Parallel Object-Oriented Software And Tools

Laxmikant V. Kale, James Kohl, Nikos Chrisochoides, Katherine Yelick,

Laxmikant V. Kale, [*]
Department of Computer Science,
University of Illinois, Urbana-Champaign.
Email : kale@cs.uiuc.edu
Phone : (217) 244-0094

Nikos Chrisochoides[†]
Northeast Parallel Architectures Center
Syracuse University, Syracuse, NY 13244-4100

James Kohl,
kohl@msr.epm.ornl.gov, http://www.epm.ornl.gov/~kohl
Oak Ridge National Laboratory,
P.O. Box 2008, Bldg 6012,
Oak Ridge, TN 37831-6367

Katherine Yelick, [‡]
Computer Science Division, University of California, Berkeley
yelick@cs.berkeley.edu

July 4, 1995

Abstract

The inevitable transition to parallel programming can be facilitated by appropriate tools, including languages and libraries. After describing the needs of application developers, this paper presents three specific approaches aimed at development of efficient and reusable parallel software for irregular and dynamic-structured problems. A salient feature of all three approaches is their exploitation of concurrency within a processor. Benefits of individual approaches such as these can be leveraged by an interoperability environment which permits modules written using different approaches to co-exist in single applications.

[*]This work was supported in part by NSF grant: CCR-91-06608

[†]This work was supported by an Alex Nason Prize Award, by NSF ASC 93 18152/ PHY 93 18152 (ARPA supplemented), and by NASA Contract No. NAS1-19480, while the author was in residence at ICASE, NASA Langley Research Center, Hapton, VA 23681.

[‡]This work was supported in part by ARPA contract DABT63-92-C-0026, by DOE grant DE-FG03-94ER25206, by NSF grants CCR-9210260 and CDA-8722788. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

1 Introduction

The transition to parallel programming from sequential programming may be somewhat painful for application developers. Yet, it is a transition that is inevitable, and ultimately worth making. A well-designed set of tools — specifically, those based on an object-oriented methodology — will help application developers make this transition, and attain good performance on the new generations of parallel machines.

One of the problems faced by parallel programmers is that of obtaining good performance on irregular and dynamic problems. Many of the early successes of parallel processing were obtained on relatively regular problems — computations whose structures were *regular* such as 3-dimensional grids, and *static*, not changing with the progression of the computation. As the class of problems addressed by parallelism has broadened, one finds an increasing number of irregular problems. For example, many finite-element based problems generate irregular yet static structure. Problems based on adaptive meshes for fluid flows generate structures that change over time, and so do tree structured computations in state-space search, and branch-and-bound computations encountered in Operations Research.

How can one help the programmer deal with such irregularity? Concurrency within a processor is a helpful mechanism in this regard. A processor executes many computational actions; each action is typically dependent on results of local or remote computational actions. Instead of scheduling these computational actions in a rigid, fixed sequence we should allow them to be adaptively scheduled based on availability of the data they depend on. With such a mechanism the processor can be kept better utilized while running irregular and dynamic applications, where predicting the exact timing of availability of remote data isn't possible. Message driven objects, and multithreading are two different ways of specifying such adaptive concurrency within a processor. The three tools and approaches described later in this paper are based on such concurrency mechanisms.

The next section (contributed by J. Kohl) is written from an application developers perspective. It underscores the importance of explicit parallel programming, at least to complement the implicit or partially automated methods such as languages such as HPF. It also states the desirability of object based languages and identifies features of programming environments and tools that are particularly useful.

Multipol, described in Section 3 (contributed by K. Yelick), is a library of distributed data structures that are useful for programming irregular applications. The library uses the mechanism of split-phase transactions to generate concurrency within the processor and uses it to adaptively tolerate communication latencies. Applications developed using multipol are also briefly summarized.

Section 4 (contributed by N. Chrisochoides) describes an approach for dynamic load balancing of adaptive computations. The approach is aimed at adaptive PDE computations. It uses multithreading as its concurrency mechanism, using which it overlaps the load balancing computations with the actual application computations. This allows it to mask the inherent delays in synchronizations involved in traditional load balancing.

