DPAC: An Object-Oriented Distributed and Parallel Computing Framework for Manufacturing Applications

N. R. Srinivasa Raghavan and Tanmay Waghmare

***Abstract*—Parallel and distributed computing infrastructure are increasingly being embraced in the context of manufacturing applications, including real-time scheduling. In this paper, we present the design and implementation of one such framework that can work on the Internet, with applications in manufacturing. The architecture, aliased as DPAC (Distributed and Parallel Computing framework), has the goal of harnessing the Internet's vast, growing computational capacity for ultra-large, coarse-grained parallel applications. The idea is to bring together diverse, heterogeneous, geographically distributed computing environments in order to attack large-scale computing problems. We present a scalable and fault-tolerant architecture in DPAC and the results of running performance experiments. DPAC is implemented on the interoperable, increasingly secure, and ubiquitous platform, viz, Java. The unique feature of DPAC is that it frees application developers from concerns about complex interprocess communication and fault tolerance among Internet-worked hosts and supports piecework and branch-and-bound computational models. We describe an implementation and present case studies showing the effectiveness in solving complex combinatorial optimization problems in the context of manufacturing systems.**

***Index Terms*—Application programming interface, distributed computing, fault tolerance, large-scale computing, manufacturing automation.**

I. Introduction

DISTRIBUTED and parallel computing framework (DPAC) is an object-oriented infrastructure for global computing. To bring together computational resources to attack a single large problem, existing algorithms must be made to work in a distributed fashion. The problem of distributing a computation has been studied and is understood in the context of traditional computing environments. Projects such as Legion [1], Linda [2], Condor [3], and Globus [4] all provide basic software infrastructure for supporting parallel computing. For application programmers, however, using these infrastructures to build a metacomputing application can be quite difficult. Most of these frameworks either are not designed with heterogeneity of hardware and operating systems in mind, or require knowledge of network programming to be able to connect the various computing components. Our software programming framework is a complete, easy-to-use tool whereby users can distribute large, diverse, scientific computation in a metacomputing environment.

The primary purpose of this paper is to describe the implementation of a software framework based on Master-Worker paradigm [5] that enables users to build applications quickly and easily, and that runs on a metacomputing platform. Contributions to fault tolerance and scalability, intended to support larger sets of computational resources, are presented. The paper presents a branch-and-bound computational model, piecework computational model, a corresponding application programming interface, implementation of a scalable task scheduler, and a fault-tolerance scheme.

By providing a portable, secure programming system, Java holds the promise of harnessing this large heterogeneous computer network as a single, homogeneous, multiuser multiprocessor. There are many issues that affect every global computation, including performance, correctness, scalability, and fault tolerance. Some research projects that are designed to exploit these include Popcorn [6] and Atlas [7]. DPAC is designed to obtain the performance of massively parallel implementation and providing a simple application programming interface (API), allowing designers to focus on recursive decomposition of the parallelizable part of the computation. The application programmer gets the performance benefits of massive parallelism without adulterating the application logic with interprocess communication and fault tolerance details. The resulting code should run well, as a set of processes that changes during execution. DPAC handles all interprocess communication and fault tolerance for the application programmer.

The goal of DPAC system is to exploit the networked resources of the (mini) world as a distributed computer and to develop an infrastructure that exploits idle resources both within and among institutions.

A. DPAC and Manufacturing Control

Although the infrastructure in DPAC is intended for use with computers on the Internet, it indeed can be applied in the context of manufacturing control applications that require real-time decision making. For instance, we refer to the Manufacturing Automation Protocol (MAP) protocol sponsored by General Motors [8]. One can build a distributed control architecture using MAP, as indicated in Fig. 1.

The host computer needs to solve several real-time problems. For instance, it has to decide on the beginning time of jobs that

Manuscript received December 19, 2001. This paper was recommended for publication by Associate Editor M. Fayad and Editor N. Viswanadham upon evaluation of the reviewers' comments.

N. R. S. Raghavan is with Management Studies, Indian Institute of Science, Bangalore 560 012, India (e-mail: raghavan@mgmt.iisc.ernet.in).

T. Waghmare is with Microsoft India Development Center, HITEC City, Hyderabad 500 033, India.

Digital Object Identifier 10.1109/TRA.2002.802236

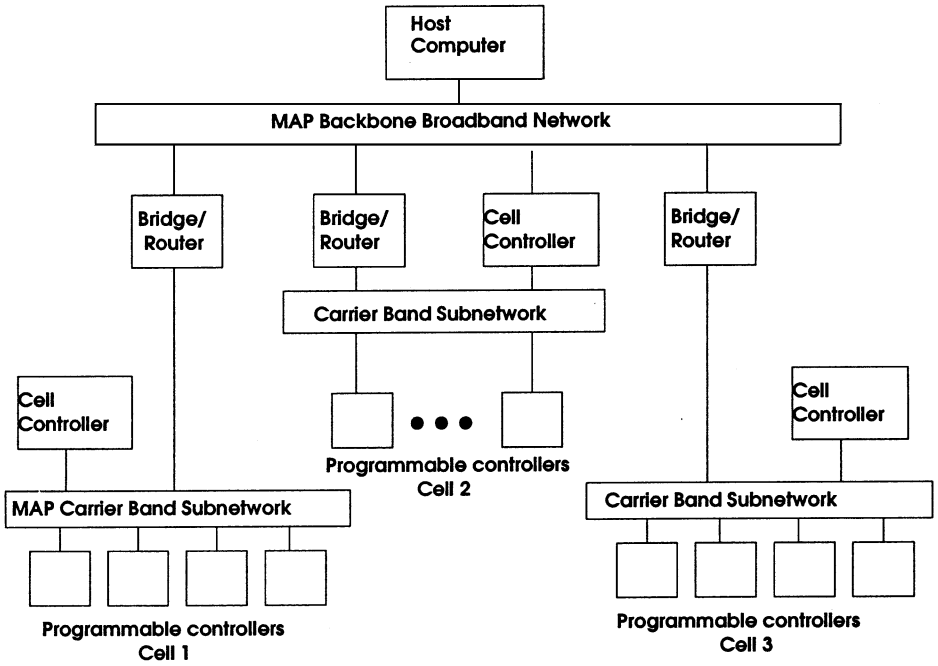


Fig. 1. Factory communications using MAP.

need to be processed and controlled by individual cell controllers; in this case, a scheduling problem is solved. Typically, solving scheduling problems to optimality is computationally challenging. Hence, one resorts to some form of enumeration in combination with heuristics [9]. Since the computer has to decide in real time, sequential algorithms need to be parallelized so as to reduce the time required for computation. In this context, one can utilize the processing capacity available at the individual cell controllers, or even the programmable controllers at the machine level. Thus, DPAC can be utilized over these computers to solve real-time problems in the context of manufacturing. In fact, with the proliferation in the use of information technology in manufacturing, it is not inappropriate to have *web-enabled* virtual factories, where the various controllers need not be co-located. They can be on the Internet, if not on the corporate intranet.

In this paper, a scheduling application is presented that typically fits a distributed manufacturing control system. We also solve the same using our architecture and present some encouraging results.

B. Desirable Properties of DPAC

The design goal of DPAC is to achieve several desirable properties for any global computing infrastructure, including the following.

