In Proc. of the International Workshop on Parallel Symbolic Languages and Systems, Oct. 1995 (Springer)

Understanding Language Support for Irregular Parallelism

Mukund Raghavachari and Anne Rogers

Princeton University, Princeton NJ 08544
CS-TR-506-96

Abstract. While software support for array-based, data-parallel algorithms has been studied extensively, less attention has been devoted to irregular parallel applications. The majority of these applications are *unstructured*, that is, they possess asynchronous components that do not fit the data-parallel model. Examples of unstructured applications include sparse matrix and n-body problems. Previous research, such as Parti[11] and CHAOS[13], has concentrated on extending the array-based data-parallel model to handle structured irregular algorithms. For unstructured irregular applications, however, the appropriate language abstractions, compiler technology, and runtime techniques have yet to be discovered.

In this paper, we analyze, under a systematic framework, implementations of a representative set of algorithms— Blocked Sparse Cholesky (BSC)[20], Barnes-Hut[3], and EM3D[10]— on Tempest/Blizzard[18], a platform that supports both shared memory and message passing. Using our framework and corroborating experiments, we isolate a set of abstractions that allows easy and efficient expression of our benchmarks. The philosophy behind this set is to provide, for each component of a parallel language, mechanisms that support user control, along with good default behavior. Our contributions are the following: a framework for the evaluation of abstractions, a set of recommendations for language support for irregular applications, and an analysis of the ability of current languages to support them.

1 Introduction

While software support for array-based, data-parallel algorithms has been studied extensively, less attention has been devoted to irregular parallel applications.

The majority of irregular applications are *unstructured*, that is, they possess asynchronous components that do not fit the data-parallel model. Examples of unstructured applications include sparse matrix and n-body problems. Previous research, such as Parti[11] and CHAOS[13], has concentrated on extending the array-based data-parallel model to handle structured irregular algorithms. For unstructured irregular applications, however, the appropriate language abstractions, compiler technology, and runtime techniques have yet to be discovered. The emergence of machines, such as Alewife[1], FLASH[14], SHRIMP[15], and Tempest/Typhoon[18], that support the use of both shared memory and message passing in an application, further necessitates the development of new abstractions and techniques.

We take a methodical approach to the problem of language support for irregular applications. In this paper, we analyze, under a systematic framework, implementations of a representative set of algorithms — two unstructured irregular applications, Blocked Sparse Cholesky (BSC)[20], Barnes-Hut[3], and a structured irregular application, EM3D[10] — on Tempest/Blizzard[18], a platform that supports both shared memory and message passing. Using our experiments and analysis as a guide, we isolate an effective set of abstractions necessary for easy and efficient expression of our benchmarks. The philosophy behind this set is to provide, for each component of a parallel language, mechanisms that support user control, along with good default behavior. Our contributions are the following: a framework for the evaluation of language mechanisms, a set of recommendations for language support for irregular applications, and an analysis of the ability of current languages to support them.

Our framework divides the design space for parallel languages into two components: data model and task model.[1] We discuss the data model in three parts: name space, data locality management, and communication. We divide the task model into four topics: thread model, task creation, task allocation, and synchronization. For each component of our framework, we derive the set of mechanisms necessary for efficient expression of the algorithms and compare them with the capabilities of current languages. We validate our recommendations with experimental evidence from our three benchmarks. Table 1 summarizes our results; it lists our recommendations and compares them with the needs of our benchmarks. It also serves as a roadmap for our discussion.

Though the list of languages considered in this paper is far from exhaustive, we have attempted to choose a set of languages that represents the current state of parallel languages. We believe that for a parallel language to be accepted widely, it must be based on a familiar language, such as C, C++, or Fortran. Therefore, we only consider systems based on these languages. Due to space constraints, we offer only a brief summary of the languages mentioned in the paper in Appendix A. Table 2 summarizes the capabilities of current languages to provide the recommended abstractions. We discuss the table in some depth in Sect. 6.

The remainder of the paper is organized as follows: we give overviews of Tem-

[1] We do not consider the issue of I/O in this paper.

Table 1. Summary of recommendations for benchmarks. $\surd$ denotes recommended mechanism is used; $\star$ denotes that a less general abstraction suffices; $\mathcal{D}$ means that the default mechanism is used. A blank space denotes that the issue does not affect the application.

