A Hierarchical Load-Balancing Framework for Dynamic Multithreaded Computations

Vijay Karamcheti
Department of Computer Science
New York University
vijayk@cs.nyu.edu

Andrew A. Chien
Department of Computer Science
University of Illinois at Urbana-Champaign
achien@cs.uiuc.edu

Abstract

High-level parallel programming models supporting dynamic fine-grained threads in a global object space, are becoming increasingly popular for expressing irregular applications based on sophisticated adaptive algorithms and pointer-based data structures. However, implementing these multithreaded computations on scalable parallel machines poses significant challenges, particularly with respect to load-balancing. Load-balancing techniques must simultaneously incur low overhead to support fine-grained threads as well as be sophisticated enough to preserve data locality and thread execution priority.

This paper presents a hierarchical framework which addresses these conflicting goals by viewing the computation as being made up of different thread subsets, each of which are load-balanced independently. In contrast to previous processor-centric approaches that have advocated the use of a uniform policy for load-balancing all threads in a computation, our framework allows each thread subset to be load-balanced using a policy most suited to its characteristics (e.g., locality or priority sensitivity). The framework consists of two parts: (i) language support which permits a programmer to tag different thread subsets with appropriate policies, and (ii) run-time support which synthesizes overall application load-balance by composing these individual policies.

This framework has been implemented in the Illinois Concert runtime system, an execution platform for fine-grained concurrent object-oriented languages. Results for four large irregular applications on the Cray T3D and the SGI Origin 2000 demonstrate advantages of the hierarchical framework: performance improves by up to an order of magnitude as compared to using a uniform load-balancing policy.

1 Introduction

Irregular parallel computations are increasingly being encountered in applications such as molecular dynamics, particle simulations, and visualization, spanning a number of engineering and scientific domains. State-of-the-art techniques for these computations rely on sophisticated adaptive [2] and tree-based algorithms [1, 15], and pointer-based data structures such as linked lists, trees, and sparse matrices. High-level programming models supporting dynamic thread creation and multithreading [17, 4, 26, 7, 20, 12] enable convenient expression of such computations by relieving the programmer from having to manage details of thread and data structuring. These models, exemplified by several concurrent object-oriented [16, 34, 10] and message-driven [20] languages, permit application concurrency to be expressed in terms of arbitrary user-defined computation units and simplify implementation of pointer-based data structures by providing a global name space.

However implementing these computations on scalable parallel machines is challenging because of the irregular thread structure and unpredictable access pattern inherent in application algorithms and data structures. One of the key challenges is ensuring load balance in the presence of three characteristics: *wide* *variations in thread granularity*, *locality sensitivity*, and *priority sensitivity*. The first two characteristics arise from user-specified threads operating in a global object name space, while the third is common to search applications where prioritization of thread execution is required to avoid redundant work. The implication for load-balancing techniques is that they must simultaneously incur low-overhead to support fine-grained threads, and be sophisticated enough to respect locality- and priority-sensitivity. Unfortunately, load-balancing solutions in literature do not do a good job of meeting these requirements: most are either too heavyweight for use with fine-grained threads [12, 17, 23, 21, 7], completely ignore locality [4, 24], or work well only with regular computations [7, 26].

In this paper, we describe a general load-balancing framework for systems with all of the above characteristics. Our framework views the computation as a collection of different thread subsets, each of which can be load-balanced independently. The conflicting objectives of low-overhead sophisticated techniques are satisfied by allowing each thread subset to be load-balanced according to the policy most suitable for its characteristics. Overall application load-balance is achieved by composing these individual policies at run time. Our approach represents a significant departure from traditional processor-centric approaches, which employ a single uniform policy to load-balance all threads, and consequently are unable to meet the requirements of fine-grained multithreaded computations. Structurally, our framework consists of two components:

1. Language support to associate a subset of threads with a specific load balancing policy, and
2. Run-time support to efficiently synthesize overall application load balance by composing multiple load balancing policies within the same application.

Language support consists of object- and thread-creation annotations that permit the programmer to specify how a given subset of threads will be load-balanced. Run-time support defines a custom-scheduler interface that enables the construction of application-specific load-balancing policies, and integrates them with the rest of the language implementation.

This framework has been implemented in the Illinois Concert System [9, 8], an execution platform for concurrent object-oriented languages on scalable parallel platforms. Studies using four large irregular applications—IC-Cedar, Hierarchical Radiosity, Gröbner, and Phylogeny—show the advantages of using the framework. Each application shows a significant performance improvement on both our target platforms, the Cray T3D and the SGI Cray Origin 2000, often improving performance over using a single uniform load-balancing policy by up to an order of magnitude. Our results show that complex irregular applications with varying thread behaviors can be load-balanced by viewing them as a composition of individual thread subsets, each of which exhibit simpler behaviors.

The rest of this paper is organized as follows. Section 2 introduces concurrent object-oriented languages and their implementation. The load-balancing framework is described in Section 3 and evaluated using application studies in Section 4. Section 5 describes related efforts to place our work in context, and we conclude in Section 6.

2 Background

Concurrent object-oriented programming models express computation as interactions among a set of autonomous communicating entities (objects). Objects reside in a globally shared name space and communicate via method invocations. We consider a specific language, Illinois Concert C++ (ICC++) [11] which extends C++ with concurrent statements, concurrent objects, and multiaccess object collections. *Concurrent statements* express non-binding concurrency by specifying a partial order among program statements (typically, method invocations) and are declared by annotating standard blocks and loops with the `conc` keyword: statements within a `conc` block can be executed concurrently as long as local data dependences are preserved. *Concurrent objects* provide a consistent interface in a concurrent environment and are implicitly supported by requiring that method invocations on an object execute as if they had exclusive access. *Multiaccess object collections* support large-scale concurrency by defining an indexable set of element objects, each of which are aware of the collection. ICC++ extends the C++ array syntax to support collections, providing predefined methods which allow collections to implement an aggregate behavior and interface.

ICC++ programs are executed using the Illinois Concert System [8, 9], a high-performance compiler and run-time system for implementing concurrent object-oriented languages on scalable parallel platforms. The Concert compiler employs global interprocedural analyses and transformations such as method and object inlining, and procedure cloning to successfully eliminate overheads arising from the high level object-oriented nature of the programming model. The run-time system synthesizes the global object name space by mapping the communicating objects across the machine, enforces object-level concurrency control, and interleaves processing resources among dynamically created concurrent activities (threads). The default object creation policy evenly distributes top level objects (e.g., collections) by allocating them on random nodes, with all other objects allocated locally to enhance data locality. Threads execute, by default, on the processor where the target object (of the corresponding method invocation) resides. Multiple threads mapped to the same processing node are managed by a FIFO scheduler.