Section 5 (contributed by L. V. Kale) describes Charm, a system that uses message driven objects at its concurrency mechanism. It elaborates the relationship between message driven execution and modularity and describes other aspects of the Charm system. There are many different languages and approaches with distinct benefits. Providing a system which allows linking multiple modules written in different paradigms is therefore highly desirable, yet technically challenging. *Converse*, a system aimed at these challenges is also described in the last section.

2 Tools for Parallel Programming

Many programmers run away from programming a parallel computing system, fearful that the complexities involved present an impassable barrier. Many so-called "parallel" programming environments are really only simplifications of the underlying parallel architecture, and as such hide the parallelism, resulting in a deceivingly sequential-looking interface. This approach to parallel programming, while acceptable for certain legacy requirements such as parallelizing dusty-deck applications, is inadequate for "real" parallel programming. The ultimate goal of parallel computing is *performance* – either in terms of faster execution or the ability to solve larger problems. Ignoring the nature of the underlying system limits the ability of an application to fully utilize it. The only way to extract maximum performance is to specify the important details of the parallelism *explicitly*.

Sequential programming models are often forced onto parallel systems. These serial models are neither scalable nor high performance. And because such models do not correlate to the actual system, it may be difficult to determine the behavior of an application. It may not be obvious what to do to improve performance, let alone fix problems with functionality. While some programming environments may successfully provide good performance without explicit parallelism, they may do so by exchanging parallelism for a different type of complexity, e.g. shared memory cache management.

Parallelism need not be feared. Although it carries several burdens, in terms of communication and synchronization, race conditions, and problem/data partitioning, concurrency is a fundamental concept. The physical universe is overflowing with parallelism, as can be seen in our own human physiology. Just as a student complains about the perils of calculus relative to algebra, we resist parallelism. But we must also accept that algebra is not a suitable substitute for calculus. The primary difficulties with parallel programming may in fact result from a lack of adequate programming environments and tools. The remainder of section motivates this perspective and discusses the need for effective tools for parallel programming.

2.1 Application of Object-Oriented Techniques

One alternative to overcoming the complexities of parallel programming is to apply techniques from object-oriented programming. Using abstraction where appropriate, a programmer can focus on the critical aspects of program functionality and control complexity. The first step is to simplify the categorization of parallelism, to remove the redundancy found in common programming models. Whether using threads, monitors, message passing, RPC or active messages, the fundamental activities that arise in parallelism are *data movement* and *synchronization*. These activities must be performed efficiently to yield good performance. Data movement is used here to illustrate object-oriented techniques for specifying parallelism.

Data movement occurs when information must be shared among distinct parallel tasks. Using an object-oriented approach, a "data transfer" class should be defined which encapsulates the movement of data. For flexibility, there should be functions to send and receive data directly, and indirectly using asynchronous postings. Such functionality can be built on existing parallel communication systems such as PVM [4] or MPI [1].

Data movement performance is fundamentally different than sequential memory management, and the concept of a "cache" is less relevant. The locality of a parallel program is controlled by the data distribution and how data flows among the parallel tasks. Object-oriented techniques could assist the programmer with data partitioning. A global "data structuring" class could define the computational region and provide a variety of partitioning topologies. A region could be divided into subregions, one for each parallel computation engine. If sufficient information were provided

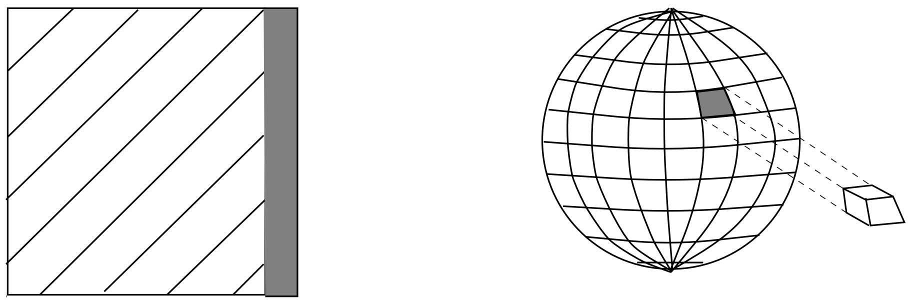

Figure 1: Global Data Structuring View

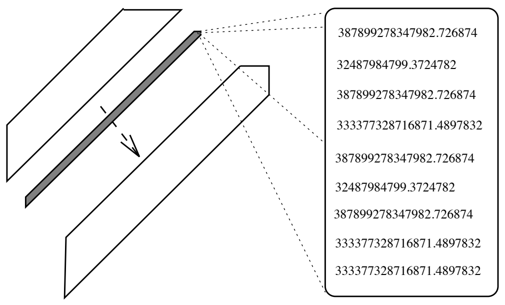

Figure 2: Data Movement View

by such a specification, better optimizations might be possible at compilation time.