1) Scalability: It must scale to millions of nodes (processors).
2) Heterogeneity: It must include a variety of hardware platforms and operating systems, all of which must interoperate.
3) Fault Tolerance: An application should tolerate most failures and should have a probability of completion consistent with traditional applications.
4) Adaptive parallelism: The applications must dynamically exploit a varying collection of resources, so that long-running applications can grow, shrink, and migrate as required.
5) Safety: The system must ensure that foreign applications obey local limits on data access and resource utilization, so that users will be willing to add their machines to the global pool.
6) Anonymity: When they wish, users should be able to farm out their applications without risking any proprietary data.
7) Ease of use: The system should be easy to use.
8) Reasonable performance: The performance should show linear speedup for large classes of applications, and should have sufficiently low overhead so as to see a benefit even with only few machines.

We observe here that although the above properties are ideal, we were able to achieve, to a reasonable extent, some of the above properties, about which we discuss further on.

This paper is organized as follows. The following section discusses related work in this area. In Section III, we present the architecture of DPAC. The associated technical issues are discussed in Section IV. We present the DPAC application programming interface in Section V. In Section VI, we present two important case studies (applications) in manufacturing: one on parallel machine scheduling and the other on the traveling salesperson problem. We conclude in Section VII by summarizing our work and presenting future research and implementation issues.

II. Related Work

As high-performance distributed computation became more feasible and more widespread, the challenges involved in

constructing distributed and parallel computing infrastructure received increasingly more attention. Some of the most recent network computing approaches include Condor [3], Linda [2], PVM [10], JMPI [11], Legion [1], and Globus [4]. The goal of Legion project is to provide application-controlled fault tolerance, improved response time, and greater throughput. Globus is an execution environment involving high-speed networks connecting supercomputers. All the above systems require the maintenance of binaries for all architectures used in the computation.

The flexibility of Java for global parallel computing has been observed by several other researchers. These projects include Popcorn [6] and Atlas [7] designed to run parallel applications and provide a specific programming model. Other recent systems such as JPVM [10] and Java-MPI [11] use Java to overcome heterogeneity, but are not intended to execute on anonymous machines. There has been substantial work in the last decade geared toward heterogeneity and open systems. However, standards for interoperability and portability emerged only a couple of years ago.

We now present in some detail, work which is close in spirit to ours. Piranha [12] is a complete open source solution for Linux providing high availability (HA), Internet protocol (IP) service monitoring, and failover and load balancing. The Red Hat High Availability Server is a software product designed to provide HA services (clustering, load balancing, node failover, etc.) in the Linux marketplace. It is currently an integration of Red Hat Linux, the Piranha clustering software, and the Linux Virtual Server (LVS) software. Piranha is a collection of programs that interact with each other to provide a clustering solution. It is vital to note that cluster computing consists of two distinct branches.

1) Compute clustering that uses multiple machines to provide greater computing power for computationally intensive tasks. This type of clustering is not addressed by Piranha [13].
2) High Availability (or HA) clustering uses various technologies to gain an extra level of reliability for a service. HA clustering is the focal point for Piranha.

Thus, Piranha does not provide for the computing services that are provided in DPAC.

Another framework worth considering is PLinda [14]. A variant of Linda [2], persistent Linda (PLinda) strives to provide for fault tolerance and the effective utilization of idle workstations, an objective very close to that of DPAC. The architecture apparently provides a combination of checkpointing and transaction support on both data and program state. The traditional (database) transaction model is optimized and extended to support robust parallel computation. Treatable failures in PLinda include processor and main memory hard and slowdown failures and network omission and corruption failures. It may be noted that the implementation was done in C++ and experiments were run on Sun Sparc machines. The architecture does not support heterogeneity, while DPAC can be run on several types of machines, as our experimentation indicates. Also, in PLinda, one cannot add compute servers (clients, as per PLinda) dynamically, once a computation has been initiated, which is possible in DPAC. The way we handle fault tolerance is by maintaining a computation tree which will be discussed in Section IV-A. We do not recover failed transactions, unlike in PLinda. If a worker does not come up with a response due to failure, we time out and assign it to a new worker. We feel that procedures like checkpointing will result in overheads. Finally, PLinda does not accommodate machines on the Internet. It seems to have been implemented on a LAN. Whereas in DPAC, workers can participate via the Internet as long as the DPAC-API is available.

The SuperWeb [15] architecture, on the other hand, supports global computing with three kinds of participants: brokers, clients, and hosts. Brokers coordinate the supply and demand for computing resources. Hosts offer their resources as a commodity by registering with the broker. Clients are users or computers that need extra computational resources. The architecture is still under implementation, with initial ones being run on intranets of large organizations. According to the authors, SuperWeb is ideally suited where communications among the parallel processes is minimal. We have tested DPAC on algorithms where communication does exist among the workers (the scheduling case study indeed is fine grained) and obtained encouraging results. Also, SuperWeb does not account for fault tolerance.

The search for extraterrestrial intelligence (SETI) project [16] is yet another recent attempt at creating a massive supercomputer using several publicly available personal computers. Individuals will have to download a screen-saver program that will not only provide the usual attractive graphics when their computer is idle, but will also perform sophisticated analysis of SETI data using the host computer. The architecture is tailor made for solving the scientific computing problem of locating lives in other galaxies. It is essentially equivalent to the client-server paradigm of computation rather than the master-worker-client paradigm adapted in DPAC. Validity checks are done at the server end so as to eliminate incorrect responses from clients. The code for solving the pieces of computation was ported to several platforms, and the client has to download the program depending on the configuration of the computer at its end. It may be noted that SETI uses Internet computing in a more or less passive way, while DPAC adopts the grid [17] approach to computing, providing the necessary software and services to programmers, in the form of APIs.

III. Architecture of DPAC

DPAC architecture has three basic system entities, Client, Master, and Workers. A Client is a process seeking computing resources; a Worker is a process offering computing resources; a Master is a process that coordinates the allocation of computing resources.

Fig. 2 illustrates the DPAC architecture. Clients register their tasks to be run with the master, workers register their intention to run tasks with the master. The master assigns tasks to workers, that then run the tasks and send results back to the client. Even a client can host the computation.

DPAC master, worker registry, naming service, and communication is based on Java Remote Method Invocation (RMI).

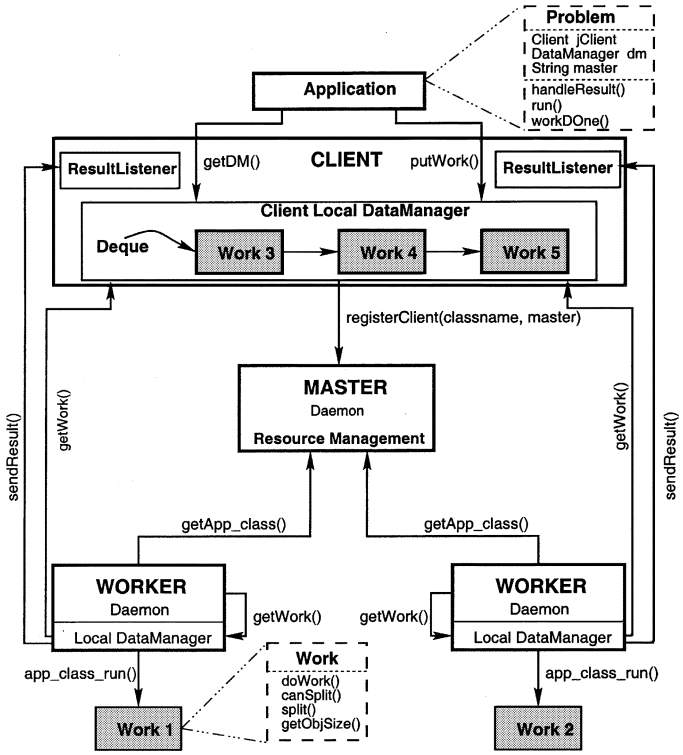


Fig. 2. DPAC architecture.