3 Load-Balancing Framework

Our strategy for load balancing an irregular computation decomposes the overall problem into several subproblems, each of which ensures load-balance for a subset of threads. To construct thread subsets, we group all threads created at the same static *thread creation site* in the source program. This decomposition allows different threads to be controlled by different policies, based upon considerations of thread granularity, irregularity, and required sensitivity to locality and priority. Such policies can broadly be classified into three categories: object-placement, thread-placement, and thread-ordering policies.

Object-placement policies distribute threads by controlling the location of data objects and represent a low-cost solution to the load-balancing problem because thread distribution incurs low decision and scheduling overheads. *Thread-placement* policies enable thread distribution independent of object placement, and can additionally be sensitive to data locality, albeit with increased cost. *Thread-ordering* policies incur higher cost than thread placement, providing load balance while prioritizing thread execution. An example of our overall load-balancing strategy is depicted in Figure 1 using three sets of threads, shown using white, cross-hatched, and black circles: the white threads use object placement while the others rely on explicit thread-placement policies.

Realization of this strategy requires two components:

1. Language support to associate a set of threads with a specific load-balancing policy, and
2. Run-time system support to synthesize application load balance by composing multiple load-balancing mechanisms within the same application.

These two components enable complex irregular applications with varying thread behaviors to be viewed as the composition of individual thread subsets, each of which exhibit simpler behaviors. We describe each component in detail below.

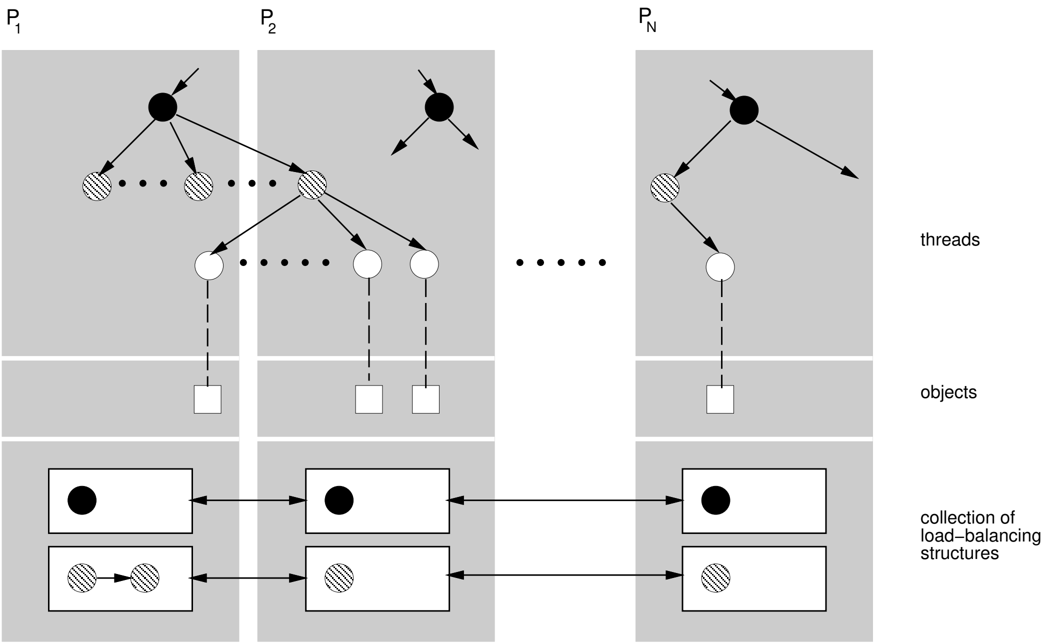

Figure 1: Overall load-balancing strategy: Different subsets of threads (shown by the white, cross-hatched, and black circles) are subject to different load-balancing policies.

3.1 Language support: Associating load-balancing policies

Language support identifies the set of threads that will be subject to a particular load-balancing policy and creates the association. We discuss in turn, associating threads with object placement, thread placement, and thread-ordering policies:

Object placement policies indirectly balance the threads created at a program site by first controlling the placement of a distinguished data object, and then ensuring that the thread executes local to this object. In ICC++, these policies control placement of individual objects and distribution of object collections, and are specified using the standard placement syntax of the `new` operator. The following code fragment shows examples of three object placement and collection distribution policies:[1]

```
Foo *obj1 = new ( LOCAL ) Foo(4);                 // object placement
Foo *obj2 = new ( location_of(obj1) ) Foo(5);    // object collocation
Accum *coll1 = new ( BLOCK ) Accum[16];          // collection distribution
```

Thread placement policies directly control the balanced distribution of threads belonging to a particular subset. In ICC++, such policies are specified by annotating thread creation sites using the `!scheduler` keyword which takes as an argument the specific load-balancing policy. For example, in the following code fragment, threads corresponding to `thread_function` invocations are balanced using a `WorkSharing` scheduler which distributes threads upon creation to random targets:

```
...
( thread_function( arg1, arg2, ... ) !scheduler WorkSharing );
...
```

[1]We assume the existence of an implementation-specific function, `location_of`, which provides the run-time location of its argument.

Thread ordering policies directly control both the processor where threads belonging to a particular subset execute, and the order of thread execution. In ICC++, such policies are specified by extending the thread placement annotation with the `!priority` keyword which takes an integer argument specifying the thread priority. In the following code fragment, threads corresponding to `thread_function` invocations execute with priority `arg1` and are managed by a `WorkSharingPriority` scheduler. This scheduler achieves load balance by distributing threads upon creation to random targets, and preserves thread priorities by enqueuing these threads into per-processor priority queues.

```
...
( thread_function( arg1, arg2, ... ) !priority arg1
                                     !scheduler WorkSharingPriority );
...
```

3.2 Run-time support: Synthesizing application load balance

Run-time support synthesizes overall application load balance by composing the individual load-balancing policies specified for each thread subset. Support for individual object placement, thread placement and thread-ordering policies in the Concert run-time system builds upon the default object and thread creation policies described in Section 2.

Object-placement policies adhere to the default thread-placement policy, so their integration into the Concert run-time system only requires the capability of creating objects on arbitrary processing nodes.

On the other hand, explicit thread-placement and thread-ordering policies require additional support, provided by a *custom scheduler framework*. This framework augments the default FIFO scheduler of the Concert run-time system with a hierarchy of application-specific thread schedulers as shown in Figure 2. On creation, each thread is enqueued into the appropriate scheduling structure. Each nonempty custom scheduler is periodically assigned scheduling quanta by the FIFO scheduler. The custom scheduler uses these quanta to execute the set of enqueued threads, as well as to redistribute threads across the machine according to a specific load-balancing policy. Coexistence between different custom schedulers is ensured by a contract of operation (see below) which prevents any individual scheduler from receiving an inequitable share of processing resources.