2.2 Need for Tools

Unfortunately, no matter how superbly a parallel system is designed, no one will use it unless there are effective tools for analyzing programs. The tools should show what is going on inside a parallel program, and should provide an intuitive understanding of the parallelism. The use of "program visualization" [3], to animate program behavior using computer graphics, can be powerful if used wisely [2]. Visualization can be applied to represent the activity of the program using the fundamental language constructs and classes. The programmer should be able to relate the behavior depicted back to the source code.

An example of such a view could illustrate the structural information provided in the global data structuring class. A geometric representation could be automatically constructed to show the layout and activity of subregions, to verify the intended distribution or load balance (See Figure 1). This type of view could also be used to animate actual data transfers, if the subregions were stretched apart as shown in Figure 2. Small portions of a data partition could be animated as they transfer from one subregion to another. To be truly useful, such a view should also allow examination of the actual data values textually. The ability to control the level of detail in a view can be critical, especially in a massively parallel system. The object hierarchies used to construct a parallel program are a natural means for controlling the abstraction level.

In summary, parallel programming should not be the terrifying monster that programmers perceive it as being. To truly utilize a parallel system, parallelism must be faced and expressed *explicitly*. Object-oriented techniques can be applied to control parallel complexities. The primary obstacle to parallel programming is the lack of effective programming environments and tools. By taking advantage of the organization provided by object-oriented constructs, more powerful tools can be constructed to alleviate some of the difficulties of parallel programming.

3 A Multiported Object Library

Multipol is a distributed data structure library, designed to support applications with dynamic data structures, unpredictable computational costs, and irregular data access patterns. It includes parallel versions of trees, sets, lists, graphs, and queues and runs on several platforms. It is built on a portable runtime layer that provides basic communication, synchronization, and caching. To address the trade-off between locality and load balance data structures use a combination of replication, partitioning, and dynamic caching. To tolerate remote communication latencies, operations are split into a separate initiation and completion phase, allowing for computation and communication overlap at the library interface level. This leads to a form of relaxed consistency semantics for the data types. To allow for optimized local computation, the data structures are multi-ported: each processors has its own entry point or "port" to the data structures, so the client program can switch between a global and local view of the object. In this paper we give an overview of Multipol, discuss the performance trade-offs and interface issues, and describe some of the applications that motivated its development.

3.1 Multipol Applications

Multipol supports three types of irregular applications: bulk synchronous algorithms that manipulate irregular structures, divide-and-conquer algorithms, and asynchronous event-driven simulations. In each of these applications there are a small number of data structures that hold the shared state, which on become distributed on scalable multiprocessors and workstation networks.

The simplest irregular applications are the bulk synchronous ones, such as synchronous simulations on unstructured or adaptive meshes, which are irregular in space but not in time. The communication happens in distinct phases but has a irregular communication pattern that may be static over the execution or may vary with each iteration. For example, in the EM3D application kernel, computation proceeds by nearest neighbor computations on an unstructured mesh. Because of the separate phases, these applications are amenable to runtime communication structure analysis, as in the inspector/executor strategy used in Parti in which communication buffers are preallocated to speed communication.

Search and other divide-and-conquer problems have irregular task structure—the search tree unfolds dynamically into a tree of tasks with unknown pruning and task costs. In these applications, load balance is a primary concern, whereas the amount of shared state may be small, so locality is less important. Dynamic load balancing is done in Multipol using a distributed task queue and shared state to store solutions, failures, or pruning information is represented with a replicated or dynamically cached structure. For example, in the Phylogeny problem from biology, we use a trie [11] to represent the set of failures in a search space, and in the Göbner basis problem we use a list [10].

