Metacomputing with MILAN

A. Baratloo   P. Dasgupta[*]   V. Karamcheti   Z.M. Kedem
{baratloo,dasgupta,vijayk,kedem}@cs.nyu.edu
Courant Institute of Mathematical Sciences, New York University

Abstract

The MILAN project, a joint effort involving Arizona State University and New York University, has produced and validated fundamental techniques for the realization of efficient, reliable, predictable virtual machines on top of metacomputing environments that consist of an unreliable and dynamically changing set of machines. In addition to the techniques, the principal outcomes of the project include three parallel programming systems—Calypso, Chime, and Charlotte—which enable applications be developed for ideal, shared memory, parallel machines to execute on distributed platforms that are subject to failures, slowdowns, and changing resource availability. The lessons learnt from the MILAN project are being used to design Computing Communities, a metacomputing framework for general computations.

1. Motivation

MILAN (**M**etacomputing **I**n **L**arge **A**synchronous **N**etworks) is a joint project of Arizona State University and New York University. The primary objective of the MILAN project is to provide middleware layers that would enable the efficient, predictable execution of applications on an unreliable and dynamically changing set of machines. Such a middleware layer, will in effect create a *metacomputer*, that is a reliable stable platform for the execution of applications.

Improvements in networking hardware, communication software, distributed shared memory techniques, programming languages and their implementations have made it feasible to employ distributed collections of computers for executing a wide range of parallel applications. These "metacomputing environments," built from commodity machine nodes and connected using commodity interconnects, afford significant cost advantages in addition to their widespread availability (e.g., a machine on every desktop in an organization). However, such environments also present unique challenges for constructing metacomputers on them, because the component machines and networks may: (1) exhibit wide variations in performance and capacity, (2) become unavailable either partially or completely because of their use for other (non-metacomputing related) tasks. These challenges force parallel applications running on metacomputers to deal with an unreliable, dynamically changing set of machines and have thus, limited their use on all but the most decoupled of parallel computations.

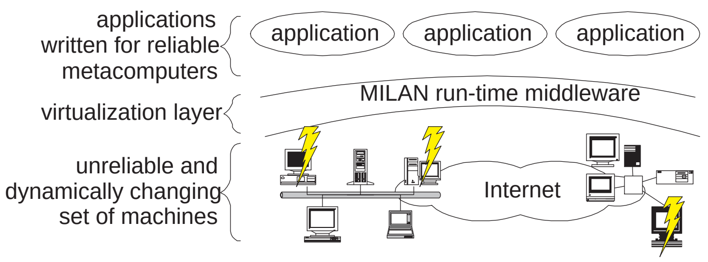
[*]On leave from Arizona State University

As part of the MILAN project, we have been investigating fundamental techniques which would enable the effective use of metacomputing environments for a wide class of applications, originally concentrating on parallel ones. The key thrust of the project has been to develop run-time middleware that builds an *efficient, predictable, reliable virtual machine model* on top of unreliable and dynamically changing platforms. Such a virtual machine model would enable applications developed for idealized, reliable, homogeneous parallel machines to run unchanged on unreliable, heterogeneous metacomputing environments. Figure 1 shows the MILAN middleware in context. Our approach for realizing the virtual machine takes advantage of two general characteristics of computation behavior: *adaptivity* and *tunability*.

Adaptivity refers to a flexibility in execution. Specifically, a computation is adaptive if it exhibits at least one of these two properties: (1) it can statically (at start time) and/or dynamically (during the execution) ask for resources satisfying certain characteristics and incorporate such resources when they are given to it, and (2) it can continue executing even when some resources are taken away from it.

Tunability refers to a flexibility in specification. Specifically, a computation is tunable if it is able to trade off resource requirements over its lifetime, compensating for a smaller allocation of resources in one stage with a larger allocation in another stage and/or a change in the quality of output produced by the computation.

Our techniques leverage this flexibility in execution and specification to provide reliability, load balancing, and predictability, even when the underlying set of machines is unreliable and changing dynamically.



applications written for reliable metacomputers
application
application
application
virtualization layer
MILAN run-time middleware
unreliable and dynamically changing set of machines
Internet

The principal outcomes of the MILAN project are (1) a core set of fundamental resource management techniques [21, 3, 23, 9] enabling construction of efficient, reliable, predictable virtual machines, and (2) the realization of these techniques in three complete programming systems: *Calypso* [3], *Chime* [27], and *Charlotte* [6]. Calypso extends C++ with parallel steps interleaved into a sequential program. Each parallel step specifies the independent execution of multiple concurrent tasks or a family of such tasks. Chime extends Calypso to provide nested parallel steps and inter-thread communication primitives (as expressed in the shared memory parallel language, Compositional C++ [8]). Charlotte provides a Calypso-like programming system and runtime environment for the Web. In addition to these systems, the MILAN project has also produced two general tools: *ResourceBroker* [4] and *KnittingFactory* [5], which support resource discovery and integration in distributed and web-based environments, respectively. More recently, as part of the *Computing Communities* project, we have been examining how the experience gained from designing, implementing, and evaluating these systems can be extended to supporting general applications on metacomputing environments.

The rest of the paper is organized as follows. Section 2 overviews the fundamental techniques central to all of the MILAN project's programming systems. The design, implementation, and performance of the various programming systems and general tools is described in detail in Section 3. Finally, Section 4 presents the rationale and preliminary design of Computing Communities, a metacomputing framework for general computations.

2 Key Techniques

To execute parallel programs on networks of commodity machines, one frequently assumes *a priori* knowledge—at program development—of the number, relative speeds, and the reliability of the machines involved in the computation. Using this information, the program can then distribute its load evenly for efficient execution. This knowledge can not be assumed for distributed multiuser environments, and hence, it is imperative that programs adapt to machine availability. That is, a program running on a metacomputer must be able to integrate new machines into a running computation, mask and remove failed machines, and balance the work load in such a way that slow machines do not dictate the progress of the computation.

The traditional solution to overcome this type of dynamically changing environment has been to write *self-scheduling* parallel programs (also referred to as the master/slave [16], the manager/worker [17], or the bag-of-tasks [7] programming model). In self-scheduled programs, the computation is divided into a large number of small computational units, or tasks. Participating machines pick up and execute a task, one at a time, until all tasks are done, enabling the computation to progress at a rate proportional to available resources. However, self scheduling does not solve all the problems associated with executing programs on distributed multiuser environments. First, self scheduling does not address machine and network failures. Second, a very slow machine can slow down the progress of faster machines if it picks up a compute-intensive task. Finally, self scheduling increases the number of tasks comprising a computation and, thereby, increases the effects of the overhead associated with assigning tasks to machines. Depending on the network, this overhead may be large and, in many cases, unpredictable.

The MILAN project extends the basic self-scheduling scheme in various ways to adequately address the above shortcomings. These extensions are embodied in five techniques: *eager scheduling*, *two-phase idempotent execution strategy* (TIES), *dynamic granularity management*, *preemptive scheduling*, and *predictable scheduling* for tunable computations. We describe the principal ideas behind each of these techniques here, deferring a detailed discussion of their implementation and impact on performance to the description of various programming systems in Section 3.