We are aware that, since RMI is implemented on the top of transfer communication protocol (TCP), we will suffer a slight performance penalty. However, since the focus of this work is on enhancing scalability and heterogeneity, we are more interested in a convenient distributed object technology that hides the peculiarities of the communication subsystem from the developer. The application programmer no longer needs to implement communication primitives. Of course, the use of RMI requires the presence of JDK1.1.x or later at any host participating in DPAC computation.

One of the most important goals of DPAC is simplicity, i.e., to enable everyone connected to Internet or Intranet to easily participate in DPAC. The design is based on a widely used component like portable-language Java. Java offers the basic infrastructure needed to integrate computers connected to the Internet into a seamless distributed computational resource, an infrastructure for running coarse-grained parallel applications on numerous, anonymous machines.

By simply running worker class on local Java Virtual Machine (JVM) and pointing to master running on an anonymous machine with its IP address, users automatically make their resources available to host parts of parallel computations. This is achieved by starting a small daemon that waits and listens for tasks from the master. The simplicity of this approach makes it easy for a worker to participate. All that is needed is JVM, DPAC class library, and IP address of the master machine. All communication must be routed through the master. Therefore, in general, coarse-grained applications with high computation-to-communication ratios are well suited to DPAC. With a new JDK1.2 security model, it is possible to run Java applications just as secure as applets.

A. Naming Service

When a worker or client wants to connect to DPAC, it first must find a master that is willing to serve it. DPAC uses scalable, fault-tolerant RMI registry service that enables lookup of master. It is used not only to aid workers who are searching for master, but also to aid workers and master who are looking for application class of client to load.

B. Master Network and Resource Management

Master runs a daemon thread registering itself to the registry. It can only handle a constant number of workers. Since master is expected to be a lot more stable and reliable than other workers, certain conditions are to be met. The master must run on a host with a permanent connection to the network, and the user donating a master host must be prepared to run master for a long duration.

When a worker connects to master, the master enters the worker in a tree data structure with client as parent, and null if no client is running. All workers will not receive the parent, instead, they will later become children of the client. This way master maintains the workers which are set on standby until a client becomes active. When client connects, all workers are activated in a single operation and client information is passed to the workers. Master can individually set how many workers it can administer. Master has a thread running for each worker and has remote handle to function wait_for_death() in the respective workers. In case of a worker failure, master receives an exception, and the master unregisters the worker, removing its entry from the worker table and cleans up the worker resources.

C. Code Distribution

Client and master do not actively lookup for workers to join a computation. Workers can join at any time by contacting the master. If every worker that participates in a computation had to go to the client to load the application class, this would soon lead to a bottleneck for a large number of workers. Therefore, first the master acts as a *cache* on behalf of the client. The loading and caching mechanism is implemented as a modification to the standard Java classloader. Wherever a loadclass() fails at worker, it is translated to an RMI call to the master, which in turn will deliver the requested class, the client will finally be contacted and delivers the original class file, which will be then cached at master. Subsequent requests by workers will not reach the client again, thus eliminating the bottleneck in the system.

Presently, like a standard classloader, the DPAC classloader loads all classes on demand, i.e., only when they are needed. To increase execution performance, it might be beneficial to preload application classes in future. We describe the sequence (see Fig. 3) from the moment a client application is willing to execute until a worker has received the code to participate in the computation.

1) The worker registers with the master and shows intention to provide computing resources to run the tasks.
2) Master sets the registered workers on standby till it has no client and hence, no application to run.

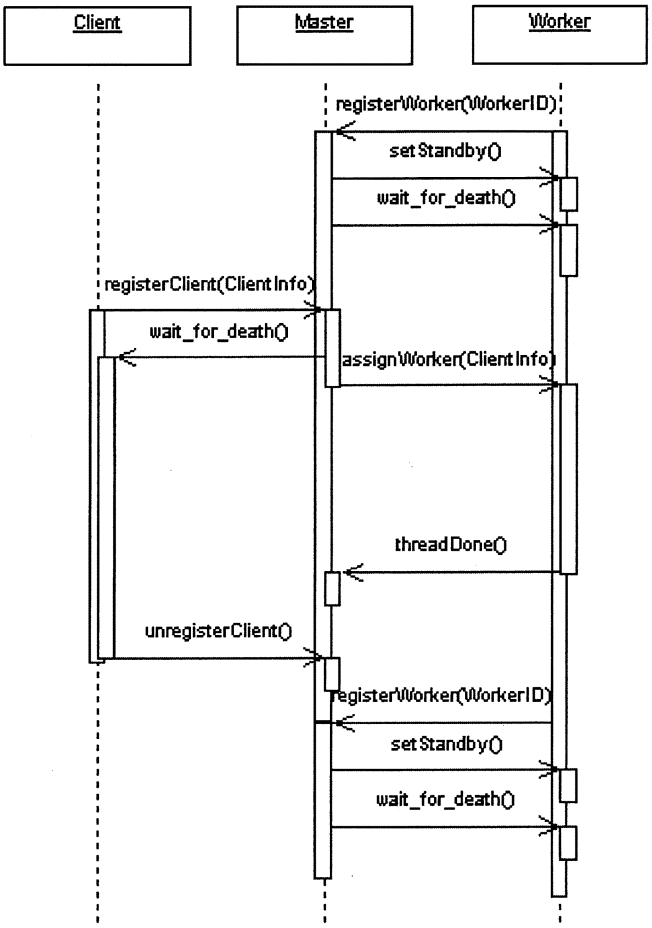


Fig. 3. Sequence diagram.

3) The client registers with the master seeking computing resources. It also sends a description of the application (name of the application class and ID of the client).
4) Master assigns the workers to this client and returns the application information to all the workers.
5) A new worker can join the system by starting the daemon and contacting master asking for code to execute.
6) The worker daemon, after getting the application information, executes the above mentioned class-loading mechanism to load the application.
7) A new thread is created and the application starts to execute on this worker.

To summarize, the interface to DPAC has two parts. Firstly, users submit jobs to the local datamanager and send information about the application to the master. Secondly, master assigns the workers to this client so that workers can start stealing work from client and send back the results after executing those tasks.

D. Model of Computation

DPAC supports both piecework as well as the branch-and-bound computation model. The branch-and-bound method, which generalizes the piecework model of computation, intelligently enumerates all feasible points of a combinatorial optimization problem: not all feasible solutions are examined. Branch-and-bound, in effect, proves that the best solution is found without necessarily examining all feasible solutions. The method successively partitions the solution space (branches) and prunes a subspace, when there is sufficient information to infer that none of the subspace's solutions are as good as a current solution (bound). The computational model implies the following requirements.

1) Tasks are to be generated during computation.
2) When worker discovers a new best cost, it propagates it to all other workers.
3) Detecting termination in a distributed implementation requires knowing when all subspaces (children) have been either fully examined or killed.

The challenge is, with minimum communication, to enable the creation of tasks which can be distributed, propagate new bounds rapidly to all workers, and to enable the scheduler to detect tasks that have been completed or killed. Scheduler is not just needed for termination detection, but also for fault tolerance, i.e., to determine which tasks need to be rescheduled.