Event-driven simulation is perhaps the most challenging class of applications, because there are no natural global synchronization points at which load balancing and other communication can be done. These algorithms are used to allow different time steps in different parts of the simulation, to resolve a trade-off between accuracy and performance. Multipol provides n event graph data structure, containing a logical process at each node and events paths on each edge. Both conservative and speculative [13] can be done using event graphs. Like search problems, the computational costs are unpredictable, but unlike search, there is typically a large state associated with each task, so while dynamic load balancing is sometimes useful, it cannot be as aggressively as it is in search problems.

3.2 The Data Structures

Multipol data structures include objects to hold elements in a spacial domain, such as a bipartite graph for two-phase synchronous simulation, an event-graph for asynchronous simulation, an oct-tree for n-body simulations, and multi-dimensional arrays [12]. All of these structures are partitioned across processors, with small amounts of cached data for boundary values between processors. It also has data structures for storing unordered sets of values or sets of key/value pairs. A trie, B-tree, linked list, and hash table all provide this functionality, with the trie and linked list lazily replicated across processors, and the B-tree and hash table are partitioned. These differences in representation allow the user to choose a data structure that is faster for either updates or lookups, depending on the application demands. For dynamic load balancing, two forms of a distributed task queue are provided: one based on random placement of tasks with strong load balancing guarantees and one that uses heuristics to improve locality.

3.3 Performance Optimizations

A key problem in any parallel library effort is portability. Our primary machine targets are distributed memory machines such as Thinking Machines CM5, IBM SP1, Cray T3D and Intel Paragon, as well as networks of workstations. While at a functional level these platforms are very similar, the performance characteristics vary significantly. All of the machines have higher costs for accessing non-local memory than remote memory, whether this is done in hardware or in software, but the relative speeds of computation, the startup overhead of communication, the latency and the observed bandwidth all vary. The interface design and implementation of Multipol structures are aimed at coping with communication overhead and latency, using the following techniques.

Latency Masking: In Multipol we use message pipelining and multithreading to hide the cost of remote operations. The cost of performing these operations is made up of communication latency as well as the remote computation cost and any delays incurred remotely because the processor is busy with other work. We distinguished between remote operations with a relatively small bounded latency, such as reading or write a remote memory location, and those with long unpredictable latency, such as acquiring a remote lock or starting a remote thread. The former are done using pipelined active message communication, so latency is masked by the continuation of the current thread. The latter is done using lightweight multithreading—computation is divided into a set of threads, each of which runs to completion in finite time.

Message Aggregation: Some communication cannot be avoided, but its cost can be reduced by minimizing the number (as opposed to the volume) of messages by aggregating messages together. For machines like the Paragon and workstation networks, which have high communication start-up costs, this is very important. Many small messages are aggregated into one large physical message to amortize the overhead. Message aggregation can be performed by the programmer, by the compiler, semi-statically, in a separate preprocessing phase by the runtime system, or on-the-fly by the runtime system. Because Multipol supports asynchronous communication patterns, in which the message patterns are not known in advance, we do aggregation on-the-fly.

One-way Communication: The second technique for reducing communication cost is to avoid acknowledgement traffic. Acknowledgements may consume a significant fraction of available bandwidth when the messages are small. In the hash table, a factor of two in performance was gained when split-phase inserts with acknowledgements were replaced by batches of inserts followed by periodic global synchronization points.

3.4 Summary

The irregular application supported by Multipol represent some of the more challenging problems for parallelism. The library approach has proved to be a good compromise between hand-coded applications that are machine-specific, and general purpose compiler and language approaches that give too little control to the programmer. The Multipol design reflects the delicate balance between clean interfaces and performance. It makes several compromises for performance, such as split-phase operations and multi-ported access, but manages to hide many details of synchronization and communication that simplify the application programmer's job.

4 Dynamic Load Balancing: Multithreaded Approach

We present a multithreaded model for the dynamic load balancing of adaptive computations on distributed-memory MIMD machines and time-sharing clustered workstations. Our objective is to provide runtime software support that: (1) minimizes message and scheduling latencies, and (2) simplifies the tasks of designing, implementing and testing new load-balancing policies/methods for parallel adaptive PDE computations.

Traditional load-balancing methods for single-threaded computations under-utilize multiprocessor resources such as CPU, memory, and network. The load balancing of single-threaded computations is carried out in sequential phases that require global synchronizations [5], [6]. In contrast, load-balancing methods based on the proposed multithreaded model allow the concurrent execution of tasks required for load balancing —such as information dissemination, decision making, and data migration— with tasks required for the computation of the actual application.