Eager scheduling extends self scheduling to deal with network and machine failures, as well as any disparity in machine speeds. The key idea behind eager scheduling, initially proposed in [21], is that a single computation task can be concurrently worked upon by multiple machines. Eager scheduling works in a manner similar to self scheduling at the beginning of a parallel step, but once the number of remaining tasks goes below the number of available machines, eager scheduling aggressively assigns and re-assigns tasks until all tasks have been executed to completion. Concurrent assignment of tasks to multiple machines guarantees that slow machines, even very slow machines, do not slow down the computation. Furthermore, by considering failure as a special case of a slow machine (an infinitely slow machine), even if machines crash or become less accessible, for example due to network delays, the entire computation will finish as long as at least one machine is available for a sufficiently long period of time. Thus, eager scheduling masks machine failures without the need to actually detect failures.

Multiple executions of a program fragment (which is possible under eager scheduling) can result in an incorrect program state. TIES [21] ensures *idempotent memory semantics* in the presence of multiple executions. The computation of each parallel step is divided into two phases. In the first phase, modifications to the shared data region, that is the write-set of tasks, are computed but kept aside in a buffer. The second phase begins when all tasks have executed to completion. Then, a single write-set for each completed task is applied to the shared data, thus atomically updating the memory. Note that each phase is idempotent, since its inputs and outputs are disjoint. Informally, in the first phase the input is shared data and the output is the buffer, and in the second phase the input is the buffer and the output is shared memory.

The interplay of eager scheduling and TIES addresses fault masking and load balancing. *Dynamic granularity management* (*bunching* for short) is used to amortize overheads and mask network latencies associated with the process of assigning tasks to machines. Bunching extends self scheduling by assigning a set of tasks (a bunch) as "a single assignment." Bunching has three benefits. First, it reduces the number of task assignments, and hence, the associated overhead. Second, it overlaps computation with communication by allowing machines to execute the next task (of a bunch) while the results of the previous task are being sent back on the network. Finally, bunching allows the programmer to write fine-grained parallel programs that are automatically and transparently executed in a coarse-grained manner.

We have implemented *factoring* [19], an algorithm that computes the bunch size based on the number of remaining tasks and the number of currently available machines.

Eager scheduling provides load balancing and fault isolation in a dynamic environment. However, our description so far has considered only non-preemptive tasks which run to completion once assigned to a worker. Non-preemptive scheduling has the disadvantage of delivering sub-optimal performance when there is a mismatch between the set of tasks and the set of machines. Examples of situations include when the number of tasks is not divisible by the number of machines, when the tasks are of unequal lengths, and when the number of tasks is not static (i.e., new tasks are created and/or terminated on the fly). To address inefficiencies resulting from these situations, the MILAN project complements eager scheduling with preemptive scheduling techniques. Our results, discussed in Section 3, show that despite preemption overheads, use of preemptive scheduling on distributed platforms can improve execution time of parallel programs by reducing the number of tasks that need to be repeatedly executed by eager scheduling [23].

We have developed a family of preemptive algorithms, of which we present three here. The *Optimal Algorithm* is targeted for situations where the number of tasks to be executed is slightly larger than the number of machines available. This algorithm precomputes a schedule that minimizes the execution time and the number of context switches needed. However it requires that the task execution time be known in advance and therefore is not always practical. The *Distributed, Fault-tolerant Round Robin Algorithm* is suited for a set of n tasks scheduled on m machines, where $n > m$. Initially, the first m tasks are assigned to the m machines. Then, after a specified time quantum, all the tasks are preempted and the next m tasks are assigned. This continues in a circular fashion until all tasks are completed. The *Preemptive Task Bunching Algorithm* is applicable over a wider range of circumstances. All n tasks are bunched into m bunches and assigned to the m machines. When a machine finishes its assigned bunch, all the tasks on all the other machines are pre-empted and all the remaining tasks are collected, re-bunched (into m sets), and assigned again. This algorithm works well for both large-grained and fine-grained tasks even when machine speeds and task lengths vary.

While the techniques described earlier enable the building of an efficient, fault-tolerant virtual machine on top of an unreliable and dynamically changing set of machines, they alone are unable to address the predictability requirements of applications such as image recognition, virtual reality, and media processing that are increasingly running on metacomputers. One of the key challenges deals with providing sufficient resources to computations to enable them to meet their time deadlines in the face of changing resource availability.

Our technique [9] relies upon an explicit specification of application *tunability*, which refers to an application's ability to absorb and relinquish resources during its lifetime, possibly trading off resource requirements versus quality of its output. Tunability provides the freedom of choosing amongst multiple execution paths, each with their own resource allocation profile. Given such a specification and short-term knowledge about the availability of resources, the MILAN resource manager chooses an appropriate execution path in the computation that would allow the computation to meet its predictability requirements. In general, the resource manager will need to renegotiate both the level of resource allocation and the choice of execution path in response to changes in resource characteristics. Thus, application tunability increases its likelihood of achieving predictable behavior in a dynamic environment.

3. Programming Systems

Commercial realities dictate that parallel computations typically will not be given a dedicated set of identical machines. Non-dedicated computing platforms suffer from non-uniform processing speeds, unpredictable behavior, and transient availability. These characteristics result from external factors that exist in "real" networks of machines. Unfortunately, load balancing, fault masking, and adaptive execution of programs on a set of dynamically changing machines are neglected by most programming systems. The neglect of these issues has complicated the already difficult job of developing parallel programs.

Calypso [3] is a parallel programming system and a runtime system designed for adaptive parallel computing on networks of machines. The work on Calypso has resulted in several original contributions which are summarized below.

Calypso separates the programming model from the execution environment: programs are written for a reliable virtual shared-memory computer with unbounded number of processors, i.e., a metacomputer, but execute on a network of dynamically changing machines. This presents the programmer with the illusion of a reliable machine for program development and verification. Furthermore, the separation allows programs to be parallelized based on the inherent properties of the problem they solve, rather than the execution environment.

Programs without any modifications can execute on a single machine, a multiprocessor, or a network of unreliable machines. The Calypso runtime system is able to adapt executing programs to use available resources—computations can dynamically scale up or down as machines become available, or unavailable. It uses TIES and allows parts of a computation executing on remote machines to fail, and possibly recover, at any point without affecting the correctness of the computation. Unlike other fault-tolerant systems, there is no significant additional overhead associated with this feature.

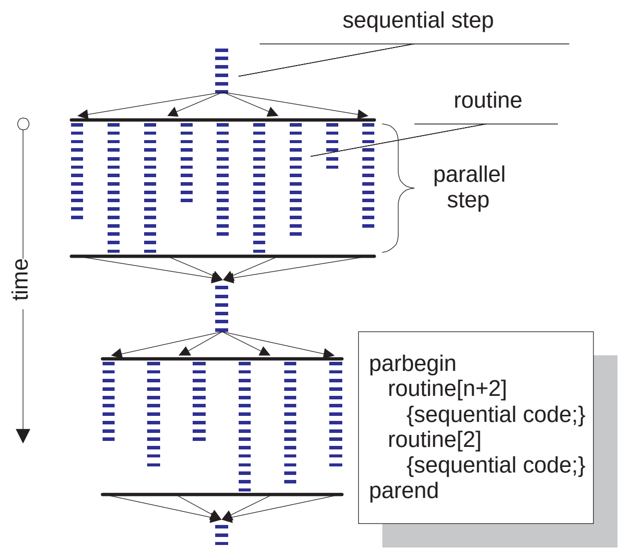
Calypso automatically distributes the work-load depending on the dynamics of participating machines, using eager scheduling and bunching. The result is that fine-grain computations are efficiently executed in coarse-grain fashion, and faster machines perform more of the computation than slower machines. Not only is there no cost associated with this feature, but it actually speeds up the computation, because fast machines are never blocked while waiting for slower machines to finish their work assignments—they bypass the slower machines. As a consequence, the use of slow machines will never be detrimental to the performance of a parallel program.