	Recom-	EM3D	BSC		Barnes-Hut		
	mendations		Blocks	Domains	Tree Build	Force Calc	Sec.
Data Model							§4
Naming	Global	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	§4.1
Data Locality							§4.2
Distribution	User Control	$\surd$	$\surd$	$\surd$			
Replication	Access Control (TSM[a] default)	$\surd$	$\star$	$\mathcal{D}$	$\surd$	$\surd$	
Communication							§4.3
	Bulk Transport	$\surd$	$\surd$				
	Asynchronous		$\surd$	$\surd$	$\surd$	$\surd$	
Task Model							§5
Thread Model	Multithreading	$\star$	$\surd$	$\star$	$\surd$	$\star$	§5.1
Task Creation	Asynchronous	$\star$	$\surd$	$\star$	$\surd$	$\star$	§5.2
Task Allocation	User Control (Data-Centric default)	$\mathcal{D}$	$\surd$	$\mathcal{D}$	$\surd$	$\surd$	§5.3
Synchronization							§5.4
	Barriers	$\surd$		$\surd$		$\surd$	
	Asynchronous		$\surd$		$\surd$		

[a] Transparent shared memory

pest and our three benchmarks in Sect.2 and Sect. 3; we discuss issues related to data and task models in Sect. 4 and Sect. 5; and we summarize our findings in Sect. 6.

2 Platform — Tempest

There has been a convergence in the architectures of massively parallel machines towards a set of workstation-like nodes linked by a fast interconnection network. An effect of this convergence, as is evidenced in the design of Alewife[1], FLASH[14], SHRIMP[15], and Typhoon[18], is that it has become possible to build machines that support both shared memory and message passing efficiently. We use a programming interface representative of this class of emerging architectures, Tempest[18], as the platform for our experiments.

Tempest supports the development of shared memory, message passing, or hybrid (i.e., combination) applications. It does not implement a shared address space directly, but provides four mechanisms that enable user-level software to do so: Active Messages[24], bulk data transfer, virtual memory management,

Table 2. Capability of current languages to support recommended abstractions. $\mathcal{D}$ denotes that only the default mechanism is available.

	Recommendations	Chaos	Cid	Comp. C++	Concert	COOL	Jade	Olden	Split-C
Data Model									
Naming	Global	✓	✓	✓	✓	✓	✓	✓	✓
Data Locality									
Distribution	User Control	✓	✓	✓		✓	✓	✓	✓
Replication	Access Control (TSM[b] default)	$\mathcal{D}$	$\mathcal{D}^a$	$\mathcal{D}$	$\mathcal{D}$	$\mathcal{D}$	$\mathcal{D}$	$\mathcal{D}$	
Communication									
	Bulk Transport		✓						✓
	Asynchronous	✓	✓	✓	✓	✓	✓	✓	✓
Task Model									
Thread Model	Multithreading		✓	✓	✓	✓		✓	
Task Creation	Asynchronous		✓	✓	✓	✓		✓	
Task Alloc.	User Control (Data-centric default)	$\mathcal{D}$	✓[c]	✓		✓		$\mathcal{D}$	
Synchronization									
	Barriers	✓							✓
	Asynchronous				✓	✓			

[a] Does not offer full shared-memory semantics.
[b] Transparent shared memory.
[c] Does not offer default data-centric behavior.

and fine-grain memory access control. Fine-grained memory access control allows the user to protect aligned memory blocks and have inappropriate accesses to blocks vectored to a user-level handler. By using active messages with fine-grained access control, a programmer may design customized protocols. Virtual memory management routines allow the use of local memory as a cache for shared data. Programmers can either manage the address space explicitly through application-specific protocols, or through standard protocol libraries. The default library uses Tempest's mechanisms to provide transparent shared-memory semantics. We use Blizzard[21], an implementation of Tempest on a Thinking Machines CM-5[4], for the experiments discussed in this paper. All experiments were run on 32 processors.

3 Applications

In this section, we provide an overview of our three applications: BSC, Barnes-Hut, and EM3D.

3.1 Blocked Sparse Cholesky

The Cholesky factorization of sparse, symmetric, positive definite matrices is an important component of many engineering applications. An algorithm for sparse Cholesky factorization that has been shown to have good theoretical and practical performance on parallel machines is *Blocked Sparse Cholesky* (BSC)[20]. We describe the algorithm and our implementation briefly. We refer readers interested in specific details to Rothberg's PhD thesis[20].