IV. Technical Issues

We now elaborate on the various technical issues associated with the implementation of DPAC. Some features covered include fault tolerance, scalability, heterogeneity, security, and the computational economics issues. We also justify the choice of technologies used.

A. Fault Tolerance

We may resort to the following (see [18]) types of failures in distributed computing contexts. Omission failures occur when a server omits to respond to a request. Response failure occurs when a server responds incorrectly to a request (it either returns a wrong value, or undergoes incorrect state transitions with respect to data values). Crash failures are implied in the presence of repeated omission failures. This can be either an amnesia crash, pause crash, or a halting crash [18]. Byzantine failures are a combination of all the above types. Recovery from such failures can be effected by building necessary services in the architecture, including checkpointing transactions as in [14]. In the case of DPAC, we mask these failures by rescheduling the failed processor jobs to other workers, using time-out mechanisms and the problem tree mechanism described below. Workers that never return an answer are removed from the framework due to the time-out mechanism at the master. In the case of sudden failure of workers, the partial work is not recovered; we reschedule the same to other workers.

For fault tolerance purposes, DPAC employs distributed scheduling, where a piece of work can be reassigned to an idle processor in case its result has not been reported. It is a low-overhead way of ensuring progress of failures or varying processor speeds.

The DPAC scheduling logic is located at the client. The basic data structure required is a heap-like problem tree, which the client maintains to keep track of computation status. The tree has a node for every piece of work the problem can possibly generate and is structured as follows. At its root is the complete, undivided problem itself; its children are the subproblems resulting from a single split of the root problem, and so on, until

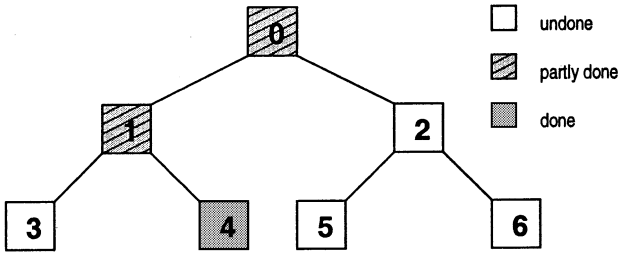


Fig. 4. Scheduling: result 4 arrives.

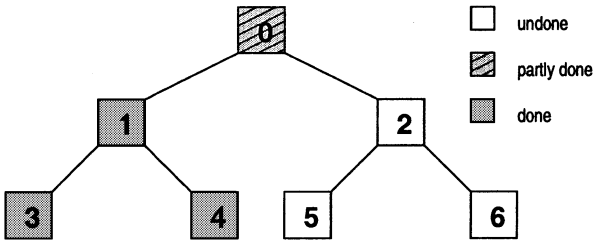


Fig. 5. Scheduling: result 3 arrives.

the atomic pieces of the problem appear at the leaves of the tree. Each node can be in one of the three states: done, meaning the result for the subproblem has been received by the client; partly done, meaning result for the subproblem has been received by the client for some but not all subproblems (descendents) of this subproblem; and undone, meaning that no results whatsoever have been received by the client for this subproblem. Initially, all nodes are in undone state.

In the first part of processing, the client records all incoming results. Specifically, the client marks the subproblem corresponding to the incoming result as done, and then recurses up the problem tree marking ancestors of the subproblem either done or partly done depending on their current status. The first result for a subproblem is always recorded and passed to the client's result handler. Any subsequent results for the same subproblem are simply discarded.

In the second part of processing, the scheduling routine is invoked only when the clients task queue is empty, and the worker fails to steal work from the client. In this case, the client selects the next piece of work marked undone for rescheduling. Since the problem tree is organized as an array with work (piece) sizes monotonically decreasing, this piece is guaranteed to be the largest available undone piece. The next time selection is invoked, it will proceed from the current point to select the next largest undone piece, and so on. By selecting largest available piece of work and reissuing it for processing, the distributed work stealing ensures that this work will be split up and distributed among the participating workers just as the pieces were distributed initially.

Figs. 4–6 give an example of scheduling. In Fig. 4, we see how the result of the atomic piece with ID 4 arrives at the client. The client subsequently marks all its ancestors including the root as partly done. In Fig. 5, another result arrives for the atomic piece with ID 3. Now the client can also mark the parent node of this piece as done. Finally, in Fig. 6, we show how, assuming no results arrive at the client and work stealing has failed, the client

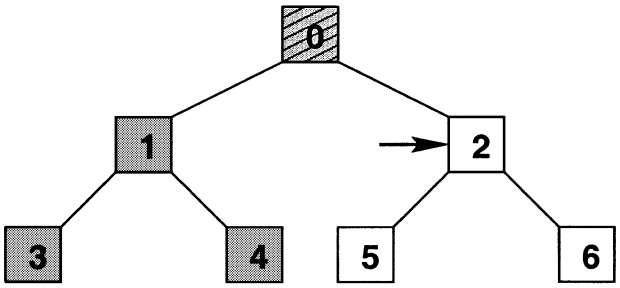


Fig. 6. Scheduling: piece 2 selected.

selects piece number 2 as the largest undone piece of work. The piece will now be reissued for host work. It may be split subsequently and part of it stolen by other workers.

B. Scalability

When we say that a global computational infrastructure is scalable, we mean that its components have bounded power, bounded computational rate, and bounded communication rate. In particular, to make DPAC scalable, the master, client, and worker each possess bounded power. These bounds imply that, for example, client, master, and workers can communicate with only a fixed number of other components during a fixed interval of time. Thus, at any point of time, there are bounds on the number of connections between the master and workers and between the client and master.

The fundamental concept underlying our approach of task scheduling is work stealing, a distributed scheduling scheme made popular by Cilk Project [19]. The approach is entirely demand driven. When a worker runs out of work it requests work from the client that it knows. The computational load is balanced as long as number of tasks is high relative to the number of workers—a property well suited for adaptively parallel systems.

In the implementation of task scheduling, we use the task queue data structure local to each host containing chunks of work. The working principle of double-ended queue (deque) is as follows. Tasks get split in a task queue until a certain minimum granularity, determined by the application, is reached, and then they are processed. When a worker runs out of local tasks, it requests work from client, retrieves it, and computes one task at a time. This helps each worker to get a quantity of work commensurate with its capabilities. The client is the root of its tree of workers and manages the worker tree. When a worker joins a computation, it is assigned a leaf position in the worker tree by the tree manager. The worker tree fanout can be set, for each application, at startup.

The key problems in building a scalable architecture are distributing the code efficiently to a potentially very large number of workers, identifying workers for clients for its computation, and managing the exchange of data between workers after an application has been successfully started. We now describe our approach to solving these problems. The complete state transition diagram of a worker during its lifetime is as shown in Fig. 7. If a worker has not joined DPAC, it is in state "NoHost." The transition to "Standby" is made by starting DPAC daemon and then registering with a master. We have already explained how the

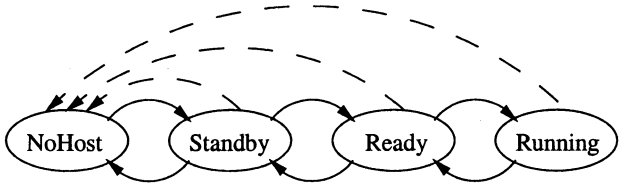


Fig. 7. State transition diagram for Worker.

code is shipped so that an application is ready to start, causing a state transition from "Standby" to "Ready." Data is exchanged amongst workers by our task scheduling mechanism that allows the worker to run the application, and therefore the transition to "Running." The diagram has two more sets of transitions, a natural way back from each state to the previous when a phase has terminated, and a set of interrupt transitions (dashed lines) that lead back to the "NoHost" state when a user withdraws the worker from the system. Adding the above mechanisms in DPAC makes it scalable. One can add as many workers and clients to the system and still handle large pieces of work since the scheduling/allocation of jobs is streamlined. We summarize as follows. The scalability of the scheduling comes from distributed work stealing. The scalability of the applications comes from the programming model, which has no predefined limit on the usable resources, such as a target number of processors.