3.1.1 Calypso Programs

A Calypso program basically consists of the standard C++ programming language, augmented by four additional keywords to express parallelism. Parallelism is obtained by embedding *parallel steps* within sequential programs. Parallel steps consist of one or more task (referred to as *jobs* in the Calypso context), which (logically) execute in parallel and are generally responsible for computationally intensive segments of the program. The sequential parts of programs are referred to as *sequential steps* and they generally perform initialization, input/output, user interactions, etc.

Figure 2 illustrates the execution of a program with two parallel steps and three sequential steps. It is important to note that parallel programs are written for a virtual shared-memory parallel machine irrespective of the number of machines that participate in a given execution.

This programming model is sometimes referred to as a block-structured parbegin/parend or fork/join model [13, 25]. Unlike other programming models where programs are decomposed (into several files or functions) for parallel execution, this model together with shared memory semantics, allows loop-level parallelization. As a result, given a working sequential program it is fairly straightforward to parallelize individual independent loops in an incremental fashion—if the semantics allows this.



Shared-memory semantics is only provided for *shared variables*, i.e., variables that are tagged with the `shared` keyword. A parallel step starts with the keyword `parbegin` and ends with the keyword `parend`. Within a parallel step, multiple parallel *jobs* may be defined using the keyword `routine`. Completion of a parallel step consists of completion of all its jobs in an indeterminate order.

3.1.2 Execution Overview

A typical execution of a Calypso program consists of a central process, called the *manager*, and one or more additional processes, called *workers*. These processes can reside on a single machine or they can be distributed on a network. In particular, when a user starts a Calypso program, in reality, she is starting a manager. Managers immediately fork a child process that executes as a worker.

The manager is responsible for the management of the computation as well as the execution of sequential steps. The current Calypso implementation only allows one manager, and therefore it does not tolerate the failure of this process. The computation of parallel jobs is left to the workers. In general, the number of workers and the resources they can devote to parallel computations can dynamically change in a completely arbitrary manner, and the program adapts to the available machines. In fact, the arbitrary slowdown of workers due to other executing programs on the same machine, failures due to process and machine crashes, and network inaccessibility due to network partitions are tolerated. Furthermore, workers can be added at any time to speed up an already executing system and to increase fault tolerance. Arbitrary slowdown of the manager is also tolerated; this would, of course, slow down the overall execution though.

3.1.3 Manager Process

The manager is responsible executing the non-parallel step of a computation as well as providing workers with *scheduling* and *memory* services.

Scheduling Service: Jobs are assigned to workers based on a self-scheduling policy. Moreover, the manager has the option of assigning a job repeatedly until it is executed to completion by at least one worker—this is eager scheduling, and provides the following benefits:

- As long as at least one worker does not fail continually, all jobs will be completed, if necessary, by this one worker.
- jobs assigned to workers that later failed are automatically reassigned to other workers; thus crash and network failures are tolerated.
- Because workers on fast machines can re-execute jobs that were assigned to slow machines, they can bypass a slow worker to avoid delaying the progress of the program.

In addition to eager scheduling, Calypso's scheduling service implements several other scheduling techniques for improved performance. Bunching masks network latencies associated with the process of assigning jobs to workers.It is implemented by sending the worker a range of job Ids in each assignment. The overhead associated with this implementation is one extra integer value per job assignment message, which is negligible.

Memory Service: Since multiple executions of jobs caused by eager scheduling may lead to an inconsistent memory state, managers implements TIES as follows. Before each parallel step, a manager creates a twin copy of the shared pages and unprotects the shared region. The memory management service then waits until a worker either requests a page or reports the completion of a job. The manager uses the twin copy of the shared pages to service worker page requests. The message that workers send to the manager to report the completion of a job also contains the modifications that resulted from executing the job. Specifically, workers logically bit-wise `xor`s the modified shared pages before and after executing the job, and send the results (diffs) to the manager. When a manager receives such a message, it first checks whether the job has been completed by another worker. If so, the diffs are discarded, otherwise, the diffs are applied (by an `xor` operation) to manager's memory space. Notice that the twin copies of the shared pages, which are used to service worker page requests, are not modified. The memory management of a parallel step halts once all the jobs have run to completion, and the program execution then continues with the next sequential step.

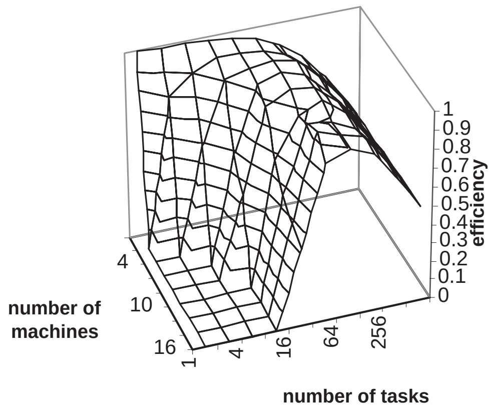
3.1.4 Worker Process

A worker repeatedly contacts the manager for jobs to execute. The manager sends the worker an assignment (a bunch of jobs) specified by the following parameters: the address of the function, the number of instances of the job, and a range of job Ids. After receiving a work assignment, a worker first access-protects the shared pages, and then calls the function that represents an assigned job. The worker handles page-faults by fetching the appropriate page from the manager, installing process' address space, and unprotecting the page so that subsequent accesses to the same page will proceed undisturbed. Once the execution of the function (i.e. the job) completes, the worker identifies all the modified shared pages and sends the diffs to the manager and starts executing the next job in the assignment. Notice that bunching overlaps computation with communication by allowing a worker to execute the next job while the diffs are on the network heading to the manager.

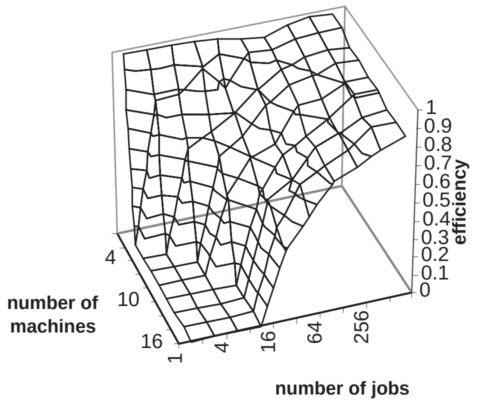
Additional optimizations have been implemented, including the following:

Caching: For each shared page, the manager keeps track of the logical step-number in which the page was last modified. This vector is is piggybacked on a job assignment the first time a worker is assigned a job in a new parallel step. Hence, the associated network overhead is negligible. Workers can use this vector on page-faults to *locally determine* whether the cached copy o the page is still valid. Thus, shared pages that have paged-in by workers are kept valid as long as possible without a need for an invalidation protocol. Modified shared pages are re-fetched only when necessary. Furthermore, read-only shared pages are fetched by a worker at most once and write-only shared pages are never fetched. As a result, programmer does not declare the type of coherence or caching technique to use, rather, the system dynamically adapts. Invalidation requests are piggybacked on work assignment messages and bear very little additional cost.