The basis of BSC is to partition the input matrix, based on its sparse structure, into *blocks*. There are two operations defined on blocks: *block update* and *block factorization*. A block can be factored once all updates to it have completed, the number of updates determined by the sparse structure of the matrix. Block updates induce a directed, acyclic, task precedence graph, where no block is completed until its ancestors are factored, and once a block is factored, no further updates to it will be made. The algorithm completes once all blocks have been factored.

An optimization that increases the performance of BSC on parallel machines is to divide a subset of the blocks into disjoint sets called *domains*. The domains are chosen so that they are independent, that is, they receive no updates from any other block or domain. At the onset of computation, domains can be factored concurrently. After a domain is factored, the series of block updates defined by its structure is performed. At the end of the domain-block update phase, the remaining factorizations and updates of blocks proceed as described in the previous paragraph. Of the stages in this algorithm, the block factorization and update phase is the most computationally intensive.

Task creation in BSC is handled by a task queue-based approach. In subsequent sections, we describe our experiments, using various communication and synchronization strategies, on BSC. The implementation with the best performance takes advantage of the flexibility of Tempest by using several communication strategies: transparent shared memory for domain references, bulk transport for block references, and Active Messages for task queue modifications.

3.2 Barnes-Hut

Barnes-Hut[3], simulates the evolution, over time, of bodies in a gravitational system. Each body is modeled as a point mass and exerts gravitational force on other bodies in the system. At each time step, the program computes new positions and velocities for all bodies. Using the observation that the force a body exerts on another falls rapidly with distance, the algorithm approximates the force exerted by a collection of bodies beyond a certain distance by the force exerted by a point mass at the center of mass of the collection.

Each iteration in the algorithm is composed of two phases. The first phase builds an oct-tree based on the locations of the bodies in 3-space. This tree is not balanced, it is deeper in regions that contain many bodies. The second phase consists of a force calculation, which computes new accelerations and velocities for the bodies, and a position update, which computes bodies' new positions.

To calculate the force on a particular body, the algorithm performs a depth-first traversal of the tree from the root. For interior tree nodes, known as cells, the algorithm first determines if the bodies in the cell are distant enough to be approximated as a point mass. If so, the point mass's contribution is computed and the traversal is terminated. If not, the algorithm traverses the cell's eight subtrees.

Our implementation of Barnes-Hut builds upon the implementation discussed in the second author's previous work on application specific protocols for user-level shared memory[12]. This version uses a specialized shared-memory protocol to eliminate unnecessary invalidations (and subsequent requests). We have modified this implementation to use a bottom-up tree building scheme that outperforms the standard lock-based method.

3.3 EM3D

Our third application, EM3D, models the propagation of electro-magnetic waves through objects in three dimensions [10]. The problem is formulated as a computation on a bipartite graph with directed edges from `E` nodes, which represent electric fields, to `H` nodes, which represent magnetic fields, and vice versa. In this paper, we base our discussion on implementations described in Falsafi *et al.*[12].

The main loop of EM3D computes the change in `E` and `H` values over time. In each iteration, new `E` values are computed from the weighted sum of neighboring `H` nodes, and then new `H` values are computed from the weighted sum of neighboring `E` nodes. The graph structure is invariant over the main loop. The best implementation uses a specialized shared-memory protocol and bulk transport to minimize communication costs.

4 Data Model

The most important issue in the design of a parallel language is its data model. The data model of a language can be divided into three components: name space, data locality control, and communication mechanisms. In this section, we argue that a global name space, user-control over data distribution, memory access control with default transparent shared-memory semantics, bulk transport, and asynchronous communication are useful in the implementation of irregular applications.

4.1 Name Space

The ability to refer to remote data using a global name space simplifies the expression of algorithms. All languages discussed in this paper support a global name space. In explicit message-passing systems, where processors may refer only to data in their local address spaces, programmers often emulate a shared name space[22].

4.2 Data Locality Management

To avoid excessive communication for repeated accesses to data, which may occur if caching is not supported or data is replaced in the cache, it is often necessary to control data distribution, replication, and replacement explicitly. While some researchers claim that data locality control is difficult to use effectively[2, 9], our experience is to the contrary. Managing locality of blocks in BSC is straightforward. The algorithm assigns a logical "home" to each block; each update to a block is allocated to the block's home processor. A natural distribution is to locate each block on its home processor. Since processors only request blocks to which no further modifications are to be made, blocks may be replicated safely. Finally, it can easily be determined when a block is no longer required and can be freed. While these data distribution, replication and replacement policies are evident to the programmer, systems that do not allow control over data locality, such as Concert[9], must use sophisticated compiler and runtime analysis to derive them.