C. Adaptive Parallelism

The work-stealing scheduler and the tree-structured programming model of DPAC allow programs to run on a set of workers that can grow (due to new workers joining) and shrink over time (due to failures). When idle, the worker can automatically join the execution of a parallel program by work stealing. Contrarily, when the worker is no longer available, all of its subcomputations are reassigned to another worker. This feature is common in many grid computing frameworks (see [17]).

D. Interoperability

We observe that for the grid computing framework to be as large based as possible (in terms of the number of participating computers), we require DPAC to support processing on heterogeneous machines. This means that the computers have different cental processing unit (CPU) makes and run different operating systems. To incorporate heterogeneity in the architecture, one could precompile the binaries on several combinations of processors and operating systems (as is done in [16]). This would mean the availability of programmers who can skillfully port applications written a native language to several operating system platforms, as well as hardware platforms. We did not adopt this approach. Instead, we make the client ship portable, interpreted bytecode (Java bytecode is an example). This clearly makes the user less burdened in terms of programming.

The basic heterogeneity comes from Java for free, however, we must be able to port the servers programmed in DPAC and all of its libraries to each platform. In the future, we expect the number of quality Java libraries to increase dramatically, making this task easy to accomplish.

Also, to circumvent the problem of low execution speeds while using Java, modern techniques such as just-in-time (JIT) compilation and dynamic compilation are available that can eliminate most of the interpretation overhead by compiling pieces of the application machine code. In fact, compared to shipping precompiled binaries as in [14] or [16] which requires the host machine to trust the binaries, Java does provide for added security features. We discuss more issues in Section IV-F.

E. Security

We look at security from two perspectives.

1) Worker's view: We should not allow client codes to make unrestricted access to the worker's resources (including processor, memory, disk, etc.). This will ensure that more users participate in DPAC as workers. This was indeed one of the motivations for us to use interpreted code (Java bytecode), unlike in PLinda [14] where C++ executables are shared.
2) Client's view: Since the workers execute the byte code of clients, it is required to ensure that the worker does not get access to client's sensitive data and algorithms. For instance, if we run stock market applications on DPAC, the data and algorithms are indeed sensitive from client's perspective. This requires the use of encrypted computing, which is itself in a stage of active research [20]. For instance, one could encrypt the data that is sent and use decryption to reclaim the results, while neither the master nor the workers have knowledge of real data. Alternatively, the algorithm itself could be encrypted (see, for instance, [15]). For the type of problems solved in this paper, encrypting the data seems to be the more easily implementable one. We do not provide support for encryption or decryption in DPAC, which we reserve for future work.

The current architecture of DPAC predominantly supports security from the worker's viewpoint. As far as the client is concerned, we provide a relatively weak form of anonymity in DPAC. A worker executing a stolen subtree cannot determine the original source of the computation, since all results are collected and processed by the client. Thus workers, at best, can gain a limited knowledge of the client's data/algorithms; they cannot get the big picture.

Also, there could be a host of issues like denial-of-service attacks by spurious clients, code containing virus (not likely in our case, since code is interpreted), etc., which can be preempted by known methods (see [21]) at the worker's or the master's end. This again is not part of DPAC.

F. Computational Economics

As has been proposed in SuperWeb [15], it would be ideal to have a scheme for charging clients for the amount of processing that they have submitted to DPAC. But this calls for more overheads in the system. A separate component has to be identified with the task of efficient billing and crediting. Protection will have to be provided against spurious workers participating with arbitrary and suspicious bids for computation. Mechanisms should be in place to ensure that the workers indeed did some computation and did not merely return some numerical figures. We see that these contribute to the computation time; neverthe-

less, because it motivates more workers to join DPAC in the Internet setting, we leave this as an issue for future implementations.

G. Choice of Technology

RMI is a set of API that allows developers to build distributed applications in the Java programming language. RMI uses Java language interfaces to define remote objects and a combination of Java serialization technology and the Java Remote Method Protocol (JRMP) to turn local method invocations into remote method invocations [22]. We have used Java RMI as already indicated, for implementing remote method invocations. We considered the common object request broker architecture (CORBA) as a competing technology, and decided in favor of RMI for the following reasons. Since our intention is to assist clients which have parallelized algorithms, we assume that the algorithm is coded in a single programming language. As such, there is no necessity to integrate with codes written in other languages. Whereas, CORBA is essentially meant to be implemented when one is trying to link disparate islands of software programs, for example, linking a business application written in COBOL with another one that runs using C/C++. Also, CORBA does not allow executable files to be sent to remote systems (whereas Java RMI does). It only allows (see, for instance, [23]) primitive data types and structures to be passed and not the actual code, while RMI allows Java objects to be passed and returned as parameters. This allows new classes to be passed across virtual machines for execution, which feature was exploited in our experimentation.

Also, describing services in CORBA requires the use of an interface definition language (IDL) and one requires IDL-to-(programming)language mapping tools to create code stubs based on the interface. All these contribute to overheads in a predominantly compute-intensive architecture like ours. CORBA may well suit an enterprise application integration solution.

Specifications including J2EE and .NET are specifically meant for building enterprise components with applications in enterprise resource planning (ERP) type of software. For instance, to reduce costs in fast-track enterprise application design and development, the J2EE platform provides a component-based approach to the design, development, assembly, and deployment of enterprise applications [22]. Features including security and persistence of transactions are tailor made for such applications. Indeed, the J2EE platform supports JRMP.

In our experiments, we designed and implemented components specific to DPAC in Java. The packages which are going to be described in the next section are indeed components. We hope that they can be potentially integrated with other J2EE based applications when DPAC is made to run with such applications.

V. DPAC API

In this section, we illustrate our system from an application programmer's point of view. We first present the classes needed by the programmer to create a DPAC application.

A DPAC application consists of one client and many workers. The client is responsible for initiating the computation, managing the problem, and collecting the results. It may or may not do part of the actual computation. The workers help the client manage and compute the problem. The client code executes on a single machine, while the host code is distributed throughout the DPAC network and executed on many different machines.

The application programmer must provide code for both the client and the worker, which may actually be joined together in a single source file as our example below shows, plus the implementation of three interfaces needed by the system.

A. DPAC Package

This package contains all the core classes needed by the client, workers, and the master. The programmer writing an application for DPAC only needs to get acquainted with the Client class.

```
public class Client
{
 public Client(String className, String
master); public void
begin(DataManager dm);
 public void terminate();
}
```

DPAC client must instantiate the client class. The only constructor of client takes the top-level class name (used to load the host classes), the master's host name, and any input data that workers may need as arguments. Once the client is ready to start the computation, the client invokes the begin() method. Its parameter is an RMI handle to the client's data manager, which will be cached on the master and passed on to a worker upon registration. The begin() method causes the client to register with the master, which in turn assigns workers to the client's computation. The terminate() method unregisters the client, allowing the master to clean up and stop assigning workers to the client. It should be called after the computation is done to detach gracefully from the master.