In multithreaded systems, message and scheduling latencies are tolerated via concurrency within a processor. For parallel adaptive PDE computations we explore concurrency within a processor by decomposing the original problem into a number of simpler and smaller problems. The decomposition of the PDE problem is based on a hierarchy of data abstractions: *domains, blocks, subdomains and regions.* Regions are mapped to scalar objects called threads.

Threads execute in a loosely synchronous mode. Based on computation and synchronization requirements, threads are grouped into distributed objects: *strings, ropes*[1] *and nets.* Threads that correspond to regions of the same subdomain belong in the same string (see Figure 4). Threads of the same string execute the same code on different data (SPMD model). Strings that correspond to the subdomains of the same block belong in the same rope. Threads on different ropes might compute the solution for different PDEs or use different grid types and apply different solution methods. They may therefore have different computational requirements and synchronization points (MPMD model). Finally, ropes that correspond to blocks of the same domain and handle the computation associated with the same application belong in the same net (see Figure 4). For multidisciplinary applications a number of nets (one for each discipline) can be distributed over a LAN or WAN.

Processor work load is balanced by: (i) migrating threads from overloaded processors to underloaded ones that handle strings of the same rope, and (ii) migrating strings from overloaded processors to underloaded ones that handle ropes of the same net. The thread and string migration is non-preemptive and, therefore, instead of thread migration we perform data migration. Data are migrated so that subsequent communication for the actual parallel computation of the PDE solver is minimized. The policy for thread migration is based on a *consumer-initiated consumer/producer* paradigm [9]. That is, every processor (consumer), upon completion of computation, searches for neighboring overloaded processors that handle strings from the same rope. After an overloaded

[1]Similar abstraction is used in [7] and [8]

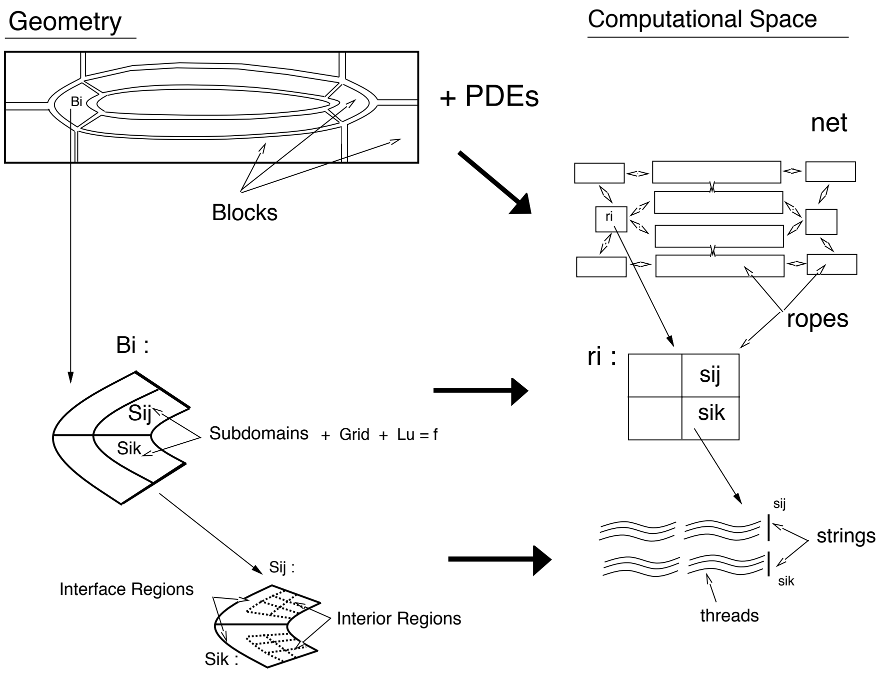

Figure 3: Global Data Structuring View

processor is identified (producer), it is interrupted by a consumer in order to migrate threads (data) to that consumer. A thread that is migrated from the producer to the consumer which initiated the request is likely to have data dependencies with other threads that already reside in the consumer's memory. The producer's decision to migrate a thread is based on priorities that are computed at runtime using a prespecified formula. Different formulae for the computation of thread priorities lead to different scheduling and load-balancing policies.