Falsafi *et al.*[12] have shown that the performance of Barnes-Hut can be improved by controlling replication using fine-grained access control. The key insight to their implementation is that the set of remote particles accessed by a processor varies slowly over time. Using the access fault handlers, each processor records requests made to its data. Every iteration, processors update their particles and migrate copies of them to processors that had requested them in the past. As a result, processors suffer fewer read-misses during the force calculation part of the algorithm resulting in a 40% improvement over the best transparent shared-memory version. A similar protocol has been shown to improve the performance of EM3D as well.

For certain data structures, customized protocols are not necessary; one may wish to rely on a default policy such as transparent shared memory. For example, in BSC, a domain usually updates several blocks on a processor, each update making a small number of accesses, mostly non-contiguous, to the domain. A natural way of handling domain updates is to use shared memory — the underlying system acquires and caches necessary data. With explicit replication, the program would have to either replicate the whole domain locally, resulting in much unnecessary communication, or explicitly simulate shared memory. Using the default shared memory mechanism for domains and using explicit replication for blocks results in a performance gain of 16% over an implementation that replicates domains as needed.

Current languages vary in their support for data locality control. None of them, however, offer access control. Programmers must either used the fixed protocol provided by a system or explicitly emulate access control. In Split-C[10], the fixed protocol does not support caching of data, fetches and stores to shared data always entail communication. Cid[17], Compositional C++[8], and COOL[6], however, do provide an automatic replication mechanism, but programmers are restricted to the protocol provided by the system. Other languages, such as Jade[19], Olden[5] and CHAOS[13], provide the means of controlling data distribution, but do not allow programmers any control over replication. All caching is handled transparently by the system. Languages such as Concert[9] do not allow programmers any control over data locality since they believe that it is better handled by the compiler or runtime system.

4.3 Communication

Because of the unstructured nature of applications such as BSC and Barnes-Hut, it is difficult to implement them using synchronous primitives such as blocking sends/receives. An asynchronous mechanism, such as Active Messages, better expresses the communication patterns since it does not require the sender and receiver to rendezvous. In addition, a bulk transfer routine can often reduce the cost of accesses to remote data. Using bulk transfer to fetch blocks in BSC results in a 26% speedup over a transparent shared-memory version. Similarly, for EM3D, using bulk updates results in a performance improvement of 86% over sending single updates[12].

Few languages support bulk transfer, Cid and Split-C being the exceptions. Most languages, however, do provide an asynchronous communication mechanism. Languages that provide transparent shared memory fetch remote data asynchronously by either relying on the underlying hardware or by simulating shared memory in software. CHAOS, for example, simulates shared memory by relying on the compiler to convert the program into explicit message-passing code. Cid and Split-C do not provide an implicit mechanism, programmers must insert calls to asynchronous routines to acquire data.

4.4 Discussion

Our studies reveal that a global name space, memory access control with default transparent shared memory, bulk transport and an asynchronous communication mechanism are useful for easy and efficient expression of irregular applications. While most of these features are present in certain languages, no language provides abstractions for access control, an important mechanism. Currently, using access control in our applications requires programming in the “assembly language”, that is, the base-level of our platform, Tempest. For example, to implement customized protocols, the user must manage data structures at the granularity of pages. We are exploring a segmented-paging based approach that should alleviate this problem. Other researchers are developing systems that simplify the design of protocols[7]. With these enhancements, one can provide a high-level interface that would allow programmers easy use of access control.

5 Task Model

We divide the task model of a language into four components: thread model, task creation, task allocation, and synchronization. In this section, we argue that multithreading, asynchronous task creation, control over task allocation with default data-centric behavior, barriers, and asynchronous mutual exclusion are useful for the implementation of irregular applications.

5.1 Thread model

EM3D can be expressed well in the SPMD model. Our implementations of BSC and Barnes-Hut, however, use non-preemptive threads. In our experience, multithreading significantly simplifies the expression of unstructured applications. Other researchers find that multithreading helps in reducing the effects of delays due to remote accesses and synchronization[25]. It is unclear, however, whether irregular applications would perform better with preemptive or non-preemptive threads. In the future, we plan use our benchmarks to compare the performance and ease of programming of preemptive and non-preemptive multithreading.