Interactions between the custom scheduling hierarchy and the default FIFO scheduler are concretely specified in terms of the following three components: (i) the interface for adding custom schedulers to build the hierarchy, (ii) the generic interface that each such scheduler must subscribe to, and (iii) the contract between the scheduler implementation and the run-time system. In the following code fragments, `Scheduler` and `CustomScheduler` refer to the default and custom scheduling structures respectively.

Adding custom schedulers. Custom schedulers are added into the scheduling hierarchy using the following functions:

```
void init_custom_schedulers( Scheduler *parent );
void attach_scheduler( Scheduler *parent, CustomScheduler *sched );
```

The `init_custom_schedulers` function is a hook in the application program to allow specification of one or more custom schedulers. Each scheduler is added into the hierarchy using the `attach_scheduler` function. The order of calls to `attach_scheduler` specifies a linear priority ordering among the schedulers: The default FIFO scheduler has the highest priority.

Custom scheduler interface. Each custom scheduler implementation must provide the following functions which are invoked as required by the run-time system:

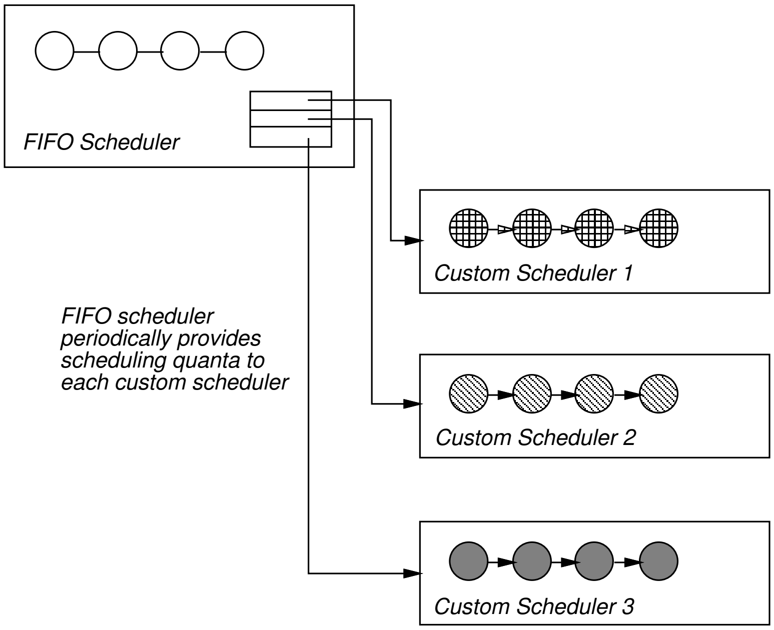

Figure 2: The custom scheduler framework augments the default FIFO scheduler with a set of application-specific custom schedulers, each of which manages a set of threads according to a different load-balancing policy.

```
void enqueue( CustomScheduler *sched, ... );
int  alive( CustomScheduler *sched )
int  size( CustomScheduler *sched );
void dispatch( CustomScheduler *sched, Scheduler *parent );
int  quiescence_callback( CustomScheduler *sched );
```

The `enqueue` function is invoked to enqueue a given thread for distribution according to the designated load-balancing policy. The remaining functions serve as a handshake for implementing the contract discussed below. The `alive` function returns TRUE if the scheduler is active, FALSE otherwise. The `size` function returns the number of threads queued in the scheduler that are awaiting processing resources. A call to `dispatch` grants a scheduling quantum to the scheduler. Finally, the run-time system invokes the `quiescence_callback` function for a custom scheduler each time it detects quiescence.

Contract of operation. The contract between the run-time system and a custom scheduler implementation captures the operational behavior of each component. The run-time system guarantees that it will periodically call the `dispatch` function for each active scheduler, both when the processor has work in its local FIFO queue, and when it is idle. The latter call can be used by the scheduler implementation to request more work from its counterparts on other processors. The scheduler implementation in turn is required to process threads in the lower levels of the scheduling hierarchy every time it dispatches a thread from its queue.

The other aspect of the contract pertains to quiescence processing. The run-time system detects quiescence when all FIFO queues and custom schedulers in the system are idle. The run-time system invokes the `quiescence_callback` function for each active custom scheduler whenever quiescence is detected. Thus, flexible scheduling strategies can be set up across program phases with different sets of schedulers (i.e., load-balancing policies) being active in different phases.

3.3 Example: A work-stealing scheduler

Figure 3 demonstrates use of the custom scheduler framework described above in the context of a work-stealing load-balancing policy. The scheduler enqueues threads on the local processor, dispatching them using scheduling quanta obtained from the FIFO scheduler. Load-balance is ensured using a work-stealing policy. An idle processor requests additional work from its counterpart on another processor, which either responds with work, or forwards the request to yet another processor.

The code fragments in Figure 3 highlight these interactions between the work-stealing scheduler and the rest of the run-time system. The *scheduler structure* contains two kinds of fields: those that implement the thread queue, and those that are used for implementing the work-stealing load-balancing policy. The *helper functions* initialize the scheduler structure and serve as handlers for messages requesting additional work, and for messages containing the response (task contents). Scheduler counterparts communicate using an active messages paradigm [30], which combines a message send with the invocation of a user-handler at the destination. The Concert run-time system is built on top of the Illinois Fast Messages communication library [25]: communication operations in Figure 3 are identified by the `FM_*` prefix. The **steal_request** handler receives steal requests, and either responds to them with the contents of a task, or forwards the request to the next processor. The **steal_response** handler enqueues the task into the local queue. The *interface functions* implement the contract between the run-time system and the scheduler. Note that we have not included the definitions of the `alive` and `size` functions, which just return the value of the corresponding fields in the scheduler structure. The **enqueue** function enqueues the thread into the local scheduler structure. The **dispatch** function is invoked by the run-time system periodically and is used by the scheduler to execute the queued threads. In addition, the scheduler uses the dispatch call to poll the network for steal requests (`FM_extract`), and obeying its contract, to process threads in the higher priority schedulers. When the local queue is empty, the **dispatch** function requests work from the counterpart on a neighboring processor. The *quiescence_callback* function, invoked when the run-time system detects quiescence, disables the scheduler. Instantiation of the scheduler is shown in the **init_custom_schedulers** function which allocates the scheduler structure, initializes it, and attaches it into the scheduler hierarchy on each processor.

As this example shows, the custom scheduler framework enables the convenient implementation of a wide range of load-balancing policies. In fact, all policies used in Section 4 follow the same basic structure as the work-stealing policy. The example also shows that although the overall run-time scheduling and load-balancing overheads are a function of the specific policy, use of the framework for expressing this policy itself incurs negligible run-time overheads. Thread enqueuing into a custom scheduler incurs the cost of an indirect function call to vector into the appropriate enqueue function. Thread dequeuing and dispatching can be amortized over multiple threads.