The multithreaded model simplifies the development of load-balancing algorithms by providing a simple mechanism to the application programmer to specify new load-balancing algorithms by merely specifying a formula that can be used by the system to compute the priorities for the execution (local or remote) of the threads. Moreover, using a consumer-initiated consumer/producer paradigm our model hides the overhead required for load-balancing by overlapping the execution of tasks required in load balancing with the execution of tasks performing the actual computation and communication required by the application.

5 Message Driven Objects, Modularity and Reuse

Message Driven Execution (MDE) is a mechanism useful for promoting modularity and efficiency. MDE is distinct from message passing. It can be thought of as a strategy for scheduling the processor. In this view, the entities associated with a processor include a collection of objects, a pool of messages, and a scheduler. Each message specifies the recipient object, a specific method defined for this object, and parameters to this method. The scheduler repeatedly selects a message from the pool and invokes the specified method. The method executes atomically, without interleaving. The objects may invoke methods on other objects within the same processor synchronously; and send messages (i.e. invoke methods on possibly remote objects), asynchronously, to objects on any processor. They may also specify creation of new parallel objects, which are created on a suitable processor by the runtime system.

Message driven execution promotes efficiency by automatically, transparently, and adaptively overlapping communication and computation. As computation is scheduled based on the presence of messages, a processor is never waiting for a particular message to arrive while other computations can go on. Performance benefits of this adaptive overlap are especially significant for irregular and dynamic problems. It is hard to predict, in general, when responses from remote processors, on which some local computation depends, will arrive. They may be unpredictably delayed by the communication subsystem; alternatively, the remote processor may not be able to schedule the computation that generates the response at a predictable time, due to other subcomputations on that processor. MDE deals with both kinds of uncertainties in an adaptive manner.

Another example of the utility of MDE is provided by reductions. A reduction, such as global sum—requires a long critical path dominated by multiple communication operations. Moreover, all processors must arrive at the reduction before its results are computed. In traditional systems all processors block at a reduction and wait for it to finish. Considerable processor resources are therefore wasted during this operation. This is further exacerbated if different processors arrive at the reduction at significantly varying times. In a message driven reduction, an object on each processor *deposits* its reduction value into the reduction object. Reduction object asynchronously computes the reduction. On every processor, other computations can take place while the reduction's critical path is being worked out. When the reduction is completed its results are broadcast to objects on all the processors, which can then resume their computation. Useful computation can thus be overlapped with the reduction operation. Multiple reductions can also be executed concurrently.

MDE also promotes modularity. In traditional systems, when a module is invoked, control is fully transferred to it. The idle times in such modules, which may arise due to load imbalances or critical paths, cannot be overlapped with other useful computation. With MDE one can initiate the message driven computations in a module at the time of its invocation. From then on, the computations will progress based on arrival of messages, and will be interleaved with computations in other modules. If one has to invoke two independent subcomputations, written in different modules, one can start them concurrently and overlap useful computations in one with idle times in the other. In traditional systems, when one is faced with efficiency loss due to inability to bring about such an overlap, a programmer would typically resort to combining the two modules into one and trying to manually program some degree of overlap. Clearly this destroys modularity. This is the sense in which MDE promotes modularity.

Charm [15, 16] is one of the first implemented parallel programming systems based on message driven execution. (The semantics of MDE had been explored earlier in the *Actors [14]* work.) Charm is designed to be efficiently portable across MIMD machines, with and without shared memory, and runs a diverse collection of platforms, including workstation networks, and multi-computers. In addition to its message-driven substrate, Charm supports MDE by providing a suite of alternate queing strategies for scheduling from the pool of available messages. Some of the strategies also support integer and bit-vector priorities. It supports dynamic creation of objects via a collection of dynamic load balancing strategies. Some of the load balancing strategies are useful for speculatively parallel computations, such as branch-and-bound in operations research, and state-space search. These strategies balance priorities, and memory needs in addition to load [18].