Prefetching: Prefetching refers to obtaining a portion of the data before it is needed, in the hope that it will be required sometime in the future. Prefetching has been used in a variety of systems with positive results. A Calypso worker implements prefetching by monitoring its own data access patterns and page-faults, and it tries to predict future data access based on past history. The predictions are then used to pre-request shared pages from the manager. Depending on the regularity of a program's data access patterns, prefetching has shown positive results.



(a) PVM



(b) Calypso

3.1.5 Performance Experiments

The experiments were conducted on up to 17 identical 200 MHz PentiumPro machines running Linux version 2.0.34 operating system, and connected by a 100Mbps Ethernet through a non-switched hub. The network was isolated to eliminate outside effects.

A publicly available sequential ray tracing program [10] was used as the starting point to implement parallel versions in Calypso and PVM [16]. The sequential program, which traced a 512×512 image in in 53 s, is used for calculating the parallel efficiencies.

The PVM implementation used explicit master/slave programming style for load balancing, where as for Calypso, load balancing was provided transparently by the run-time system. To demonstrate the effects of adaptivity, the PVM and Calypso programs were parallelized using different number of tasks and executed from 1 to 16 machines. The performance results are illustrated in Figure 3. As the results indicate, the PVM program is very sensitive to the number and the computation requirement of the parallel tasks, and at most, a hand-tuned PVM program outperforms a Calypso program by 4%. Notice that independent of the number of machines used, the interplay of bunching, eager scheduling, and TIES allows the Calypso program to achieves its peak performance using 512 tasks—fine grain tasks: as the result of bunching, fine-grain tasks, in effect, execute in coarse-grain fashion; the combination of eager scheduling and TIES compensates any over-bunching that may occur.

Chime is a parallel processing system that retains the salient features of Calypso, but supports a far richer set of programming features. The internals of Chime are significantly different from Calypso, and it runs on the Windows NT operating system [27]. Chime is the first system that provides a true general shared memory multiprocessor environment on a network of machines. It achieves this by implementing the CC++ [8] language (shared memory part) on a distributed system. Thus in addition to Calypso features of fault-tolerance and load balancing Chime provides:

- True multi-processor shared-memory semantics on a network of machines.
- Block structured scoping of variables and non-isolated distributed parallel execution.
- Support for nested parallelism.
- Inter-task synchronization.

3.2.1 Chime Architecture

A program written in CC+ is preprocessed to convert it to C++ and compiled and linked with the Chime library. Then the executable is run, using the manager-worker scheme of Calypso.

The manager process consists of two threads, the *application thread* and the *control thread*. The application thread executes the code programmed by the programmer. The control thread executes, exclusively, the code provided by the Chime library. Hence, the application thread runs the program and the control thread runs the management routines, such as scheduling, memory service, stack management, and synchronization handling.

The worker process also consists of two threads, the application thread and the control thread. The application threads in the worker and manager are identical. However, the control thread in the worker is the client of the control thread in the manager. It requests work from the manager, retrieves data pages from the manager and flushes updated memory to the manager at the end of the task execution.

3.2.2 Chime and CC++

As mentioned earlier, Chime provides a programming interface that is based on the Compositional C++ or CC++ [8] language definition. CC++ provides language constructs for shared memory, nested parallelism and synchronization. All threads of the parallel computation share all global variables. Variables declared local to a function are private to the thread running the function, but if this thread creates more threads inside the function, then all the children share the local variables.

CC++ uses the *par* and *parfor* statements to express parallelism. Par and parfor statements can be nested. CC++ uses single assignment variables for synchronization. A single assignment variable is assigned a value by any thread called the writing thread. Any other thread, called the reading thread can read the written value. The constraint is that the writing thread has to assign before the reading thread reads, else the reading thread is blocked until the writing thread assigns the variable.

These language constructs provide significant challenges to a distributed (DSM-based) implementation that is also fault tolerant. We achieved the implementation by using a pre-processor to detect the shared variables and parallel constructs, providing stack-sharing support—called distributed cactus stacks—to implement parent-child variable sharing and innovative scheduling techniques, coupled with appropriate memory flushing to provide synchronization [28].

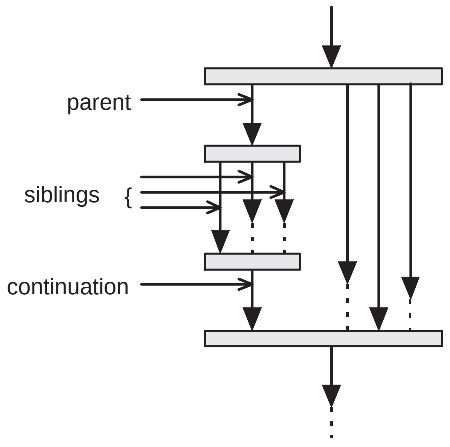
3.2.3 Preprocessing CC++

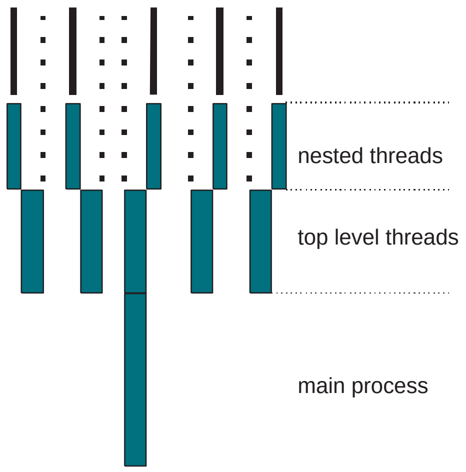
Consider the following parallel statement:

```
parfor ( int i=0; i<100; i++) {
    a[i] = 0;
};
```

This creates 100 tasks, each task assigning one element of the array `a`. The preprocessor converts the above statement to something along the following lines:

```
1.   for (int i=0; i<100;i++) {
2.      add task entry and &i
        in the scheduling table;
    }
3.  SaveContext of this thread;
```





```
4.   if worker {
        a[i] = 0;
5.      terminate task;
     }
6.   else {
7.      suspend this thread and
        request manager to
        schedule threads till
        all tasks completed;
     }
```

The above code may execute in the manager (top level parallelism) or the worker (nested parallelism). *Assume the above code executes in the manager*. Then the application thread of the manager executes the code. Lines 1 and 2 create 100 entries in the scheduling table, one per parallel task. Then line 3 saves the context of the parent task, including the parent stack. Then the parent moves to line 7 and this causes the application thread to transfer control to the control thread.

The control thread now waits for task assignment requests from the control threads of workers. When a worker requests a task, the manager control thread sends the stored context and the index value of i for a particular task to the worker.

The control thread in the worker installs the received context and the stack on the application thread in the worker and resumes the application thread. This thread now starts executing at line 4. Note that now the worker is executing at line 4, and hence does one iteration of the loop and terminates. Upon termination, the worker control thread regains control, flushes the updated memory to the manager and asks the manger for a new assignment.

3.2.4 Scheduling

The controlling thread at the manager is also responsible for task assignment, or scheduling. The manager uses a scheduling algorithm that takes care of task allocation to the workers as well as scheduling of nested parallel tasks in correct order. Nested parallel tasks in an application form a DAG as shown in Figure 4.

Each nested parallel step consists of several *sibling* parallel tasks. It also has a parent task and a continuation that must be executed, once the nested parallel step has been completed. A continuation is an object that fully describes a future computation. To complicate the scenario, a continuation may itself have nested parallel step(s).

The manager maintains an execution dependency graph to capture the dependencies between the parallel tasks and schedules them and their corresponding continuations in correct order. Eager scheduling is used to allocate tasks to the workers.

