pages written.

In order to ensure that a thread reads the correct data when it executes, the following algorithm is used. When a thread $P_{i,j}$ reads a page p :

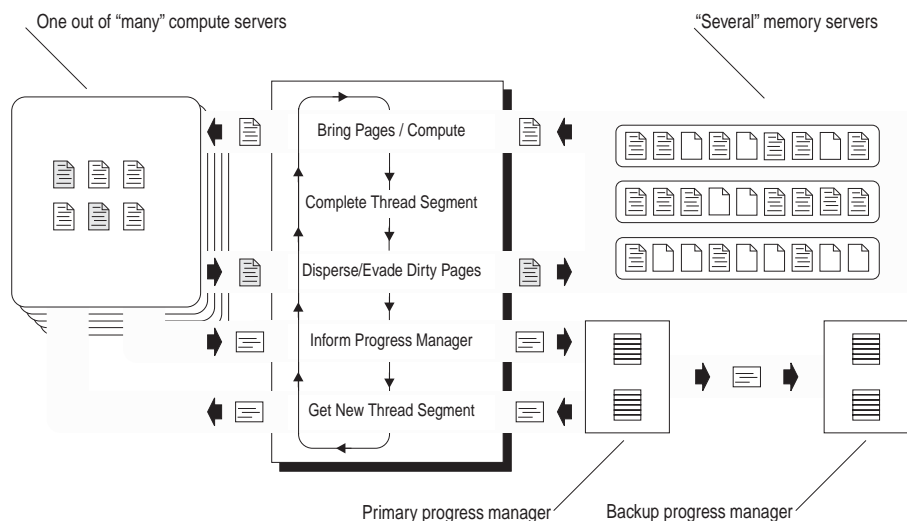


Fig. 4: The Overall Runtime System Structure

- It gets a copy of the Page Allocation Table.
- It finds the entry with p in the first column, such that the value in the third column is the highest value that is less than i .
- It retrieves the location q from the last column of the table and reads in block q from the memory server.

To ensure that the free space is not used up, a simple garbage collector discards pages that are too old by using the information in the Progress Table.

This implementation of evasion ensures that the execution of the threads are idempotent, slow threads do not clobber fast threads and the memory is utilized efficiently.

To make the memory server reliable, we use the dispersal technique, as described in sec. 3. To re-iterate: each block of disk storage is split into several blocks using a particularly efficient method for bulk error correction called the Information Dispersal Algorithm (IDA) [Rab89]. This method allows us to disperse a variable into n pieces so that every m pieces suffice for complete reconstruction of its value. The size of the storage required increases by n/m in the process. Thus when $n=5$ and $m=3$, we can lose any two pieces and reconstruct the value from the remaining 3; the storage and subsequent communication overhead is just 66% compared to the 200% overhead needed to achieve such resilience through standard replication.

Then these pieces are written on n different memory servers. When a variable is read, we need to retrieve m blocks from the memory servers and hence $n-m$ memory servers can be faulty at this point. Since n and m are parameterizable (depending upon the degree of fault tolerance desired) we can fine tune the system accordingly. Since the dispersed disk stores evasive memory there is no need for maintaining inter-update temporal order.

The above realization of evasion and dispersal keeps the original properties of these two schemes intact. These are: (1) Memory does *not* get clobbered. (2) A set of writes from

a single thread segment is atomic. The atomicity is established when the table entries are added. (3) A thread segment's execution, regardless of its structure, is *idempotent*. That is, it can be re-executed without adverse effects. Thus, multiple processors can run the same thread as well as allow slow processors to continue without the need for tracking them down and cleaning up after them.

4.4. Tables and Progress Managers

The key to efficient working of the system depends upon the storage and management of the Progress Table and the Page Allocation Table. The tables are kept in a designated machine called the *progress manager*. The progress manager is one central site that both stores and updates the tables and the free list.

A compute server wanting to run a thread sends a request to the progress manager, and the progress manager returns the thread-id and the start-address of the procedure to run, along with a complete copy of the page allocation table, as well as a list of free blocks for the output. When the thread completes execution, it disperses the output, and writes them to some of the free blocks. It then informs the progress manager of which pages were updated. The progress manager then marks the thread as done, and updates the page allocation table and free list as appropriate. As a part of the reply to this request, the progress manager sends the compute server another thread and the current version of the page allocation table. Thus about two messages per thread execution is required.