4 Evaluation of the Framework

We evaluate the advantages of the load-balancing framework using four parallel applications, each of which is irregular in one or more of data structure, parallel control structure, and concurrency. Table 1 lists these applications, which span a variety of computational domains. Figure 4 shows the distribution of application thread granularities for a single-processor execution on the Cray T3D, indicating that all applications exhibit wide variations in thread granularity and, with the exception of Gröbner, contain a large number of small granularity threads (50-500μs). Thus, with traditional processor-centric approaches, the computation can only be load-balanced using a low-overhead policy which can support such fine-granularity threads. All applications are also characterized by data access patterns which evolve during the computation, and two of

```
/*  SCHEDULER STRUCTURE */

typedef struct wstl_t {
  /*  task queue */
  ...
  int    alive, size, steal_pending;
  struct wstl_t* *sibling;
} WStlScheduler;


/*  HELPER FUNCTIONS */

void
initialize( WStlScheduler *sched ) {
  sched->alive = TRUE; sched->size = 0;
  sched->steal_pending = FALSE;
  /*  initialize task queue */
  ...
  /*  set up sibling array */
  sched->sibling = (WStlScheduler **)
    malloc( NPROCS*sizeof(WStlScheduler *) );
  FM_send_and_receive_bcast( PROC_ID,
    sched, (void **)sched->sibling );
}

void
steal_request( WStlScheduler *sched,
                     int req ) {
  int  nproc;

  if ( sched->size > 1 ) {
    /*  dequeue task and decrement size */
    ...; sched->size-- ;
    /*  send task contents to requester */
    FM_send( req,
      (FM_handler *)&steal_response, ... );
  }
  else {
    /*  forward request to next neighbor */
    nproc = (PROC_ID+1)%NPROCS;
    FM_send_4( nproc,
      (FM_shandler *)&steal_request,
      sched->sibling[nproc], req, 0, 0 );
  }
}

void
steal_response( ... ) {
  /*  extract scheduler pointer and task
      contents from the message buffer */
  ...;
  /*  reset flag  */
  sched->steal_pending = FALSE;
  /*  enqueue task and increment size */
  ...; sched->size++ ;
}

/*  INTERFACE FUNCTIONS */

void
enqueue( WStlScheduler *sched, ... ) {
  /*  enqueue task and increment size */
  ...; sched->size++ ;
}

void
dispatch( WStlScheduler *sched,
              Scheduler *parent ) {
  int  nproc;

  while ( sched->size > 0 ) {
    /*  dequeue, decrement size, and
        execute task */
    ...; sched->size--; ...;
    /*  respond to steal requests */
    FM_extract();
    /*  go back and process any FIFO tasks */
    dispatch( parent );
  }
  /*  idle: send request to proc_id+1  */
  if ( !sched->steal_pending ) {
    sched->steal_pending = TRUE;
    nproc = (PROC_ID+1)%NPROCS;
    FM_send_4( nproc,
      (FM_shandler *)&steal_request,
      sched->sibling[nproc], PROC_ID, 0, 0 );
  }
}

void
quiescence_callback(WStlScheduler *sched ) {
  /*  disable scheduler  */
  sched->alive = FALSE;
}


/*  SCHEDULER INSTANTIATION */

void
init_custom_schedulers( Scheduler *par ) {
  /*  allocate and initialize
      WStlScheduler structure */
  WStlScheduler *wsched = (WStlScheduler *)
    malloc( sizeof(WStlScheduler) );
  initialize( wsched );
  /*  attach to scheduler hierarchy  */
  attach_scheduler( par, wsched );
}
```

Figure 3: A work-stealing load-balancing policy expressed using the custom scheduler framework.

the applications (Gröbner and Phylogeny) require prioritization of thread execution for efficiency.

APPLICATION	DESCRIPTION	INPUT
IC-Cedar	protein dynamics [35] simulates protein macromolecular dynamics.	myoglobin
Radiosity	computer graphics [33] computes illumination using the hierarchical radiosity method.	room
Gröbner	computational algebra [6] computes Gröbner basis.	ae-4
Phylogeny	evolutionary history [19] determines the evolutionary history of a set of species.	prim.40

Table 1: The irregular applications suite.

To evaluate the advantages of the load-balancing framework, we measure the performance achieved on the Cray T3D and the SGI Origin 2000 for two versions of each application: one employing a single uniform (default) load-balancing policy for all threads, and the other where different thread subsets were balanced using different policies. To support the fine-granularity threads present in each application, we rely on a default object-placement policy that incurs the least decision and scheduling overheads. This policy evenly distributes top-level objects and object collections across the machine, with all other objects allocated locally to enhance data locality. Threads execute, by default, on the processor where the target object of the corresponding method invocation resides. On the other hand, the choice of the application-specific load-balancing policies was guided by the distinct characteristics of thread subsets in each application. Figure 5 shows the application performance (expressed as speedup with respect to the single-processor execution time) on both platforms.[2] As can be seen, each application shows a significant performance improvement, improving performance over using a single uniform load-balancing policy by up to an order of magnitude. Moreover, each application achieves performance comparable to that obtained using low-level programming approaches [35, 6, 19, 29], attesting to the framework's sufficiency. To understand this performance impact, we discuss for each application, the computation structure and its use of the load-balancing framework:

4.1 IC-Cedar

IC-Cedar is a parallel protein molecular dynamics program that models the motion of individual protein and surrounding solvent atoms using Newton's equations of motion. Computation proceeds in discrete time steps with four phases: neighbor list calculation, force calculation, motion integration, and coordinate correction. The neighbor list calculation computes, for each set of protein and water atoms, a collection of interacting neighboring atoms that are within a spatial cutoff distance. The force calculation phase computes both short-range bonded forces and long-range Coulombic and Lennard-Jones forces incident on an atom arising from other atoms in its neighbor list. Motion integration uses the computed forces to rescale velocities and update atom positions, and the coordinate correction phase ensures that the new atom positions still preserve bond length constraints. Parallelism is exploited in each phase with data-parallel operations over all the atoms (which are allocated as a collection). For example, in the force calculation phase which dominates the sequential executon time, a dynamic thread is created to compute forces for each atom pair. This thread accesses the position fields of neighboring atoms and updates their force fields.

Application structure. The IC-Cedar application is characterized by dynamic threads created at the start of each iteration. Although the thread creation structure is fixed and closely related to the data objects (a

[2] Two of the applications, Gröbner and Phylogeny, can admit super-linear speedups because the amount of work in the application is affected by non-deterministic concurrent execution. We factor out these anamolous speedups by using application-specific techniques to enforce the same deterministic thread execution order in each configuration.