3.2.5 Cactus Stacks

The cactus stacks are used to handle sharing of local variables (see Figure 5). For top level nesting, the manager process is suspended at a point in execution where its stack and context should be inherited by all the children threads. When a worker starts, it is sent the contents of the manager's stack along with the context. The controlling thread of the worker process then installs this context as well as the stack, and starts the application thread.

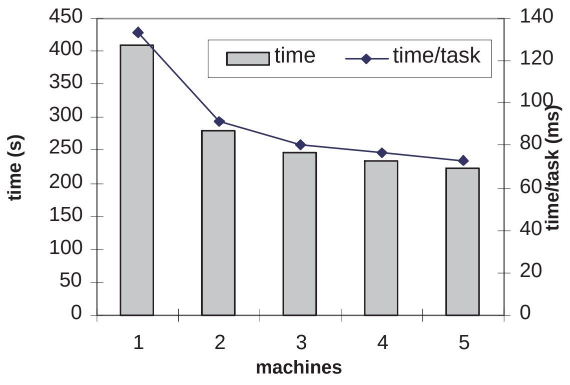
However, if a worker executes a nested parallel step, the same code as the above case is used, but the runtime system behaves slightly differently. The worker, after generating the nested parallel jobs, invokes a routine that adds the jobs and the continuation of the parent job to the manager's job table, remotely. The worker suspends and the controlling thread in the worker, sends the worker's complete context, including the newly grown stack, to the manager.

The stack for a nested parallel task, therefore, is constructed by writing the stack segments of its ancestors onto the stack of a worker's application thread. Upon completion, the local portion of the stack for a nested parallel task is unwound leaving only those portions that represent its ancestors. This portion of the stack is then `xor`'ed with its unmodified shadow and the result is returned to the manager.

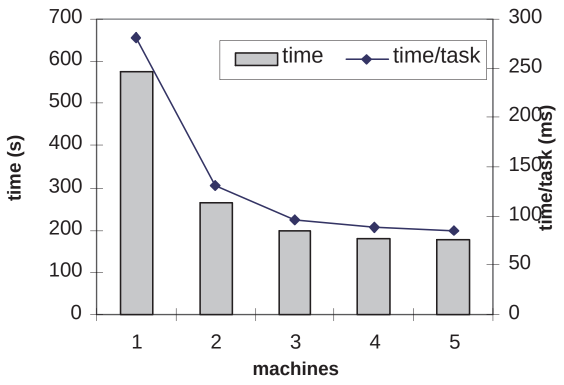
3.2.6 Performance Experiments

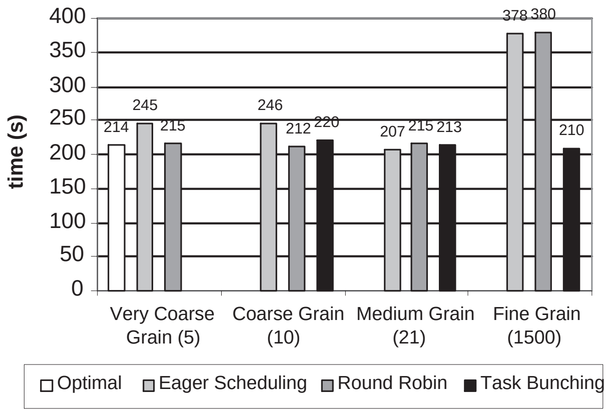
Many performance tests have been done on Chime [27], evaluating its capabilities in speedups, load balancing, and fault tolerance. The results are competitive to other systems, including Calypso. We present three micro-tests that show the performance of the nested parallelism (including cactus stacks), the Chime synchronization mechanisms, and preemptive scheduling mechanisms.

For the nested parallelism overhead, we run a program that recursively creates two child threads until 1024 leaf threads have been created. Each leaf thread assigns one integer in a shared array and then terminates. Figure 6 shows that the total runtime of the program asymptotically saturates as number of machines are increased, due to the bottleneck in stack and thread management at the manager. The time taken to handle all overhead for a thread (including cactus stacks) is 74 ms.



To measure the synchronization overhead, we use 512 single assignment variables, assign them from 512 threads and read them from 512 other threads. As can be seen in Figure 7, the synchronization overhead is about 86 ms per occurrence, showing that synchronization does not add too much overhead over basic thread creation.





To measure the impact of preemptive scheduling algorithms for programs with different grain sizes, we decomposed a matrix-multiply algorithm on two 1500×1500 matrices into 5 tasks (very coarse grain), 10 tasks (coarse grain), 21 tasks (medium grain), and 1500 tasks (fine grain). All experiments used three identical machines. Given the equal task lengths, our experiments were biased against preemptive schedulers. As shown in Figure 8, on the overall, preemptive scheduling has definite advantages over non-preemptive scheduling, not withstanding its additional overheads. Specifically, for coarse-grained and very coarse-grained tasks, round robin scheduling effectively complements eager scheduling in reducing overall execution time. For most other task sizes, the preemptive task bunching algorithm yields the best performance; for fine-grained tasks it minimizes the number of preemptions that are necessary.

Many of the assumptions made for (local-area) networks of machines are not valid for the Web. For example, the machines on the Web do not have a common shared file system, no single individual has access-rights (user-account) on every machine, and the machines are not homogeneous. Another important distinction is the concept of *users*. A user who wants to execute a program on a network of machines, typically performs the steps: logs onto a machine under her control (i.e. the *local* machine), from the local machine logs onto other machines on the network (i.e. *remote* machines) and initializes the execution environment, and then starts the program. In the case of the Web, no user can possibly hope to have the ability to log onto remote machines. Thus, another set of users who control remote machines, or software agents acting on their behalf, must voluntarily allow others access. To distinguish the two types of users, this section uses the term *end-users* to refer to individuals who start the execution (on their local machines) and await results, and *volunteers* to refer to individuals who voluntarily run parts of end-users' programs on their machines (remote to end-users). Similarly, *volunteer machines* is used to refer to machines owned by volunteers.

Simplicity and security are important objectives for volunteers. Unless the process of volunteering a machine is simple—for example as simple as a single mouse-click—and the process of withdrawing a machine is simple, it is likely that many would-be volunteer machines will be left idle. Furthermore, volunteers need assurance that the integrity of their machine and file system will not be compromised by allowing "strangers" to execute computations on their machines. Without such an assurance, it is natural to assume security concerns will outweigh the charitable willingness volunteering.

Charlotte [6] is the *first parallel programming system to provide one-click computing*. The idea behind one click computing is to allow volunteers from anywhere on the Web, and without any administrative effort, to participate in ongoing computations by simply directing a standard Java-capable browser to a Web site. A key ingredient in one-click computing is its lack of requirements: user-accounts are not required, the availability of the program on a volunteer's machine is not assumed, and system-administration is not required. Charlotte builds on the capability of the growing number of Web browsers to seamlessly load Java applets from remote sites, and the applet security model, which enables Web browsers to execute untrusted applets in a controlled environment, to provide a comprehensive programming system.

3.3.1 Charlotte Programs

A Charlotte program is written by inserting any number of *parallel steps* onto a sequential Java program. A parallel step is composed of one or more *routines*, which are (sequential) threads of control capable of executing on remote machines.