So the progress manager is the only *single point of failure* on our system. To mitigate that, we keep a backup progress manager. At certain intervals, the primary manager checkpoints the tables to the backup manager. If the primary manager fails, the backup manager can take over *even if the backup manager data is somewhat out of date*. Thus the backup does not have to be kept totally in sync with the

primary, and this is due to the availability of the evasive memory scheme.

The overall system structure is shown in Fig. 4.

4.5. Comparative Discussion

As mentioned earlier, most systems start out either as parallel processing systems or as fault-tolerant systems and then add on the remaining half. *The major conceptual advantage of this approach is that ours is neither a parallel processing environment augmented to support fault tolerance, nor is it a fault-tolerant system augmented to do parallel processing.* The same set of mechanisms provides both.

A negative aspect of systems that use replicated data or replicated computations for fault tolerance is that they bear significant runtime overhead *even when there are no faults.* Since faults are rare, the system pays a high price for running any fault-tolerant computation. Our system does not use replication. The data integrity is provided by dispersal, and when compared to replication, dispersal is cheaper. Due to parameterizability of dispersal, the possibilities of fine tuning the system is larger. Of course, these properties arise from the theoretical foundations of the system.

Our system does not replicate computations in a conventional way. Only when the execution of some thread segment seems to be delayed, and there are free computing resources, another execution of such thread segment is started at another compute server. Such *eager scheduling* used by our restart execution strategy, provides automatic load balancing, without the need for process migration.

The table management seems like a bottleneck, but in reality, it is not. The table management system can be made efficient by reducing the number of messages. Indeed each process can update all the tables and allocate the next thread to run using *one* request to the progress manager.

5. Experimental Results

In order to experimentally evaluate the techniques described in this paper, a prototype implementation of the system has been built. This effort is a part of a larger comprehensive system development effort that is expected

to yield a fully functional implementation. The current implementation runs annotated C++ programs, that is the parallel program is written in C++ with embedded constructs to express parallelism, along the lines described above.

The prototype consists of several (one or more) workstations designated as “compute servers”. At present there is one machine dedicated to run as a combination of the “memory servers” and the “progress managers” and it is called the “manager”. Failures of compute servers are tolerated but the failure of the “manager” is not currently tolerated. Thus the evasion/dispersal algorithms are not necessary (the central site discards late writes) and have not been incorporated.

We present the results of one experiment that performs a parallel sort on a large (128KByte) array [Bar94]. There were a total of 6 execution: (1) A truly, sequential execution on a single machine--the parallel program was modified into a sequential program that there is no overhead due to parallelism; and (2-6) Five parallel execution using the manager, and respectively between 1 and 5 compute servers.. The manager and the compute servers were 6 identical diskless Sun Sparcstation SLC's.

The results are listed in the table in Fig. 5. Each execution has 3 blocks, block 1 random fills the array sequentially, block 2 sorts different segments of the array in parallel and block 3 merges the sorted segments sequentially. The row marked 0 compute servers is the sequential execution. The efficiency and speedup computation is for the parallel step (block 2). Efficiency is calculated as the sequential time divided by the total parallel time multiplied by the number of compute servers.

It is not surprising that the efficiency is not monotonic. Executions in which the number of compute servers does not divide the number of threads (15), are less efficient, as the unit of assignment is a thread.

6. Conclusions

We present a design of a runtime environment that provides a parallel processing environment supporting fault tolerance and automatic load balancing and runs on a network of machines. The overhead of the system is low due to

Compute Servers	Block 1 time	Block 2 time	Block 3 time	Speedup (Block 2)	Efficiency
0	0.60	115.9	0.92	n/a	100.0%
1	0.81	119.5	0.99	0.97	97.0%
2	0.61	64.6	0.96	1.79	89.6%
3	0.65	41.2	0.91	2.81	93.8%
4	0.63	33.7	0.91	3.44	86.1%
5	0.67	25.6	0.96	4.53	90.7%

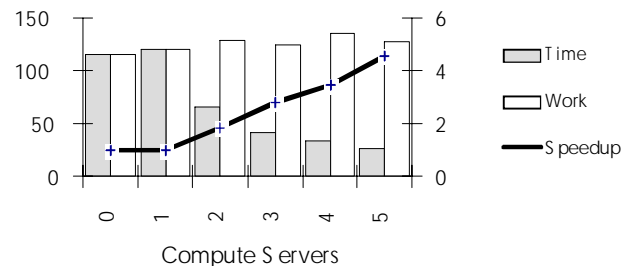


Fig. 5: The table of sort timings and the corresponding graph (all times in seconds)

the use of some novel ideas that allow us to avoid replication, synchronization and costly management schemes. The system is in the prototyping phase and preliminary results look promising.