Some of the innovative concepts in Charm include branch office Chares and information sharing abstractions. A *branch office chare* [16]is an object with a branch on every processor. This abstraction turns out to be extremely versatile and is useful for specifying static decompositions, node specific functions (such as memory management) distributed services (such as load balancing), intermodule interfaces, and distributed data structures. Objects in Charm can *share information* not only via messages but also via a specific set of shared objects. A few modes in which information is commonly shared in parallel programs have been identified and provided as information sharing abstractions for this purpose. Charm is a relatively mature system with multiple tools and a growing

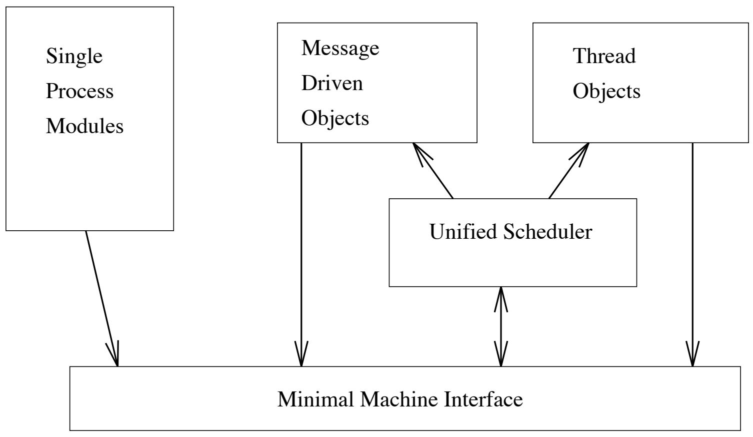

Figure 4: A Schematic Of the *Converse* Framework

set of libraries. The tools include one for performance analysis which not only provides a graphical display of performance but also carries out an expert analysis identifying specific performance bottlenecks at the source level and attempting to make suggestions for possible improvement. MDE, BOCs and dynamic load balancing make Charm highly suitable for supporting irregular problems and for incorporating asynchronous strategies such as those described in Section 3.

5.1 Multilinguual Interoperability

MDE promotes modularity and therefore reuse. However, there is another aspect to reuse. Many different languages and paradigms have evolved which are seen as beneficial in specific contexts. These approaches often complement each other. Shouldn't an application programmer be able to reuse modules written in multiple languages in their application? This is technically challenging when one considers that one must support paradigms with different views how a processor is scheduled; viz. message driven execution, threads and traditional SPMD style programming. Recognizing the benefits of such reuse, the Charm research group at Illinois is developing a framework for supporting multi-lingual interoperability, called *Converse* [17].

The above ways of scheduling a processor seem to constitute fundamental categories; all other approaches can be seen to fit in one or the other. Recognizing this, *Converse* supports threads and message driven objects through a unified scheduler (As shown in Figure 4). All three are supported by a standardized and minimal machine interface. *Converse* has a component based architecture rather than a uniform system architecture. Individual components are specified by their interfaces and multiple implementations of individual components are provided. The components include a scheduler, queueing strategies (for the scheduler), a machine interface, load balancing strategies, message managers, and thread objects. *Converse* framework will support linking together of modules written in Charm, Charm++, PVM (its message passing capabilities), nxlib (and possibly MPI,) multithreaded versions of the above, single-process as well as message driven implementations of HPF, message-passing thread packages such as Chant, parallel functional languages, etc. Individual modules written in approaches such as those described in this paper can then be used without committing the entire application to a particular language or paradigm.

6 Summary

The task of writing parallel programs, although complex, can be tackled successfully with the right set of tools.

Irregular parallel computations, which pose particular performance challenges, are likely to proliferate as the base of parallel processing broadens. Concurrency within a processor — the ability to schedule computations asynchronously and adaptively based on availability of data — helps deal with such irregularity. Tools based on such concurrency are therefore quite useful. Three different approaches based on this idea of concurrency were described. They use multithreading and message-driven objects as their basic mechanisms for producing adaptive concurrency.

Multipol is a library that supports irregular and dynamically distributed data structures. It uses split-phase operations to deal with unpredictability in the process of communication latencies. Multipol data structures include parallel trees, sets, queues, and graphs, with underlying infrasturcture for communication, caching objects, and synchronization.

A dynamic load balancing library for parallel adaptive PDE computations was described next. This library maps a hierarchy of data abstractions in PDES — domain blocks, sub-domains and regions — onto threads, strings, ropes and nets.