A parallel step starts and ends with the invocation of `parBegin()` and `parEnd()` methods, respectively. A routine is written by subclassing the `Droutine` class and overriding its `drum()` method. Routines are specified by invoking the `addRoutine()` method with two arguments: a routine object and an integer, n, representing the number of routine instances to execute. To execute a routine, the Charlotte runtime system invokes the `drun()` method of routine objects, and passes as arguments the number of routine instances created (i.e. n) and an identifier in the range $(0, \ldots, n]$ representing the current instance.

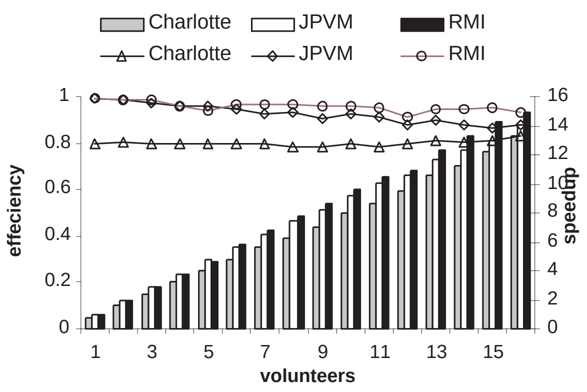
A program's data is logically partitioned into *private* and *shared* segments. Private data is local to a routine and is not visible to other routines; shared data, which consists of *shared class-type* objects, is distributed and is visible to all routines. For every basic data-type defined in Java, Charlotte implements a corresponding distributed shared class-type. For example, Java provides `int` and `float` data-types, whereas Charlotte provides `Dint` and `Dfloat` classes. The class-types are implemented as standard Java classes, and are read and written by invoking `get()` and `set()` method calls, respectively. The runtime system maintains the coherence of shared data.

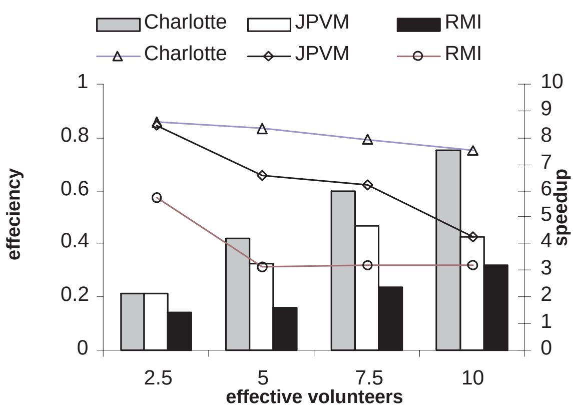
3.3.2 Implementation

Worker Process: A Charlotte worker process is implemented by the `Cdaemon` class which can run either as a Java application or as a Java applet. At instantiation, a `Cdaemon` object establishes a TCP/IP connection to the manager and maintains this connection throughout the computation.

Two implementation features are worth noting. First, since `Cdaemon` is implemented as an applet (as well as an application), the code does not need to be present on volunteer machines before the computation starts. By simply embedding the `Cdaemon` applet in an HTML page, browsers can download and execute the worker code. Second, the `Cdaemon` class, unlike its counterpart the Calypso worker, is independent of the Charlotte program it executes. Thus, not only are Charlotte workers able to execute parallel routines of any Charlotte program, but only the necessary code segments are transfered to volunteer machines.

Manager Process: A manager process begins with the `main()` method of a program and executes the non-parallel steps in a sequential fashion. It also manages the progress of parallel steps by providing scheduling and memory services





to workers. They are based on eager scheduling, bunching, and TIES.

Distributed Shared Class Types: Charlotte's distributed shared memory is implemented in pure Java at the data-type level; that is, through Java classes as stated above. For each primitive Java type like `int` and `float`, there is a corresponding Charlotte class-type `Dint` and `Dfloat`. The member variables of these classes are a `value` field of the corresponding primitive type, and a `state` flag that can be `not_valid`, `readable`, or `dirty`. It is important to note that different parts of the shared data can be updated by different worker processes without false sharing, as long as the CRCW-Common condition is met. (That is, several workers in a step can update the same data element, as long as all of them write the same value.) The shared memory is always logically coherent, independently of the order in which routines are executed.

3.3.3 Performance Experiments

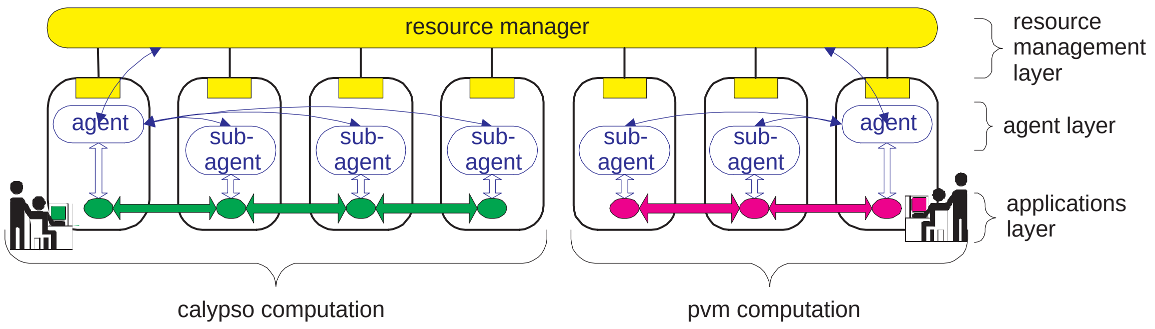
The experiments were conducted in the same execution environment as in Section 3.1.5. Programs were compiled (with compiler optimization turned on) and executed in the Java Virtual Machine (JVM) packaged with Linux JDK 1.1.5 v7. TYA version 0.07 [22] provided just-in-time compilation.

A publicly available sequential ray tracing program [24] was used as the starting point to implement parallel versions in Charlotte, Java RMI [14], and JPVM [15]. Java RMI is an integral part of Java 1.1 standard and, therefore, it is a natural choice for comparison. JPVM is a Java implementation of PVM, one of the most widely used parallel programming systems. For the experiments, a 500×500 image was traced. The sequential program took 154 s to complete, and this number is used as the base in calculating the speedups.

The first series of experiments compares the performance of the three parallel implementations of ray tracer, see Figure 9. In the case of Charlotte, the same program with the same runtime arguments was used for every run—the program tuned itself to the execution environment. For RMI and JPVM programs, on the other hand, executions with different grain sizes were timed and the best results are reported—the programs were hand-tuned for the execution environment. The results indicate that when using 16 volunteers, the Charlotte implementation runs within 5% and 10% of hand-tuned JPVM and RMI implementations, respectively. It is encouraging to see that the performance of Charlotte is competitive with other systems that do not provide load balancing and fault masking.

The final set of experiments illustrates the efficiency of the programs when executing on machines of varying speeds—a common scenario when executing programs on the Web. Exactly the same programs with the same granularity sizes as the previous experiment were run on n, $1 \leq n \leq 4$, groups of volunteers, where each group consisted of four machines: one normal machine, one machine slowed down by 25%, one machine slowed down by 50%, and one machine slowed down by 75%. Each group has a computing potential of 2.5 volunteer machines. The results are depicted in Figure 10. As the results indicate, the Charlotte program is the only one able to maintain its efficiency—the efficiency of the Charlotte program degraded by approximately 5%. In contrast, the efficiency of RMI and PVM programs dropped by as much as 60% and 45%, respectively.

ResourceBroker [4] is a resource management system for monitoring computing resources in a distributed multiuser environments and for dynamically assigning them to con-



currently executing computations. Although applicable to a wide variety of computations, including sequential ones, it especially benefits adaptive parallel computations.