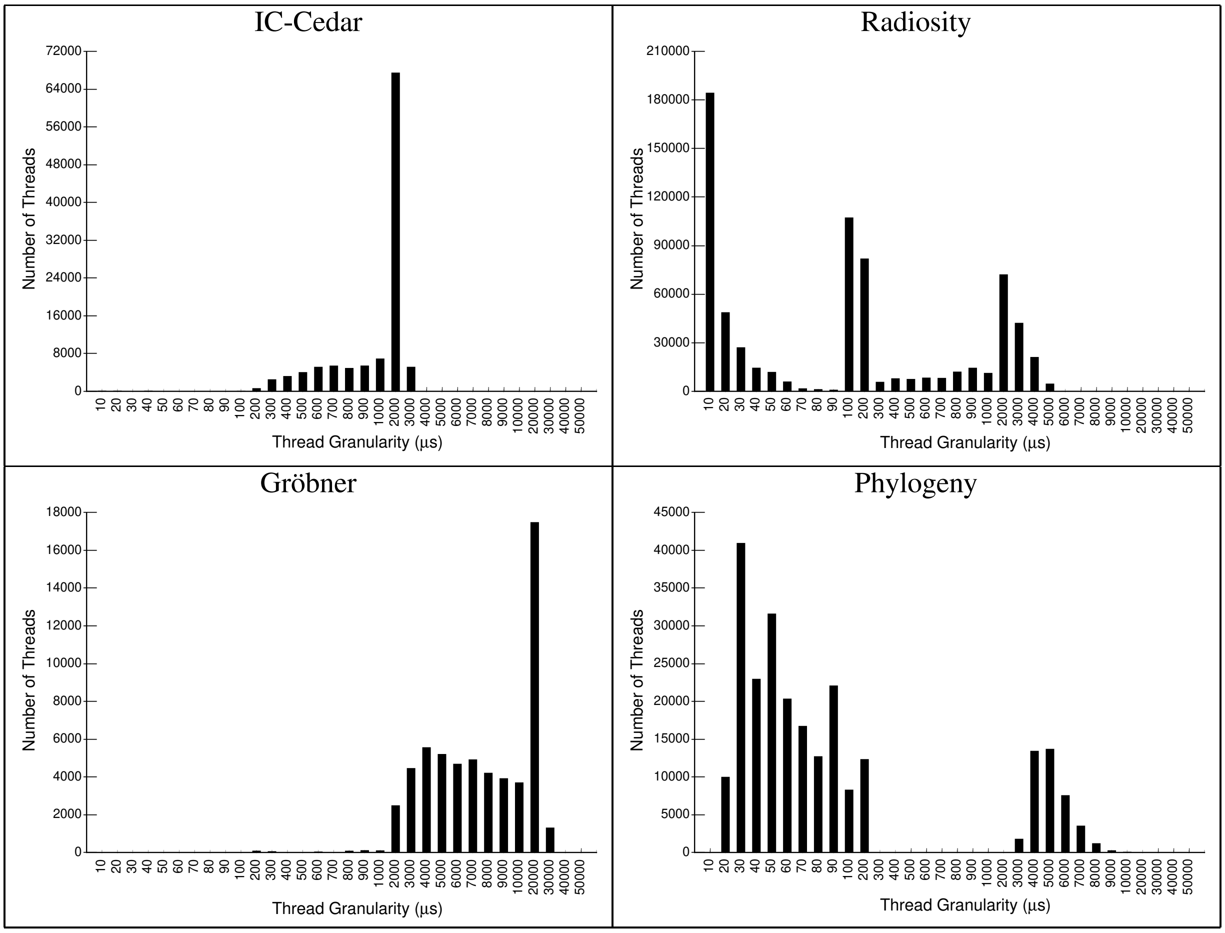

Figure 4: Distribution of task granularities on the Cray T3D for the four applications.

thread is created for each atom), these threads exhibit wide granularity variations depending on the type of the atom (protein or water), length of each atom's neighbor list, and a number of geometric constraints (such as bond angle, bond length, etc.). In addition, the data access pattern of these threads is irregular, determined by the spherical spatial locality region surrounding the atom.

Use of the framework. Although the default policy which uniformly distributes the atoms across the machine achieves good load-balance, it does not preserve spatial locality for the force-calculation threads, resulting in high-overhead irregular data access patterns. Our framework enables a custom distribution for the collection of atom objects which ensures load balance while preserving locality for these threads. We specify a distribution of the collection of atom objects which collocates objects that are spatially close (GroupDistribution). The specific distribution we use is based on a Peano-Hilbert encoding [27, 28] traversal of spatial cells, a technique borrowed from astrophysics applications [14, 32]. This object-placement policy achieves load balance while reducing data access costs.

Performance impact of the framework. The custom distribution for atom objects achieves significant performance improvements on both target platforms: 64-node speedups on the T3D improve from 15.4x to 25.2x, and 16-node speedups on the Origin improve from 2.6x to 5.4x. Using a locality-aware load-balancing policy that relies on a spatial decomposition of the atom object collection primarily reduces coherence actions required to cache the state of the remote atom objects, since most of the accessed objects