SPMD languages, such as Split-C, follow the single thread per processor model. COOL allows multiple threads per processor, but a thread retains control until it completes or is blocked on synchronization. In Compositional C++ and Concert, several threads may run concurrently on a processor. To provide maximum flexibility, we believe that preemptive execution of threads should be supported. Programmers could specify non-preemptive execution in one of two ways. The programmer can set a flag that informs the compiler/runtime system that all tasks run until completion, or at task creation the user can specify whether the task should run atomically. In the latter case, the appropriate scheduling policy for integrating the execution of atomic tasks with preemptive tasks requires further study.

5.2 Task Creation

There are many mechanisms by which programmers specify parallelism in languages: `forall` loops, remote procedure call, fork, task queues, etc. The task creation mechanisms required by the three applications that we study vary. The semantics of `forall` loops naturally represent the parallelism in structured applications such as EM3D. In EM3D, during each iteration, calculations on nodes can be performed in parallel. This parallelism is expressed well by a `forall` loop over all nodes.

On the other hand, `forall` loop semantics are not sufficient for BSC. Consider the following example, conceptually similar to BSC:

```
for (i = 1; i < N; i++) {
    A_i = sqrt(A_i);              /* Square Root step */
    A_{B_i} += Calculate(A_i);    /* Update Step */
}
```

where $B_i = k$, for some k, $i < k \leq N$. Essentially, each iteration, i, of the code performs the square root of A_i and updates one element, A_k, where k is greater than i. The `Calculate` operation is the most time-consuming portion of the computation. An iteration, k, is dependent on all iterations i, for $i < k$, that update A_k and cannot be scheduled until the updates are completed. At the onset of computation, the iterations, k, for which there is no $B_i = k$, can be scheduled. In the best case, where $B_i = N$(for all i), every `Calculate` operation can execute in parallel, though the updates to A_N would have to be synchronized.

`forall` loops imply that all iterations could execute in parallel; because of the presence of loop-carried dependencies, this example cannot be expressed with a `forall` loop.

The two stages in Barnes-Hut have different requirements. The force calculation phase is structured, and as such, can be expressed well with a `forall` loop. The tree building phase, however, involves complex interactions between the nodes and the tree being built. Due to dependencies between the insertion of different nodes into the tree, `forall` loops cannot be used. Both BSC and the tree-building phase of Barnes-Hut create tasks in an unstructured fashion that is best suited for an asynchronous task creation mechanism. Synchronous mechanisms, such as remote procedure call, are inefficient because the processor creating a task does not require information from the processor executing it and the synchronization serializes task creation.

CHAOS supports loops with `forall` semantics, and hence, cannot handle BSC and Barnes-Hut well. Split-C is based on the SPMD model, that is, a single task is created on each processor at the beginning of computation. Subsequent task creation must be managed explicitly by the programmer. Other parallel languages offer various mechanisms for task creation: forking a thread (Cid), calling a parallel function (COOL, Compositional C++), remote procedure call, etc. There is natural hierarchy in the expressive power of task creation mechanisms. Remote procedure calls can be implemented in terms of fork-joins, which in turn are less expressive than asynchronous task creation mechanisms such as forks and task queues. We believe that by providing an asynchronous task creation mechanism such as task queues, a language provides maximum flexibility, while allowing other useful abstractions such as RPC and fork-join to be implemented as library routines using this mechanism.

5.3 Task Allocation

A parallel algorithm must specify two functions: a mapping $\mathcal{T} : Tasks \rightarrow P$ of tasks to processors, and a mapping $\mathcal{D} : Data \rightarrow P$ of data to processors. $\mathcal{T}$ is a many-to-one function, while $\mathcal{D}$ is a many-to-many function. An algorithm is *data-centric* if $\mathcal{T}$ depends on $\mathcal{D}$. While many such relationships are possible, the most common one is owner computes, $\mathcal{T}(task) = \mathcal{D}(data\ written\ to\ in\ task)$. We define all mappings that are not data-centric to be *task-centric*.

EM3D is a good example of data-centric task allocation; the data is divided among the processors using a partitioner and each processor operates on its data. For Barnes-Hut, we experimented with two task allocation schemes for the tree building phase. In the data-centric version, each processor is responsible for inserting the nodes it owns into the tree. Our task-centric approach is to assign a portion of the space to each processor, and each processor is responsible for particles in the space it owns. The task-centric version performs twice as well as the data-centric version because it allows trees to be built bottom-up, removing much of the serialization due to locks. Similarly, Singh *et al.* have shown that a task-centric scheme outperforms data-centric schemes for the force calculation phase[23]. In their data-centric scheme, each processor is allocated an equal number of bodies, and at every time-step, it is responsible for calculating the forces on its bodies. The task-centric version processes the structure of the oct-tree to determine the appropriate allocation of work to each processor.