The paper also described Charm, a system based on message-driven objects. Messages here correspond to asynchronous invocations of methods on possibly remote objects. An object is scheduled for execution only when a message for it arrives; this mechanism provides Charm with its concurrency mechanism. Information sharing mechanisms, "branch office" chares, support for message scheduling strategies, prioritization, and dynamic object creation make Charm particularly suited for irregular applications. A variety of tools including visual performance display and analysis tools are available for Charm.

As different approaches, tools and languages are suitable in different (possibly overlapping) situations, it is desirable to provide a single framework that would support multiple approaches. Such a framework, *Converse*, was described next. *Converse* allows threads, message-driven objects and traditional single-process modules written in a variety of different languages to be linked in a single application. It is also a suitable language for exploring and developing new parallel languages, notations and libraries.

References

[1] "MPI: A Message-Passing Interface Standard," Message Passing Interface Forum, International Journal of Supercomputing Applications, Volume 8, Number 3/4, 1994.

[2] B. P. Miller, "What to Draw? When to Draw? An Essay on Parallel Program Visualization," Journal of Parallel and Distributed Computing, Volume 18, Number 2, June 1993, pp. 265-269.

[3] B. A. Myers, "The State of the Art in Visual Programming and Program Visualization," Report Number CMU-CS-88-114, Computer Science Department, Carnegie Mellon, 1988.

[4] G.A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V.Sunderam, "PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing," The MIT Press, 1994.

[5] S. R. Wheat, K. D. Devine, and A. B. Maccabe, Experience with automatic, dynamic load balancing and adaptive finite element computation, *Proceedings of the 27th Hawaii International Conference on Systems Sciences, January, 1994.*

[6] Ravi Ponnusamy, Yuan-Shin Hwang, Joel Saltz, Alok Choudhary, Geoffrey Fox, Supporting Irregular Distributions in FORTRAN 90D/HPF Compilers, University of Maryland, Department of Computer Science and UMIACS Technical Reports CS-TR-3268, UMIACS-TR-94-57

[7] Piyush Mehrotra and Matthew Haines, An overview of the OPUS language and runtime system, NASA CR-194921 ICASE Report No. 94-39 , Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23681-0001, May 1994.

[8] N. Sundaresan and L. Lee, An object-oriented thread model for parallel numerical applications. Proceedings of the 2n Annual Object-Oriented Numerics Conference - OONSKI 94, Sunriver, Oregon April 24-27 1994. Pages 291-308.

[9] N.P. Chrisochoides, Multithreaded Model For Communication and Load Balancing of Parallel Adaptive Computations On Multicomputers, SCCS-683, submitted to *Applied Numerical Mathematics Journal.*

[10] Soumen Chakrabarti and Katherine Yelick. Distributed data structures and algorithms for Gröbner basis computation. *Lisp and Symbolic Computation*, 1994.

[11] Jeff Jones. Parallelizing the phylogeny problem. Master's thesis, University of California, Berkeley, Computer Science Division, December 1994.

[12] Stephen Steinberg. Parallelizing a cell simulation: Analysis, abstraction, and portability. Master's thesis, University of California, Berkeley, Computer Science Division, December 1994.

[13] Chih-Po Wen and Katherine Yelick. Parallel timing simulation on a distributed memory multiprocessor. In *International Conference on CAD*, Santa Clara, CA, November 1993.

[14] G. A. Agha, Actors : A Model of Concurrent Computation in Distributed Systems, MIT press, 1986

[15] L. V. Kale and W. Shu, The Chare Kernel Language for Parallel Programming: A perspective. Department of Computer Science, University of Illinois, Tech. Report UIUCDCS-R-88-1451, August 1988

[16] L. V. Kale The Chare Kernel Parallel Programming Language and System, Proc. of Intl. Conf. on parallel processing, Vol. II, pp/ 17–25, 1990.

[17] L. V. Kale, Sanjeev Krishnan, N. Jagathesan, CONVERSE: An Interoperable Framework for Parallel Programming Technical Report, Dept. of Computer Science, University of Illinois at Urbana Champaign, March 1995.

[18] Amitabh Sinha and L.V. Kale", A Load Balancing Strategy for Prioritized Execution of Tasks, International Parallel Processing Symposium New Port Beach, CA, April 1993