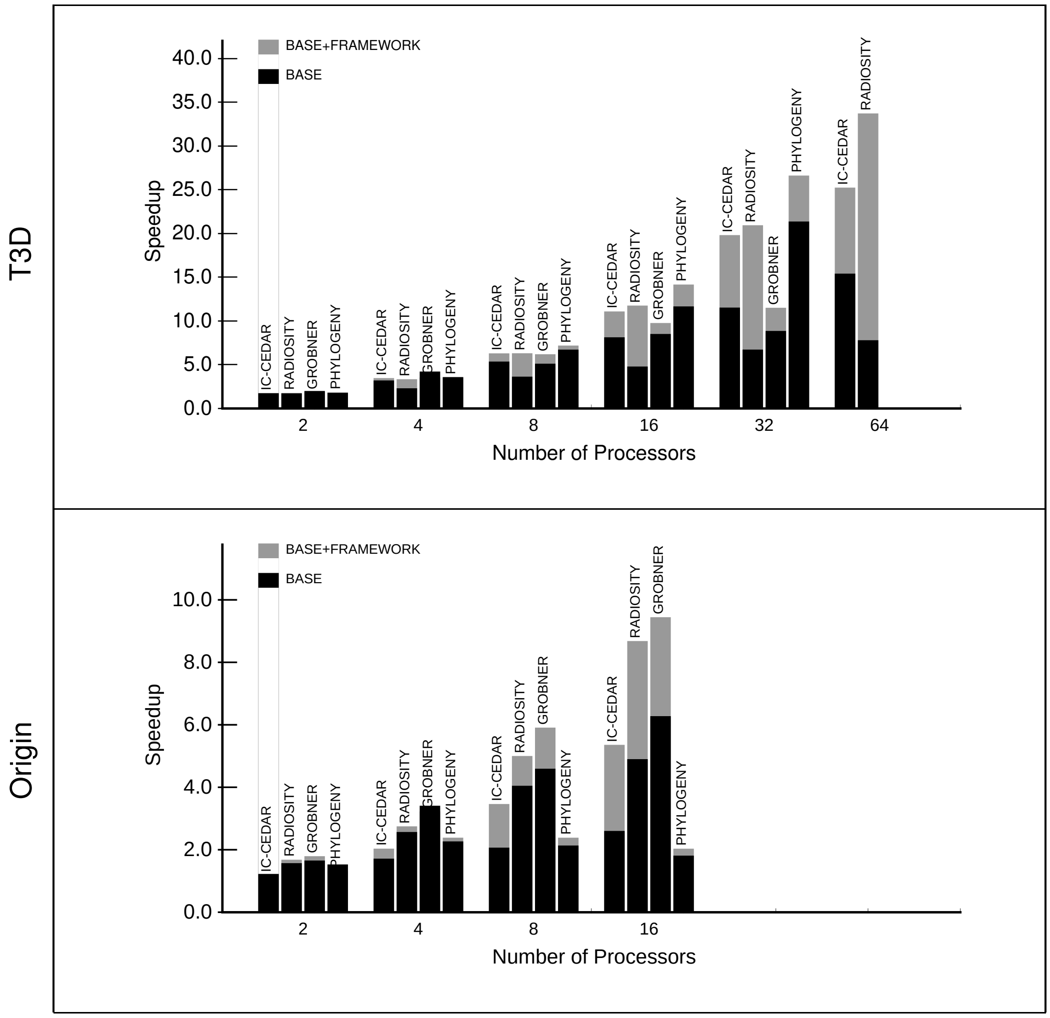

Figure 5: Speedups for four irregular applications on the Cray T3D and the SGI Origin 2000 showing the performance impact of the load-balancing framework. The performance gain obtained from using the framework is shown by the shaded portion of the bar.

are collocated.

4.2 Radiosity

The radiosity method computes global illumination in a scene made up of diffusely reflecting surfaces. Computation proceeds in iterations, where each elemental surface patch's radiosity is computed as a linear combination of the radiosities of all patches on its interaction list (initially everyone), subdividing it or the other patches hierarchically as required to ensure accuracy (based upon an error computation). Radiosity computation requires that the visibility and formfactor of each interaction be previously calculated. Formfactor calculation depends entirely on the patch geometries, whereas the visibility calculation requires ray tracing. The latter dominates the computation time, and is efficiently implemented using a BSP tree. Subdivided patches acquire their own interaction lists and contribute a weighted term to their parent's radiosity. The iterations terminate when the change in the total radiosity falls below a threshold. The application has three levels of parallelism: across all input patches (radiosity calculation), across child patches of a subdivided patch (radiosity calculation), and across neighbor patches stored in the interaction list (visibility, error, and radiosity calculation).

Application structure. The radiosity application is characterized by dynamic threads created throughout the duration of execution. Although the thread creation structure is closely related to the data objects (a radiosity thread is created for each patch, and visibility and error threads are created for each interaction), the object structure evolves in a data-dependent fashion due to patch refinement. This results in an irregular and unpredictable thread structure. Additionally, threads exhibit wide granularity variations and access data objects in an unstructured fashion.

Use of the framework. The irregular nature of the application results in poor load balance using the default object-placement policy that uniformly distributes top-level patch objects, and allocates the refined patches locally. Our framework improves overall load balance by permitting alternate policies for two of the thread subsets which account for most of the imbalance—radiosity and visibility calculation. Exploiting the fact that the creation structure of radiosity threads is closely related to the structure of the patch objects, we control their distribution using an object placement policy (ObjectLocation). The patch objects are randomly allocated (instead of locally), producing a balanced distribution of the radiosity threads. On the other hand, visibility threads do not show a similar correspondence to the structure of the interaction and patch objects, so an explicit thread placement policy is required. We use a locality-sensitive work-sharing policy (WorkSharingLocal): threads are distributed upon creation to a processing node determined by applying a hash function on the pair of patch objects used in the visibility calculation. The hash function ensures that each node receives a set of threads which reuse the same set of patch objects.

Performance impact of the framework. The two policies are sufficient to significantly improve performance on both target platforms: 64-node speedups on the Cray T3D improve from 7.8x to 33.7x, and 16-node speedups on the SGI Origin 2000 improve from 4.9x to 8.7x. These improvements mainly result from a reduction in idle time, a consequence of improved load balance. The improvements are more pronounced on the T3D because although the communication costs on both platforms are similar, the Origin is an order of magnitude faster for computation as compared to the T3D. Consequently, as compared to the Origin, idle time eliminated by the custom policies accounts for a larger fraction of the overall execution time.

4.3 Gröbner

The Gröbner application computes the Gröbner basis [5] of a set of polynomials. The algorithm starts off with an initial basis set of polynomials equal to the input set, and evaluates each pair of polynomials in a rank order determined by a heuristic metric. Evaluation of a polynomial pair (consisting of arbitrary-precision arithmetic operations to isolate irreducible polynomial terms) can potentially lead to the addition of a new polynomial into the basis. This, in turn, generates additional polynomial pairs for evaluation which correspond to the new polynomial and existing basis polynomials. The algorithm completes when there are no polynomial pairs left for evaluation. The parallelism in the application arises from parallel evaluation of polynomial pairs. Each pair creates a logical thread which accesses both the basis and polynomial objects. Threads which produce polynomials with irreducible terms invoke a method on the basis object to augment it and create new threads.

Application structure. The Gröbner application is characterized by threads created throughout the duration of execution. Although the thread structure is closely related to the data objects (threads are created for polynomial pairs), threads exhibit wide variations in granularity which arise from the data-dependent and execution order-dependent nature of pair evaluations. More importantly, threads require prioritized execution to ensure that "good" polynomials enter the basis early on: these ensure that most of the subsequent pair evaluations are completely irreducible.

Use of the framework. The default policy of randomly distributing pair evaluation threads achieves good load-balance but does not respect thread priorities, leading to redundant work. Our framework enables better load balance by permitting the distribution of pair evaluation threads to be controlled by a priority-sensitive policy. We use a global work-stealing priority scheduler (WorkStealingGlobalPriority), which enqueues the pair-evaluation threads according to priority order on a distinguished processor. When processors become idle, they steal work from this distinguished processor.