The standard BSC algorithm uses owner computes — blocks and domains are associated with processors and every update is executed on the home processor of a block or domain. We have determined experimentally that even though each processor receives roughly the same number of blocks, this version suffers from load imbalance. At any instant, only a small subset of the blocks on a processor is active, that is, receiving updates. As a result, at any given time, processors have varying amounts of work, resulting in load imbalance. It is difficult to calculate, *a priori*, the work distribution as a function of time and to account for it during task allocation.

To alleviate the load imbalance, we experimented with other task allocation schemes. The method that has the best performance is for a processor to "give" a task to another processor when its task queue length reaches a certain threshold. We implemented two versions of this "work giving" algorithm. The task-centric approach gives a task to a random processor, the data-centric approach gives the task to one of the processors owning a source block used in the update. Our experiments show that the data-centric method has slightly better performance than the task-centric one, and is 4%-10% better than strict owner computes.

The task allocation schemes used by our three applications are widely different and it is difficult for a language that offers no control over task allocation, such as Concert, to derive these schemes. Languages that assign tasks to processors automatically are either data-centric or task-centric — Olden is data-centric, while Jade and Concert are task-centric. A SPMD language, like Split-C, offers only one task per processor, and therefore, the issue of task allocation does not arise. Other languages allow users to specify the processor a task should run on, but provide a default behavior as well. For example, in COOL, the default task allocation scheme is data-centric, but the programmer can override this choice. In keeping with our philosophy of offering maximum flexibility with good defaults, we believe that languages should supply a default data-centric behavior, but permit the implementation of task-centric schemes by supporting control over task allocation.

5.4 Synchronization

The role of synchronization in parallel programs is to control the order in which tasks access data. In EM3D, a barrier ensures that all updates to data have completed before the next iteration is started. BSC uses a barrier to confirm that all processors have initialized their data structures, and an atomic task queue modification routine. We implemented two versions of the task modification routine: the first, implements the routine as an Active Message handler, taking advantage of the fact that handlers run atomically; the second, uses shared-memory MCS locks[16] for mutual exclusion. The former version performs four times better than the latter. Our measurements show that only a small amount of time in the lock-based version is spent obtaining and releasing locks; the major cause of decreased performance is unnecessary serialization of tasks. Modifications to a task queue must be atomic but need not be ordered. Locks serialize accesses to task queues, and hence, serialize the creation of tasks. Barnes-Hut requires a barrier between iterations of the force calculation phase. The tree building phase uses locks to moderate insertions into the tree. Falsafi *et al.* have shown that message-passing locks performs two times better than MCS locks[12].

Our benchmarks require barriers and an asynchronous mutual exclusion mechanism for efficient implementation. Olden, a future-based language, provides a *touch* call that is used to ensure that a task has completed and that the return value has been written. Split-C provides memory barriers to synchronize split-phase memory operations. Jade requires programmers to provide the consistency requirements of tasks, and schedules a task only once its requirements are met. This approach can often to lead to unnecessary serialization in problems such as Barnes-Hut[19]. COOL allows a parallel function to be specified to be mutually exclusive and lets tasks synchronize using join variables. Concert allows data to synchronize by choosing which methods it will accept. Among these mechanisms, only COOL and Concert provide asynchronous mutual exclusion synchronization methods.

5.5 Discussion

The abstractions offered by a language are affected by its thread model. In SPMD languages, like Split-C and CHAOS, there is a single thread of control per processor, and hence, they do not offer mechanisms for task creation and allocation, and provide only forms of barrier synchronization. While structured applications are easily expressed under this paradigm, unstructured problems require more complex abstractions.

Our studies show that, for our benchmarks, an asynchronous task creation mechanism, control over task allocation with default data-centric behavior, barriers, and an asynchronous mutual exclusion mechanism form an effective set of abstractions that allow easy and efficient implementation. Using this set, other abstractions can be built as libraries. It is an open question, however, whether preemptive or non-preemptive multithreading is required.