B. DPAC UTILITY Package

To manage the computation, clients and workers must instantiate the data manager in this package. Data managers dictate how the computation is divided, how workers obtain work, and how results return to the client. They are also responsible for providing scalability and fault tolerance. DPACDataManager implements the DataManager interface shown below.

```
public interface DataManager {
 public void addWork(Splittable work);
 public Splittable getWork();
 public void sendResult(Serializable res);
 public void propagate(Serializable val);
}
```

The three main methods are addWork(), getWork(), and returnResult(). In our model, a client uses the first method to pass new work to the data manager. In our model of computation, this method is typically only executed once by the client to initialize

the computation. The getWork() method is used by a worker to obtain an atomic piece of the computation. In case the computation produces a usable result, the worker passes that result to the client using the returnResult() method, which also invokes the result processing at the client.

To support the travelling salesperson problem (TSP) application's bound propagation, the propagateValue() method was added. It passes on a new bound value to the data manager's propagation thread, which in turn sends the new bound asynchronously to all workers.

The programmer must also tell the data manager how to notify his application whenever a new result arrives and when all the work is complete. This is done by the methods setResultListener() and setDoneListener(). The two methods are mainly needed on the client which needs to process results and is interested in knowing when the computation is complete.

We now mention how the client conveys its work to a data manager. For this, the programmer defines a class, representing the type of work to be done, that implements the Splittable interface, shown below. The data manager uses the Splittable methods to divide and distribute the work to workers.

```
public interface Splittable {
 public boolean canSplit();
 public SplitArray split();
 public int getObjectSize();
}
```

The split() method splits the work represented by a particular object into suitable subobjects. The results of the split are returned in a variable-size array. The canSplit() method determines if a split is possible and is always invoked prior to split() method. If canSplit() returns false, the split() method will not be called. Finally, the getObjectSize() method simply returns the integer size of the piece of work.

C. Application Code

```
 /* single source file containing code for
both client and worker */
public class Application implements
Runnable
{
 private void run()
 {
  /* application set up client */
  if(isClient) {
   Client cl = new Client(className,
master);
   DataManager dm = Runtime.getDM();
   dm.addWork(new Work(MAXSIZE));
   /* register with master & start */
   cl.begin();
  }
  else { /* this is the worker part */
   DataManager dm = JRuntime.getDM();
  for((chunk = (Work)dm.getWork()) !=
null)
   dm.sendResult(chunk.doWork());
 } /* end of run */
 public static void main(String[] args)
 {
  Application client = new Application();
  client.run(); /* start an application
thread */
 }
} /* end of Application */
```

VI. Case Studies

In this section, we discuss two important applications of DPAC in the context of manufacturing systems. In the first case study, we present a new parallel algorithm for solving an existing sequential one, in the area of parallel machine scheduling [9]. The second case study solves the TSP using known parallel algorithms for the problem. We present interesting experimental results for both the above cases. It may be observed that the TSP solution may be required for manufacturing of printed circuit boards (PCBs); in this case, the machine has to solder a large number of points on the PCB and in the process the machine head has to move from one point to another on the board. A TSP results and the client for DPAC, here, would be the machine controller, with all other controllers as potential workers for DPAC.

A. Piecework Computation

We have implemented DPAC for solving a problem for scheduling independent jobs with due dates on identical, parallel machines. The problem is formulated and discussed in [9]. The problem can be stated as nonpreemptive scheduling of jobs, where each job is made up of a small number of operations that must be undertaken in a particular order. Jobs also have different due dates and different levels of importance. The objective is to minimize the total weighted quadratic tardiness of the schedule. Since this problem is NP-hard (see [9]), an efficient near-optimal algorithm based on Lagrangian relaxation is implemented.

The minimization problem is decomposed into a set of small subproblems using Lagrangian relaxation, each of which can be solved in polynomial time. To obtain a feasible solution, the dual solution found by subgradient algorithm is used to form an ordered listing of jobs. A greedy method of list scheduling is then applied to the list to assigned jobs on machines. The optimal schedule generated in this way has ability to react to dynamic changes in manufacturing systems.

The discrete time integer programming formulation of the problem is as follows (we use the notation and formulation as in [9]):

δ_{ik}	A 0-1 variable; equals 1 if job i is active at time k, 0 else.
N	Number of jobs, K time horizon.
B_i	Beginning time of job i.
C_i	Completion time of job i.
D_i	Due date of job i.
J	Objective function to be optimized.
t_i	Time required from the resource by job i.
M_k	Number of machines available at time k.

T_i Tardiness of job i; equals $\max\{0, Ci - Di\}$.
w_i Weight (importance) of job i.
The scheduling problem can now be formulated as

$$P : \min_{\{B_i\}} J, \; with \; J \equiv \sum_i w_i T_i^2 \tag{1}$$

subject to capacity constraints

$$\sum_i \delta_{ik} \leq M_k, \qquad k = 1, 2, \ldots, K \tag{2}$$

and processing time requirements

$$C_i - B_i + 1 = t_i, \qquad i = 1, 2, \ldots, N. \tag{3}$$

Relaxing the capacity constraint by using Lagrange multiplier π_k, the problem P is decomposed as

$$R : \min_{\{B_i\}} \left\{ \sum_i w_i T_i^2 + \sum_k \pi_k \left(\sum_i \delta_{ik} - M_k \right) \right\}$$

subject to processing time requirements (3).

Then the dual problem is $D : \max L$, with

$$L \equiv \left\{ - \sum_k \pi_k M_k + \min_{\{B_i\}} \sum_i \left\{ W_i T_i^2 + \sum_k \pi_k \delta_{ik} \right\} \right\}$$

This leads to decomposed subproblem for each job i [given π and subject to (3)]

$$R_i : \min_{1 \leq B_i \leq K - t_i + 1} L_i, \qquad \text{with } L_i \equiv \left\{ w_i T_i^2 + \sum_k \pi_k \delta_{ik} \right\}.$$

There are several steps to obtaining an optimal solution: solving subproblems, solving dual problems, constructing a feasible solution, and finding an optimal solution, which are discussed in [9]. We make an important observation that, in the above derivation, L_i are the independent subproblems which can be run in parallel in DPAC. The master in DPAC takes care of updating the Lagrange multipliers π, and constructs a good feasible solution (see [9]) if an optimal one does not result. The workers, on the other hand, compute the optimal value of L_i for all i. The iterative solving scheme necessitates transfer of the values of the Lagrange multipliers from the master to the workers, and R_i and δ_{ik} from the workers to the master. This makes the computation a little fine grained. To facilitate this data exchange, we used JavaSpaces [24] technology where workers would deposit their current values of R_i and δ_{ik} after solving the subproblems, onto JavaSpace, while the master solves the Lagrangian dual problem every iteration and updates the value of the vector π which is then written on the JavaSpace. It may be observed that the workers have to wait until the master computes the current value of π every iteration and the master waits for all workers to submit their current values of R_i and δ_{ik}. This introduces some synchronization delay.

For purposes of experimentation, we also coded the sequential version of this algorithm as in [9]. We present the results of solving the above problem using DPAC in Table I. In our experiments, without loss of generality, we generated job and machine characteristics (specifically, the parameters w_i, t_i, D_i, K, and M_k) randomly. We ran our experiments on a 200–job problem with 50 iterations. We used two Pentium-III 800 MHz processors, one with 256–MB RAM running Windows 98, and the other with 64–MB RAM running Red Hat Linux. We simulated workers in DPAC by opening as many servers (ports) required as the number of workers on both the computers and running DPAC daemons on them. (The next case study presents a more realistic setting with real processors and not *ports* acting as workers.) We feel that this does not dilute the utility of DPAC and, in fact, it illustrates the flexibility offered by DPAC to an application programmer.