Performance impact of the framework. Specifying a custom thread-ordering policy for the pair evaluation threads considerably improves performance on both target platforms: 32-node speedups on the T3D improve from 8.8x to 11.5x, and 16-node speedups on the Origin improve from 6.3x to 9.5x. The improvements result from the ability of the thread-ordering policy to both effectively preserve thread execution priorities, and provide good load balance using a centralized queue. Our results show that for the Gröbner application, the priority preserving and load balancing advantages of the WorkStealingGlobalPriority policy outweigh contention costs for the shared queue.

4.4 Phylogeny

The Phylogeny application computes the evolutionary history of a set of species represented as a vector of character values, producing a phylogenetic tree which makes explicit their evolutionary paths. Our algorithm is based on the character compatibility method: a character is compatible if no value for that character arises more than once in any path of the tree. The algorithm searches for the largest compatible subset of characters, using as a subroutine the solution for the *perfect phylogeny problem*, which, given a set of species and a set of characters, determines whether there exists a phylogenetic tree compatible with *all* the characters.

The algorithm is essentially a bottom-up search of the character lattice (depth-first and right-to-left beginning with small subsets and progressing to larger subsets), asking whether or not each subset is compatible. The algorithm maintains a failure set of subsets to facilitate pruning based upon the property that the subsets of a compatible set are also compatible. Each subset is first checked against the failure set; no subset that has a subset that failed requires the perfect phylogeny procedure, otherwise the perfect phylogeny procedure is called. Based on the result of the procedure, either the subset is added to the failure set, or additional subsets are generated for exploration. Our parallel algorithm evaluates multiple character subsets as independent threads.

Application structure. The Phylogeny application is characterized by dynamic creation of threads spread throughout the duration of execution. Threads exhibit imbalance in creation (only compatible subsets create additional threads) and wide data-dependent variations in granularity, both to check against the failure set and to determine a solution to the perfect phylogeny problem. Additionally, thread execution needs to be prioritized to correspond to a depth-first and right-to-left traversal of the search space to reduce redundant work.

Use of the framework. The default policy randomly distributes subset evaluation threads on the different processors. However, this achieves poor performance because the policy does not preserve locality among sibling subset evaluation threads, nor does it respect their priority constraints. Our framework enables better load balance by permitting the distribution of subset evaluation threads to be controlled by a low-overhead locality- and priority-sensitive policy. Motivated by the large number of small granularity threads, we use a prioritized work-stealing policy (WorkStealingPriority), which enqueues the subset evaluation threads in per-processor queues according to priority order. When a processor becomes idle, it steals work from neighboring processors.

Performance impact of the framework. Associating a priority-sensitive load-balancing policy with the subset evaluation threads improves performance on the T3D by 20% (from 21.3x to 26.6x on 32 nodes). The improvement is primarily due to a reduction in redundant work arising from the prioritized policy, and secondarily from reduced coherence overheads because of locality between sibling threads. Unlike the default policy, the custom policy enqueues all threads created by a successful subset evaluation into the local queue. Note that, unlike Gröbner, the smaller thread granularities preclude use of a global work-stealing scheduler because of the higher scheduling costs.

On the Origin however, the performance improvements are not as impressive (16-node speedups improve from 1.8x to 2.1x). The overall low performance is an attribute of the application itself, arising from fine-grained reader-writer sharing of the failure set structure (which is implemented as a trie). These operations result in high overheads for coherence and synchronization, even with hardware support for cache coherence.

4.5 Summary and Discussion

Table 2 summarizes the custom policies used for each of the applications. Our performance results show that the framework is effective at achieving load-balance for dynamic multithreaded computations that exhibit wide-variations in thread granularity, and sensitivity to data locality and thread execution order. These latter characteristics are a consequence of a high-level, natural expression of irregular applications using a concurrent object-oriented language. The framework factors out the varied thread behaviors by enabling use of a different load-balancing policy for different thread subsets, each of which exhibit simpler behavior. By doing so, the framework enables an irregular application to be naturally expressed, relieving the programmer from having to structure the program on his/her own so that it can be load-balanced using a single uniform policy.

APPLICATION	THREAD STRUCTURE	POLICY
IC-Cedar	access patterns are sensitive to spatial locality	GroupDistribution object-placement policy which specifies a spatial distribution
Radiosity	fine-granularity threads whose creation structure is closely related to a dynamically evolving object structure	ObjectLocation object-placement policy which specifies random allocation
	thread creation structure is not related to any object structure, and access patterns are sensitive to data locality	WorkSharingLocal thread-placement policy that is locality-aware
Gröbner	threads are highly sensitive to execution priority	WorkStealingGlobalPriority thread-ordering policy that uses a centralized priority queue
Phylogeny	fine-granularity threads which are both locality- and priority-sensitive	WorkStealingPriority thread-ordering policy that incurs lower overheads, preserves locality, and uses per-processor priority queues

Table 2: Summary of custom load-balancing policies enabled by the framework.

However, these benefits in program complexity come at the cost of having to understand application thread behavior so as to identify appropriate thread subsets and associate suitable load-balancing policies for each subset. We believe that knowledge of application semantics is required for efficient execution of irregular applications. While this may appear to result in unreasonable complexity, three observations can be made from our application studies which show that the complexity is, in fact, manageable. First, the language support of the framework allows specification of the load-balancing policy to be completely orthogonal to the functional specification. This allows programmers to both add load-balancing decisions after program correctness has been verified, as well as conveniently experiment with multiple alternative policies. Second, the choice of the policy is only dependent on high-level application characteristics such as the granularity variations of threads created at a particular site, the locality patterns of data-object access, and the importance of thread execution priorities. These characteristics are all relatively independent of underlying target platform properties, as attested by the fact that for each of our applications, the same policies yield performance improvements on both the T3D and the Origin. Third, despite minor differences, the policies employed by each of the application programs are generic enough to be supplied by the run-time system as part of a library of load-balancing policies. In related work reported elsewhere [22], we show how once a library of policies is identified, a synthetic task-generation microkernel can be used to provide qualitative guidance to the programmer about which policy is suitable for a particular region of the application space. For instance, Figure 6 shows the suitability of policies in Table 2 for different regions of the application space defined by four axes—thread granularity, thread imbalance (which represents the imbalance in thread creation structure), priority sensitivity, and locality sensitivity. To consider one example, a policy such as WorkSharingLocal is suitable when threads in the subset exhibit medium to large granularity, large imbalance in thread creation structure, and high sensitivity to data locality, but with a reduced need for preserving thread priorities.