6 Summary

In the previous sections, we have used three irregular applications — BSC, EM3D, and Barnes-Hut — to study the effects of various abstractions on the ease and efficiency of programming. Table 1 summarizes our findings. Using experimental evidence and analysis, we show that it is important to provide a global name space coupled with memory access control with a good default policy. Choosing the right communication mechanism for a data structure helps reduce communication costs. An asynchronous communication primitive along with bulk transfer, enables higher-level communication abstractions to implemented as libraries while providing programmers with maximum flexibility. The set of task abstractions used by our applications include an asynchronous task creation mechanism, control over task allocation with default data-centric behavior, barriers, and an asynchronous mutual exclusion mechanism.

Table 2 summarizes the capabilities of current languages to provide the recommended abstractions. Note that none of the languages provide memory access control. SPMD systems, such as CHAOS and Split-C, do not provide the necessary abstractions for task creation, allocation, or synchronization — the programmer must manage them explicitly. Concert allows no control over data and task placement. Olden allows programmers to influence task allocation through data distribution, but neither Olden nor Jade provides explicit task allocation or appropriate synchronization and communication mechanisms.

COOL and Compositional C++ allow programmers to create and map tasks explicitly and provide synchronization abstractions, but do not provide bulk transfer. Cid allows control over the granularity of data transfer and supports explicit task creation and allocation, but global pointers do not have the functionality of standard C pointers. They provide a means of naming global objects, but do not have duality with arrays, and cannot be dereferenced; the programmer must explicitly insert calls to request remote data. In these languages, however, the replication policy is hardwired into the implementation.

A data model with semantics similar to the Tempest interface shows considerable promise as a base for a language for irregular applications. In this model, programmers may experiment with several caching mechanisms to find one well-suited to an application. When the replication policy is difficult to express explicitly, programmers can use the default coherence protocol to manage locality. Tempest, however, offers no support for any of the task model mechanisms we find necessary.

Massively parallel machines present scientists with the exciting opportunity to run applications previously thought to be too resource-intensive. Widespread acceptance, however, is being hindered by the daunting nature of programming on these machines. While it is imperative that the burden on the programmer be lightened, one must recall that the main reason for using massively parallel machines is to achieve high performance; programmers should be able to express application-specific algorithms that are essential for efficient execution. In language design, however, flexibility and ease of programming are, to an extent, contradictory goals. We believe that a good compromise is to offer reasonable default behaviors for language abstractions, but provide the means by which a programmer may override this behavior with application-specific routines. While some languages follow this principle for certain abstractions, we believe that it should apply to all aspects of programming.

Acknowledgements

We wish to thank the Wisconsin Wind Tunnel(WWT) project, led by M. Hill, J. Larus, and D. Wood, for providing access to Blizzard and their CM-5. We wish to thank the SPLASH group at Stanford for supplying an implementation of blocked, sparse Cholesky, and the WWT group for the Tempest versions of EM3D and Barnes-Hut. We would like to thank Andrew Appel, Martin Carlisle, Doug Clark, Jim Larus, John Reppy, and J.P. Singh for their helpful comments. Mukund Raghavachari was supported, in part, by ARPA grant N00014-91-J-4039 and NSF grant MIP-9420653.

References

1. A. Agarwal et al. The MIT Alewife machine. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*, 1991.
2. H. Bal and F. Kaashoek. Object distribution in Orca using compile-time and run-time techniques. In *OOPSLA*, Sept. 1993.
3. J. Barnes and P. Hut. A hierarchical O(N log N) force calculation algorithm. *Nature*, pages 446–449, Dec. 1986.
4. C. Leiserson et al. The network architecture of the connection machine CM-5. In *Symposium on Parallel and Distributed Algorithms*, pages 272–285, June 1992.
5. M. Carlisle and A. Rogers. Software caching and computation migration in olden. *Principles and Practice of Parallel Programming*, July 1995.
6. R. Chandra, A. Gupta, and J. Hennessy. Data locality and load balancing in COOL. In *Principles and Practice of parallel programming*, May 1993.
7. S. Chandra, B. Richards, and J. Larus. Teapot: Language support for writing memory coherence protocls. Technical Report TR 1291, Computer Science Department, University of Wisconsin, Madison, WI, Oct. 1995.
8. M. Chandy and C. Kesselman. A declarative, concurrent object oriented programming notation. Technical Report CS-92-01, California Institute of Technology, Pasadena, Ca, 1992.
9. A. Chien, V. Karamcheti, and J. Pleyvak. The concert system – compiler and runtime support for efficient, fine-grained concurrent object-oriented programs. Technical Report UIUCDCS-R-93-1815, Dept of Computer Science, University of Illinois(UC), 1993.
10. D. Culler et al. Parallel programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, Nov. 1993.
11. R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. Technical Report CS-TR-3163, University of Maryland, Oct. 1993.
12. B. Falsafi, A. Lebeck, S. Reinhardt, I. Schoinas, M. Hill, J. Larus, A. Rogers, and D. Wood. Application-specific protocols for user-level shared memory. In *Proceedings of Supercomputing '94*, pages 380–389, Nov. 1994.
13. Y.-S. Hwang, B. Moon, S. Sharma, R. Ponnuswamy, R. Das, and J. Saltz. Runtime and language support for compiling adaptive irregular programs on distributed memory machines. *Software — Practice and Experience*, 25, June 1995.
14. J. Kuskin et al. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, Apr. 1994.
15. M. Blumrich et al. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 142–153, Apr. 1994.