TABLE I
PERFORMANCE OF DPAC WITH THE SCHEDULING PROBLEM

Number of workers	Speed up
1 (Serial case)	1.0000
2	1.2579
4	1.8099
6	2.7886

We define speed up in this case as the ratio of time taken to solve in the serial case to that of the time taken by DPAC. We find that the speed up appears to be super linear. This is indeed encouraging. However, with the above setting of having workers simulated as ports on both the computers, when we increased the number of workers to ten, the speed up was found to decrease to 1.5387, which is counterintuitive. We argue that this is because of using ports for servers, rather than individual processors themselves, since the more the number of ports, the more are the overheads on the operating system.

A more interesting application of our framework was with the solution to the TSP using the branch-and-bound technique, which is illustrated below.

B. Branch-and-Bound Computation

Branch-and-bound method intelligently enumerates all feasible points of a combinatorial optimization problem. The TSP is a computationally complex combinatorial problem.

Communication latency is too large among Internetworked processors. However, for the TSP application of branch-and-bound, a weak shared memory model is sufficient. For our implementation, one shared variable is enough to perform well. All workers have a cached copy of the current minimum cost. If the worker's copy of minimum cost is stale, correctness is unaffected. However, performance suffers because it might process a partial solution that could be killed. When a worker discovers a solution with a smaller cost than its cached minimum cost, it sends this solution to the client. If client agrees that this indeed is a new minimum cost solution, it updates its cached minimum cost solution and propagates the new minimum cost solution to all workers. This propagation is handled asynchronously by a separate propagator thread, so that a worker is not blocked until all the workers have acknowledged the new bound.

We tested the performance of DPAC with TSP application. The performance was measured by recording the time to find the shortest tour in a given complete, undirected, weighted graph of 22 nodes with randomly generated edge weights. The graph is

TABLE II
EXPERIMENTAL RESULT: PRACTICAL SPEEDUP

Size (*nodes*)	Serial (*min*)	DPAC (*min*)
16	1	0.11
17	3.7	0.29
18	5.3	0.34
19	7.2	0.27
20	10	0.46
21	212	4.11
22	169	3.73
23	52	1.36
24	653	17
25	-	143
26	-	712

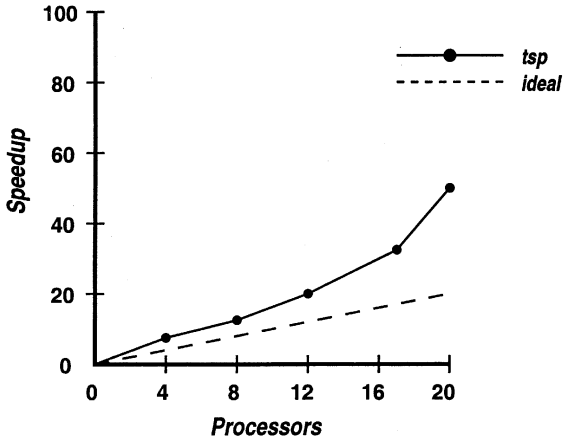


Fig. 8. Practical speedup for TSP on DPAC.

complex enough to justify parallel computing, but small enough to enable us to run the tests in a reasonable amount of time. Experiments were run in computing labs under typical workload. The heterogeneous test environment consists of the following.

1) Ten Pentium III 800 MHz processors with 128–MB RAM running Red Hat Linux 6.1.
2) Two Dual Pentium III 450 MHz with 256–MB RAM running Red Hat Linux 6.1.
3) Ten Pentium III 450 MHz processors with 128–MB RAM running Red Hat Linux 6.0.
4) Three Sun sparc quadprocessors running SunOS 5.4.

The 22-node graph took approximately three hours to process on a Pentium III 800–MHz processor. The term *speedup* is somewhat confusing here. Traditionally, speedup is measured on a dedicated multiprocessor, where all processors are homogeneous in hardware and software configuration and varying workloads between processors do not exist. Thus, speedup is well defined T_1/T_p, where T_1 is the time a program takes on one processor and T_p is the time the same program takes on p processors. Therefore, strictly speaking, in a heterogeneous environment like ours, the term speedup cannot be used anymore. Even if one tries to run tests in as homogeneous a hardware setup as possible, the varying workloads on both the operating system (OS) and the network can amount to big differences in the individual performance of hosts. However, from a practical standpoint, a user running an application on DPAC with a large set of hosts will definitely see speedup; the application will run faster than on a single machine. We will use *practical speedup* to distinguish between the two scenarios. This definition does not incorporate machine and network load factors, it does reflect the heterogeneous nature of the set of machines.

The graph that took almost three hours to calculate on a single computer took just under four minutes on 25 machines that were also servicing students with their normal, everyday workloads.

We consider these results highly encouraging, although they need to be evaluated further with different input graphs and a higher number of machines.

Table II shows the practical speedup we measured in our experiments with below mentioned test environment of Serial Configuration and Parallel Configuration. Varying graph size from 16 to 26 nodes, time taken by one processor, and parallel configuration of processors is recorded as shown in the table.

1) Serial Configuration
— A Pentium III 800–MHz processor with 128–MB RAM running Linux.
2) Parallel Configuration
— 14 Pentium III 450–MHz processors with 128 MB-RAM running Linux.
— Six Pentium III 450–MHz processors with 128–MB RAM running Windows-NT.

Fig. 8 shows the superlinear speedup in all measured configurations in our experiments. To observe superlinear speedup for the parallel TSP is quite common, due to the inherent irregularity of the input graph. For instance, say that the optimal tour is found by the sequential algorithm in one of the last pieces processed. If a parallel version finds the optimum much faster, it can spread the bound to all other hosts and they can prune the search tree much more efficiently, thus resulting in a much better running time and superlinear speedup. However, looking at results, it seems that the given test graph was *very* favorable to this kind of phenomenon.

VII. DISCUSSION

In this section, we discuss how the DPAC architecture achieves the desired properties listed in the introduction.

1) **Scalability:** All parts of the system are scalable. The scalability of the scheduling comes from distributed work stealing. The scalability of the applications comes from the programming model, which has no predefined limit on the usable resources, such as a target number of processors.
2) **Heterogeneity:** The basic heterogeneity comes from Java for free, however, we must be able to port the server and all of its libraries to each platform. In the future, we expect the number of quality Java libraries will increase dramatically.
3) **Fault tolerance:** The fault tolerance comes from the tree structure of the DPAC programming model. DPAC employs distributed scheduling, where a piece of work can be reassigned to an idle processor in case its result has not been reported. It is a low-overhead way of ensuring progress of failures or varying processor speeds.

4) **Adaptive parallelism:** The work-stealing scheduler and the tree-structured programming model allow programs to run on a set of workers that grows and shrinks over time. When idle, the worker can automatically join the execution of a parallel program by work stealing. When the worker is no longer available, all of its subcomputations are reassigned to another worker.
5) **Safety:** The safety of DPAC depends on the safety of Java and of the native libraries. We take both of these for granted. Complete coverage of security and safety is beyond the scope of this paper.
6) **Anonymity:** We provide a relatively weak form of anonymity. A host executing a stolen subtree cannot determine the original source of the computation, since all results are collected and processed by the client.
7) **Ease of use:** The interface has two parts. First, users submit jobs to the local datamanager and send information about the application to the master. Second, master assigns the workers to this client so that workers can start stealing work from client and send back the results after executing those tasks.
8) **Reasonable performance:** Superlinear speedup has been observed in all measured configurations of our experiments conducted for TSP computations.