Adaptive parallel computations can effectively use networked machines because they dynamically expand as machines become available and dynamically acquire machines as needed. While most parallel programming systems provide the means to develop adaptive programs, they do not provide any functional interface to external resource management systems. Thus, no existing resource management system has the capability to manage resources on commodity system software, arbitrating the demands of multiple adaptive computations written using diverse programming environments. Indeed, existing resource management systems are tightly integrated with the programming system they support and their inability to support more than one programming system severely limits their applicability.

ResourceBroker is built to validate a set of novel mechanisms that facilitate dynamic allocation of resources to adaptive parallel computations. The mechanisms utilize low-level features common to many programming systems, and unique in their ability to transparently manage *adaptive parallel programs that were not developed to have their resources managed by external systems*. The ResourceBroker prototype is the first system that can support adaptive programs written in more than one programming system, and has been tested using a mix of programs written in PVM, MPI, Calypso, and PLinda.

4 Computing Communities: Metacomputing for General Computations

So far, we have addressed metacomputing for parallel computations. Operating systems such as Amoeba [29], Plan-9 [26], Clouds [12] and to an extent Mach [1] had targeted the use of distributed systems for seamless general purpose computing. However, the rise of commodity operating systems and the need for application binary compatibility have made such approaches less attractive, necessitating instead that general computations also be supported on metacomputing environments. To enable the latter, we have designed and will implement the *Computing Community* (CC) framework.

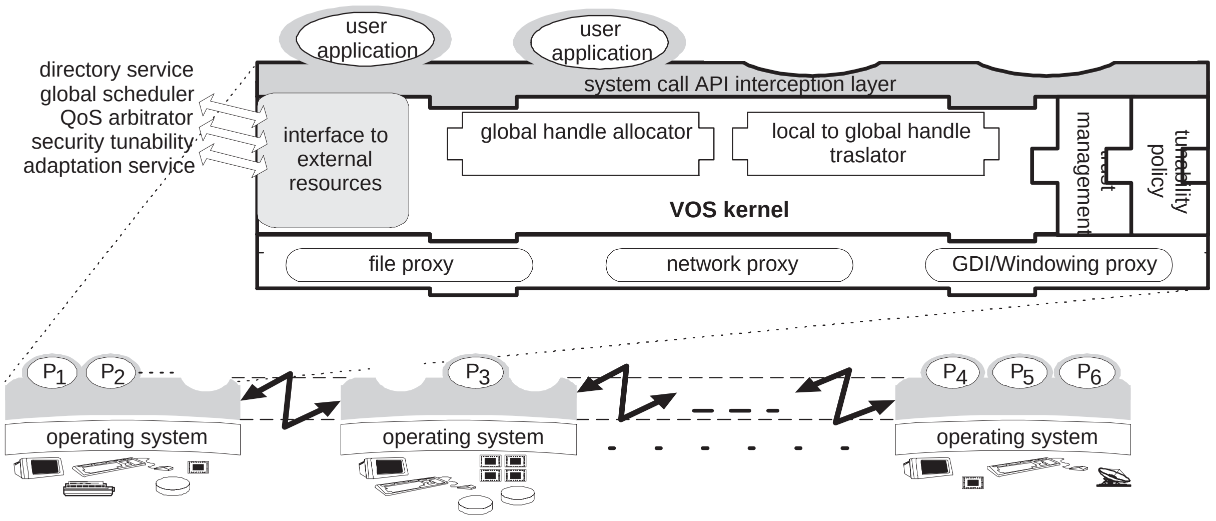
A Computing Community (CC) is a collection of machines (with dynamic membership) that form a single, dynamically changing, virtual multiprocessor system. It has global resource management, dynamic (automatic) reconfigurability, and the ability to run binaries of all applications designed for a base operating system. The physical network disappears from the view of the computations that run on the CC.

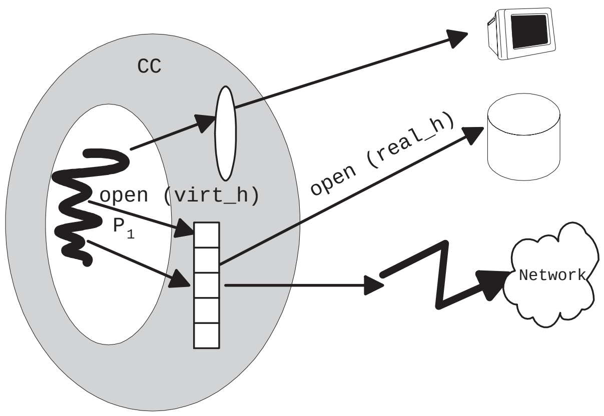
The CC brings flexibility of well-designed, distributed computing environments to the world of non-distributed applications-including legacy applications-without the need for distributed programming, new APIs, RPCs, object-brokerage, or similar mechanisms.

We are in the process of building a CC on top of the Windows NT operating system, with the initial software architecture shown in Figure 12. The CC comprises three synergistic components: (1) Virtual Operating System (2) Global Resource Manager (3) Application Adaptation System.

The *Virtual Operating System* (VOS) is a layer of software that non-intrusively operates between the applications and the standard operating system. The VOS presents the standard Windows NT API to the application, but can execute the same API calls differently, thereby extending the OS's power. The VOS essentially decouples the virtual entities required for executing a computation from their mappings to physical resources in the CC.

The *Global Resource Manager* manages all CC resources, dynamically discovering the availability of new resources,





integrating them into the CC, and making them available for use by CC computations. It handles resource requests from other components of the system and satisfies them as per scheduling requirements.

The *Application Adaptation System* enables the computations to take full advantage of CC resources and provides dynamic reconfiguration capabilities. Adaptation techniques allow computations to become aware of and gracefully adapt themselves to changes in CC resource characteristics.

Figure 13 shows a conceptual view of a CC. It takes a set of operating systems, and a set of resources, and via a layer of middleware converts it into an integrated community. CCs can expand and contract dynamically, and the computations are completely mobile within CCs. In short, using the CC framework, the computation transparently acquires the benefit of operating in a distributed environment.

Under a standard OS, a process runs in a logical address space, is bound to a machine, and interacts with the OS local to this machine. In fact, the processes (and their threads) are virtualizations of the real CPUs. However, such virtualization is low-level and limited in scope.

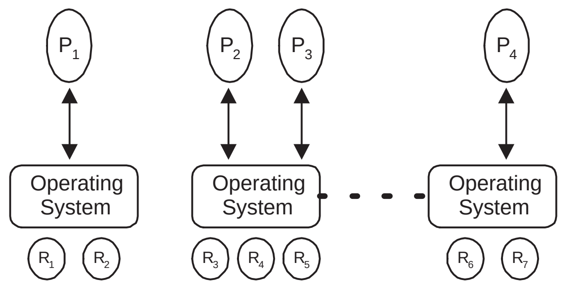
In the CC, virtualization is defined at a much higher level, and all physical resources (CPU, memory, disks, and networks) as well as the OSs on to all the machines are aggregated into a single, unified (distributed) virtual resource space.

A process in the CC is enveloped in a virtual shell (Figure 14), which makes the process feel that it is running on a standard OS. However, the shell creates a virtual world made of the aggregate of the physical worlds in the CC.