16. J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, Feb. 1991.
17. R. S. Nikhil. Cid: A parallel, shared-memory C for distributed-memory machines. In *Languages and Compilers for Parallel Computing*, pages 376–390, Aug. 1994.
18. S. Reinhardt, J. Larus, and D. Wood. Tempest and Typhoon: user-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, Apr. 1994.
19. M. Rinard. *The design, implementation and evaluation of Jade: a portable, implicitly parallel programming language*. PhD thesis, Stanford University, Aug. 1994.
20. E. Rothberg. *Exploiting the memory hierarchy in sequential and sparse Cholesky factorization*. PhD thesis, Stanford University Department of Computer Science, Jan. 1993.
21. I. Schoinas, B. Falsafi, A. Lebeck, S. Reinhardt, J. Larus, and D. Wood. Fine-grain access control for distributed shared memory. In *Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–307, Nov. 1994.
22. J. P. Singh, J. L. Hennessy, and A. Gupta. Implications of hierarchical n-body methods for multiprocessor architecture. *ACM Transactions on Computer Systems*, May 1995.
23. J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. L. Hennessy. Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multipole, and radiosity. *Journal of Parallel and Distributed Computing*, June 1995.
24. T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: a mechanism for integrated commmunication and computing. In *Proceedings of the International Symposium on Computer Architecture*, pages 256–266, May 1992.
25. K. Yelick, S. Chakrabarti, E. Deprit, J. Jones, A. Krishnamurthy, and C.-P. Wen. Parallel data structures for symbolic computation. In *Workshop on Parallel Symbolic Languages and Systems*, Oct. 1995.

A Overview of the parallel languages mentioned in the paper

CHAOS[13]	A runtime library based system, designed for structured irregular problems. Using the data distribution and loop iteration partitioning provided by the user, CHAOS remaps data to the new distributions, performs loop preprocessing to translate global references into local references, and generates communication schedules for exchanging data between processors. Data structures must be expressed in terms of arrays.
Cid[17]	Cid is a superset of C that supports a global address space. Remote objects are manipulated by forking a remote thread or with asynchronous remote-access calls. Programmers can control object placement, thread placement, and thread granularity. Data transfer is performed at object-granularity. Implemented using a preprocessor and runtime system.
Compositional C++[8]	A concurrent object-oriented language consisting of six extensions to C++: parallel block, spawn, atomic functions, logical processors, global pointers, sync variables. Remote procedure call is a primitive in this language.
Concert[9]	A fine-grained concurrent object-oriented language. Each object lies in global shared memory, is single-threaded, and communicates with other objects by asynchronous invocations. Synchronization constraints are enforced by selective processing of messages. Programmers control concurrency and the enabling of computations; location, storage, and detailed scheduling are left to the compiler and runtime system.
COOL[6]	In COOL, an extension of C++, parallelism is expressed by declaring functions to be *parallel*. Operations on objects may be specified to be mutually exclusive. Allows programmers to locate and migrate objects and provide affinity hints that the runtime system uses to map tasks.
Jade[19]	Starting with a sequential program, a programmer annotates certain portions of the code with data-usage information. The implementation uses the program's serial order and the data-usage requirements to decide when tasks can be launched safely.
Olden[5]	An extension of C designed for recursive data structures. The programmer allocates data to the processors, and depending on the locality of data, the runtime system decides whether to migrate threads or execute them locally, caching data as needed.
Split-C[10]	An extension of C that follows the SPMD model. Global pointers address the global address space while standard pointers address the portion owned by the accessing processor. Offers support for split-phase memory operations and bulk transfer.

This article was processed using the LaTeX macro package with LLNCS style