We are aware that there is considerable more work to be done, especially in the area of optimizing the scheduling, managing the memory architecture better and handling of the data structures. Also, we are working on extending the scheme to support computations that do I/O and have non-deterministic executions. Preliminary results exist but are outside the scope of this paper.

7. Bibliography

- [AKPR93]
Y. Aumann, Z. Kedem, K. Palem, and M. Rabin. Highly Efficient Asynchronous Execution of Large-Grained Parallel Programs. In *34th IEEE Ann. Symp. on Foundations of Computer Science*, pages 271–280, 1993.
- [AR93]
Y. Aumann and M. Rabin. Clock Construction in Fully Asynchronous Parallel Systems and PRAM simulation. In *33rd IEEE Ann. Symp. on Foundations of Computer Science*, 1993, pages 147–156.
- [Bar94]
A. Baratloo. Performance Experiments with a Fault-Tolerant Parallel Processing System. unpublished manuscript, 1994.
- [Bir93]
K. Birman. The Process Group Approach To Reliable Distributed Computing. Technical report, Cornell University, 1993.
- [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. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158. ACM, 1989.
- [CG89]
N. Carriero and D. Gelernter. Linda in Context. *Communication of ACM*, 32(4):444–458, 1989.
- [CLG+94]
P. Chen, E. Lee, G. Gibson, R. Katz, and D. Patterson: RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, Volume 26, Number 2, June 1994, pp. 145–186.
- [Coo85]
E. Cooper. Replicated distributed programs. *Operating Systems Review*, 19(5):63–78, December 1985.
- [DAM+91]
P. Dasgupta, R. Ananthanarayanan, S. Menon, A. Mohindra, and R. Chen. Distributed Programming with Objects and Threads in the Clouds System. *Computing Systems*, 4(3):243–276, 1991.
- [DCM+90]
P. Dasgupta, R. C. Chen, S. Menon, M. P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. J. LeBlanc, W. F. Appelbe, J. M. Bernabeu-Auban, P. W. Hutto, M. Y. A. Khalidi, and C. J. Wilkenloh. The Design and Implementation of the Clouds Distributed Operating System. *Computing Systems*, 3, 1990.
- [DLAR90]
P. Dasgupta, R. LeBlanc Jr., M. Ahamad, and U. Ramachandran. The Clouds Distributed Operating System. *IEEE Computer*, 1990.
- [FD93]
M. Fu and P. Dasgupta. A Concurrent Programming Environment for Memory-Mapped Persistent Objects. *the 17th Intl. Computer Software and Application Conference (COMPSAC-93)*.
- [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.
- [Ked92]
Z. Kedem. Methods for Handling Faults and Asynchrony in Parallel Computations. In *1992 DARPA Software Technology Conference*, pages 189–193, May 1992.
- [KP92]
Z. Kedem and K. Palem. Transformations for the Automatic Derivation of Resilient Parallel Programs. In *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 15–25, 1992.
- [KPRR92]
Z. Kedem, K. Palem, M. Rabin, and A. Raghunathan. Efficient Program Transformations for Resilient Parallel Computation via Randomization. In *24th ACM Symp. on Theory of Computing*, pages 306–317, May 1992.
- [KPRS91]
Z. Kedem, K. Palem, A. Raghunathan, and P. Spirakis. Combining Tentative and Definite Algorithms For Very Fast Dependable Parallel Computing. In *23rd ACM Symp. on Theory of Computing*, pages 381–390, May 1991.
- [KPRS93]
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.
- [KPS90]
Z. Kedem, K. Palem, and P. Spirakis. Efficient Robust Parallel Computations. In *22nd ACM Symp. on Theory of Computing*, pages 138–148, May 1990.
- [LH89]
K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.
- [MDRK92]
B. Millard, P. Dasgupta, S. Rao, R. Kuramkote. Run-Time Support and Storage Management for Memory mapped Persistent Stores. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, 1992.
- [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.
- [PWC+81]
G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. Locus: A Network Transparent, High Reliability Distributed System. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 169–177. ACM, 1981.
- [Ra81]
M. Rabin. Fingerprinting by Random Polynomials, *Technical Report, Harvard University*, 1981.
- [Rab89]
M. Rabin. Efficient Dispersal of Information for Security, Load Balancing and Fault Tolerance. *J. ACM*, 36:335–348, 1989.
- [Sun90]
V.S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, 1990.