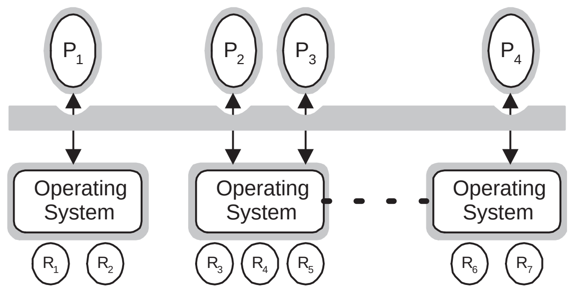
Consider a user *U* who starts an application *A* (and its GUI) on some machine M_1. Soon, *U* abruptly moves to another machine M_2. Now *U* can instruct the CC to connect the virtual screen, virtual keyboard, and the virtual mouse of *A* to the physical resources of M_2. The CC complies and *U* continues working on M_2, as if *A* executed there. Later the CC might decide it preferable to run the application on M_2. The scheduler then transparently moves *A* to M_2 preserving process state and open files and network connections.

The above simple scenario shows a particular aspect of the power of virtualization. In general:

- The users can move their virtual "home machines" at will, even for applications that are currently executing. This is the ultimate mobile computing scenario.
- A critical service running on machine M_1 can be moved



Processes (P_i), operating systems, and resources (R_j) in a conventional system



CC augments the original system with thin per-process, per-resource, and global system-wide entities

to machine M_2 if M_1 has to be relinquished.

- Schedulers can control the complete set of resources.

- The provision of multiple physical resources for a single virtual resource delivers important new capabilities ranging from duplicating application displays on multiple screens to replicating processes for fault tolerance.

The CC functionality relies upon three key mechanisms: API interception, proxies, and translations between physical and logical handles. API interception allows the API calls from an application to the operating system to be intercepted and the behavior of the API call to be modified. After intercepting a call, the virtual operating system (VOS) does one of the following operations. (1) Passes the call on to the local Windows NT operating system. (2) Passes the call to a remote Windows NT operating system. (3) Executes the call inside the VOS. (4) Executes some VOS code and then passes the call to a local or remote Windows NT system.

In order to reallocate processes to machines, a general form of process migration is necessary. To move a process from one location to another, just moving the state is not enough, all connections and handles have to be moved. This can be achieved by having proxies that emulate the connections of the process after the process has moved. For example, if a process *P* moving from M_1 to M_2 has an open networking connection to M_3, a proxy is created on M_1, which keeps the original connection to M_3 open, and then forwards messages between *P* and M_3, after *P* has moved.

Equally essential to successful virtualization of resources for migrating processes is the use of virtual handles. For example when a process opens a file on top of a VOS, the VOS intercepts this call and stores the returned physical handle but returns to the process a handle, which we refer to as virtual. The virtual handle can be used by the process, regardless of migrations, to access that file, due to the transparent translation service provided by the VOS. The virtual handles are used to virtualize I/O connections, sub-processes, threads, files, network sockets, etc.

Acknowledgements

This paper describes work of not only the authors but also other participants in the MILAN project, particularly Fangzhe Chang, Ayal Itzkovitz, Mehmet Karaul, Holger Karl, Donald McLaughlin, Shantanu Sardesi, Peter Wyckoff, and Yuanyuan Zhao. This research was sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0320; by the National Science Foundation under grants number CCR-94-11590 and CCR-95-05519; by Deutsche Forschunggemeinschaft; by Intel; and by Microsoft. 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.

References

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. *Proceedings Summer USENIX*, July 1986.

[2] Y. Aumann, Z. M. Kedem, K. Palem, and M. Rabin. Highly efficient asynchronous execution of large-grained parallel programs. In *Proceedings of the Annual Symposium on the Foundations of Computer Science (FOCS)*, 1993.

[3] A. Baratloo, P. Dasgupta, and Z. M. Kedem. Calypso: A novel software system for fault-tolerant parallel processing on distributed platforms. In *Proceedings of International Symposium on High-Performance Distributed Computing (HPDC)*, 1995.

[4] A. Baratloo, A. Itzkovitz, Z. M. Kedem, and Y. Zhao. Mechanism for just-in-time allocation of resources to adaptive parallel programs. In *Proceedings of International Parallel Processing Symposium (IPPS/SPDP)*, 1999.

[5] A. Baratloo, M. Karaul, H. Karl, and Z. M. Kedem. KnittingFactory: An infrastructure for distributed web applications. *Concurrency: Practice and Experience*, 10(11–13):1029–1041, 1998.

[6] A. Baratloo, M. Karaul, Z. M. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, 1996.

[7] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, Apr. 1989.

[8] M. Chandy and C. Kesselman. Compositional C++: Compositional parallel programming. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, 1992.

[9] F. Chang, V. Karamcheti, and Z. M. Kedem. Exploiting application tunability for efficient, predictable parallel resource management. In *Proceedings of International Parallel Processing Symposium (IPPS/SPDP)*, 1999.

[10] R. L. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. *Computer Graphics*, 18(3):137–145, July 1984.

[11] P. Dasgupta, Z. M. Kedem, and M. Rabin. Parallel Processing on Networks of Workstations: A Fault-Tolerant, High Performance Approach. In *Proceedings of 15th International Conference on Distributed Computing Systems (ICDCS)*, 1995.

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

[13] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, Sept. 1965.

[14] T. Downing. *Java RMI: Remote Method Invocation*. IDG Books Worldwide, 1998.

[15] A. Ferrari. JPVM—network parallel computing in Java. In *Proceedings of the Workshop on Java for High-Performance Network Computing*, 1998.

[16] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel virtual machine*. MIT Press, 1994.

[17] W. Gropp, E. Lust, and A. Skjellum. *Using MPI: Portable parallel programming with the message-passing interface*. MIT Press, 1994.

[18] S. Huang and Z. M. Kedem. Supporting a flexible parallel programming model on a network of workstations. In *Proceedings of International Conference on Distributed Computing Systems (ICDCS)*, 1996.

[19] S. F. Hummel, E. Edith Schonberg, and L. E. Flynn. Factoring: A method for scheduling parallel loops. *Communications of the ACM*, 35(8):90–101, Aug. 1992.

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

[21] Z. M. Kedem, K. Palem, and P. Spirakis. Efficient robust parallel computations. In *Proceedings of the ACM Symposium on the Theory of Computing (STOC)*, 1990.

[22] A. Kleine. Tya archive. Availabe at `http://www.dragon1.net/software/tya`.

[23] D. McLaughlin. *Scheduling Fault-tolerant, Parallel Computations in a Distributed Environment*. PhD thesis, Arizona State University, December 1997.

[24] L. McMillan. An instructional ray-tracing renderer written for UNC COMP 136 Fall ’96. Available at `http://graphics.lcs.mit.edu/˜capps/iap/class3/RayTracing/RayTrace.java`, 1996.

[25] OpenMP Architecture Review Board. *OpenMP: A Proposed Industry Standard API for Shared Memory Programming*, 1997.

[26] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell labs. In USENIX, editor, *Computing Systems, Summer, 1995.*, volume 8, pages 221–254, Berkeley, CA, USA, Summer 1995. USENIX.

[27] S. Sardesai. *Chime: A Versatile Distributed Parallel Processing Environment*. PhD thesis, Arizona State University, July 1997.

[28] S. Sardesai, D. McLaughlin, and P. Dasgupta. Distributed Cactus Stacks: Runtime stack-sharing support for distributed parallel programs. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1998.

[29] A. S. Tanenbaum and S. Mullender. An overview of the Amoeba distributed operating system. *Operating Systems Review*, 15(3):51–64, July 1981.