VIII. Conclusion and Future Work

Parallel computing infrastructures must scale to large numbers of resources, which indicates the fundamental need for fault tolerance. Accordingly, the focus of DPAC is to provide a Java-based high-performance network computing architecture that is both *scalable* and *fault tolerant*. We supported a piecework model of computation and the branch-and-bound model of computation. The principal requirement of branch-and-bound computation is broadcasting a new bound from the worker that discovered it to all other workers. The semantically weak shared-memory model is sufficient for branch-and-bound computation. Many combinatorial optimization versions of NP-hard problems are solved with branch-and-bound (e.g., Integer Linear Programming Problem). We conjecture that all such problems are well suited for the Java-based distributed and parallel computing, DPAC.

We have presented an approach to distribute application code in a scalable manner through the master, which acts as a code cache for the workers connected to it. We have also implemented a distributed task scheduler, providing essential fault tolerance. DPAC has adopted the master–worker paradigm which makes it simple for the application programmer to program, and can neatly handle an environment where resources are *unreliable*; the users can also easily incorporate *heterogenous* resources for computation.

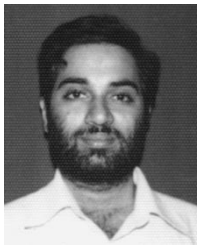
Future work includes addressing limitations due to network latency and exploiting different paradigms, such as having a hierarchy of masters that control the computation. Correctness checking, checkpointing for partial work recovery, and including the accounting mechanisms into DPAC are some more core issues that merit our attention.

Acknowledgment

The authors wish to thank Prof. Y. Narahari, Department of Computer Science and Automation, Indian Institute of Science for his critical comments and suggestions during the course of our work. They also thank A. Malik for assisting them while implementing the first case study presented in this work.

References

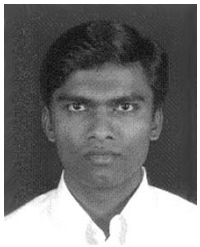
[1] A. S. Grimshaw and W. A. Wulf, "The legion vision of a worldwide virtual computer," *Commun. ACM*, vol. 40, no. 1, Jan. 1997.
[2] R. A. Whiteside and J. S. Leichter, "Using Linda for Supercomputing on a Local Area Network," Dept. Comp. Sci., Yale Univ., New Haven, CT, Rep. YALEU/DCS/TR638, 1988.
[3] M. Litzkow, M. Livny, and M. W. Mutka, "Condor—A hunter of idle workstations," in *Proc. 8th Int. Conf. Distributed Computing Systems*, June 1988.
[4] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," *Int. J. Supercomp. Applicat. High Perform. Comput.*, vol. 11, no. 2, pp. 115–128, Summer 1997.
[5] M. Yoder, S. Kulkami, and J. Linderoth. An Enabling Framework for Master-Worker Application on the Computational Grid. [Online]. Available: http://www.mcs.anl.gov/metaneos/papers/mw2.ps
[6] N. Carniel, S. London, N. Nisan, and O. Regev, "The popcorn project: Distributed computation over the internet in java," in *Proc. 6th Int. World Wide Web Conf.*, Apr. 1997.
[7] J. E. Baldeschwieler, R. D. Blumofe, and E. A. Brewer, "Atlas: An infrastructure for global computing," in *Proc. Seventh ACM SIGOPS Eur. Workshop on System Support for Worldwide Applications*, Connemara, Ireland, Sept. 9–11, 1996.
[8] N. Viswanadharn and Y. Narahari, *Performance Modeling of Automated Manufacturing Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1992.
[9] D. Hoitomt, P. Luh, K. Pattipati, and E. Max, "Scheduling jobs with simple precedence constraints on parallel machines," *IEEE Trans. Contr., Syst. Manage.*, vol. 10, pp. 34–40, Feb. 1990.
[10] A. Ferrari. Jpvm—The Java Parallel Virtual Machine. [Online]. Available: http://www.cs.virginia.edu/~ajf2j/Jpvm.html
[11] S. Taylor. Prototype Java-Mpi Package. [Online]. Available: http://cisr.anu.edu.au/~sam/java/java_mpi_prototype.html
[12] High Availability Server Project [Online]. Available: http://ha.redhat.com/
[13] S. Tweedie, "Designing a Linux Cluster,".
[14] K. Jeong, "Fault-tolerant parallel processing combining Linda, checkpointing and transactions," Ph.D. dissertation, Dept. Comp. Sci., State Univ. of NY, 1995.
[15] D. A. Albert *et al.*, "Superweb: Research issues in java-based global computing," *Concurrency: Prac. Exp.*, vol. 9, no. 6, pp. 535–553, 1997.
[16] W. T. Sullivan *et al.*. (1997) A New Major SETI Project Based on Project Serendip Data and 100 000 Personal Computers. [Online]. Available: http://setiathome.ssl.berkeley.edu/woody_paper.html
[17] I. Foster. (2000) Internet Computing and the Emerging Grid. [Online]. Available: http://www.nature.com/nature/webmatters/grid/grid.html
[18] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design*, 2nd ed. Reading, MA: Addison-Wesley, 2000.
[19] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proc. 5th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPOPP '95)*, Santa Barbara, CA, July 1995, pp. 207–216.
[20] D. Chess and B. Crosof *et al.*, "Itinerant Agents for Mobile Computing," IBM, Rep. RC 20 010, 1995.
[21] W. Stallings, *Cryptography and Network Security*. Englewood Cliffs, NJ: Prentice-Hall, 2000.
[22] J2EE Blueprints [Online]. Available: http://java.sun.com/blueprints
[23] Corba Meets Java, B. Morgan. (1998). [Online]. Available: http://www.javaworld.com/jw-10-1997/jw-10-corbajava.html
[24] JavaSpaces (TM) Technology [Online]. Available: http://java.sun.com/products/javaspace



N. R. Srinivasa Raghavan received the B.Tech. degree in mechanical engineering from S. V. University College of Engineering, Tirupati, India. He received the M.Tech. degree in management studies from the Indian Institute of Science (IISc), Bangalore, India in 1995, securing first rank and the Institute gold medal, and the Ph.D. degree from the Department of Computer Science and Automation, IISc, in 1999.

He is an Assistant Professor in the Department of Management Studies, IISc, Bangalore, India. His research interests include performance modeling and analysis of extended manufacturing networks, scheduling in the context of supply chains, intelligent agent-based modeling, and parallel and distributed computing. He has published his research in the *Journal of Operational Research Society* and *International Journal of Production Research*, apart from several international conferences, including IEEE ICRA, IEEE CDC, etc.

Dr. Raghavan was awarded the Prof. B. G. Raghavendra Memorial Trophy for part of his Ph.D. work at the First National Doctoral Consortium in Management held in India in 1997.



Tanmay Waghmare received the B.E. degree in mechanical engineering from Shivaji University, India, and, in 2000, the M.E. degree in computer science from the Indian Institute of Science, Bangalore, India.

He is currently working as a software engineer at Microsoft India Development Center at Hyderabad, India. His research interests include parallel and distributed computing, distributed operating systems, and object-oriented programming.