5 Related Work

Our framework provides language- and run-time support for decomposing the overall load-balancing problem for a complex application into several simpler problems, each of which focus on ensuring load-balance for a subset of the threads. This approach represents a major departure from conventional approaches to load-balancing [18, 12, 17, 23, 21, 7, 26, 4, 24] which employ a a single unified solution for application wide load balancing. While the latter approaches may be appropriate for regular or medium-grained computations, our studies show that a single uniform policy is insufficient to achieve efficient load balancing for fine-grained dynamic computations. However, these approaches can be viewed as addressing an orthogonal problem of load balancing a particular subset of threads. Our framework builds on such component policies to synthesize overall application load balance.

Most similar to our approach are systems that allow different subsets of threads to be subject to different load balancing and scheduling policies during execution. Custom scheduling is a feature of several real-time and multimedia operating systems [13, 31]. The primary difference of our approach with respect to real-time and multimedia operating systems is that the integration between different schedulers is considerably more efficient, relying on non-preemptive scheduling of cooperating application threads.

On the other hand, only a handful of programming systems, such as Presto [3] and Converse [21], possess similar capabilities. While support for attaching custom schedulers in both Presto and Converse share similar objectives to our approach, there are important differences. First, the thread scheduler interface in both systems is more suitable for medium-grained threads, requiring multiple indirect function calls per scheduling operation. Second, the interfaces do not specify how processing resources are allocated to multiple custom schedulers, nor how quiescence is detected for schedulers across processing nodes. Finally, although both Presto and Converse provide mechanisms for customization, few large-scale experiments have discussed the resulting benefits.

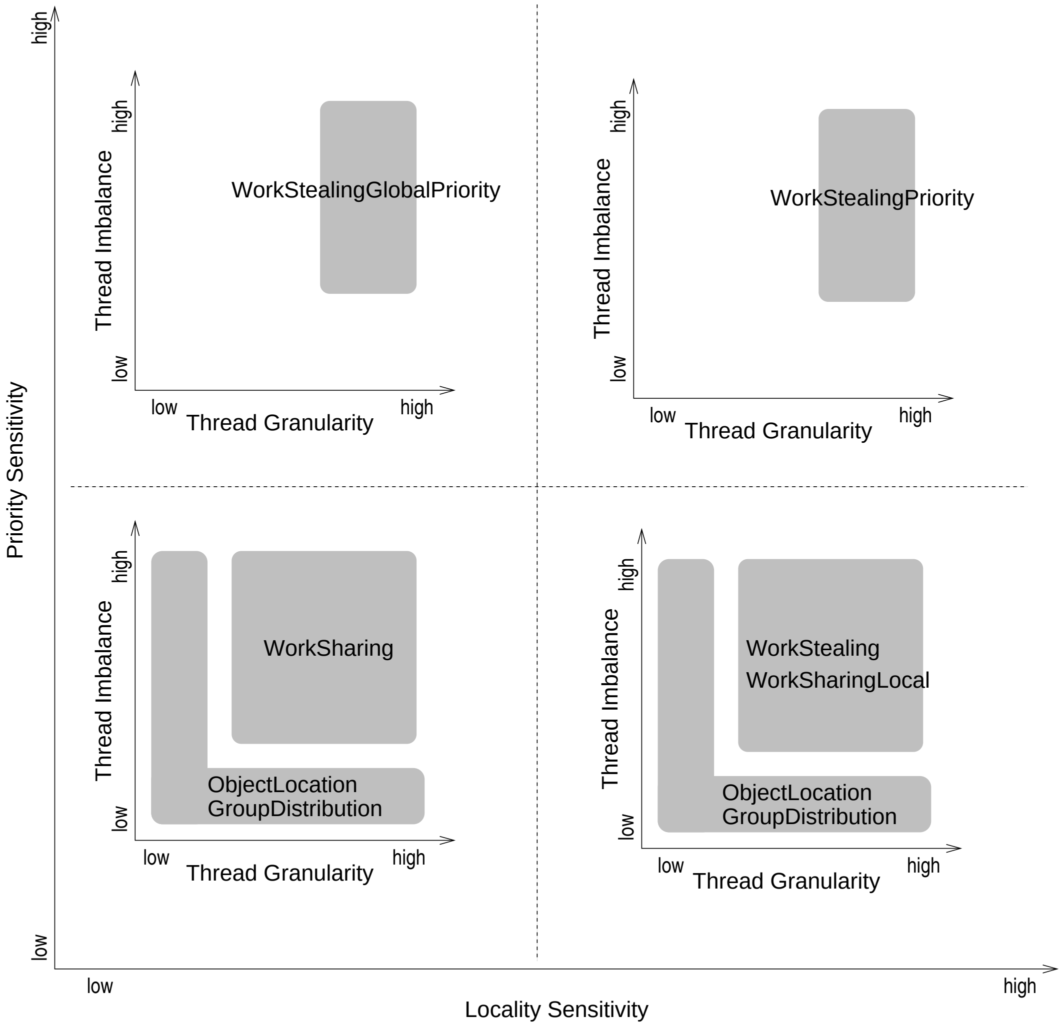

Figure 6: Suitability of different load-balancing policies in different regions of the application space.

6 Conclusions

We have described a general load-balancing framework for dynamic multithreaded applications characterized by wide variations in thread granularity, and sensitivity to data locality and thread execution priority. Unlike traditional approaches which employ a single processor-centric policy for load-balancing all threads, our approach provides language and run-time support to allow the application to be viewed as being made up of several thread subsets, each of which exhibit simpler behaviors and can be load-balanced independently. Results for four large irregular applications on the Cray T3D and the SGI Origin 2000 demonstrate advantages of the framework: performance improves by up to an order of magnitude as compared to using a uniform load-balancing policy.

References

[1] J. Barnes and P. Hut. A hierarchical O(N log N) force calculation algorithm. Technical report, The Institute for Advanced Study, Princeton, New Jersey, 1986.

[2] Marsha J. Berger and Joseph Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, 1984.

[3] B.N. Bershad, E.D. Lazowska, and H.M. Levy. Presto: A system for object-oriented parallel programming. *Software—Practice and Experience*, 18(8):713–732, August 1988.

[4] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, Andrew Shaw, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of Principles and Practice of Parallel Programming*, pages 207–216, 1995.

[5] B. Buchberger. *Multidimensional Systems Theory*, chapter Grobner basis: an algorithmic method in polynomial ideal theory, pages 184–232. D. Reidel Publishing Company, Dordrecht, 1985.

[6] S. Chakrabarti and K. Yelick. Implementing an irregular application on a distributed memory multiprocessor. In *Proceedings of the Fourth ACM/SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 169–179, May 1993.

[7] Rohit Chandra, Anoop Gupta, and John L. Hennessy. Data locality and load balancing in COOL. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 249–259, 1993.

[8] Andrew Chien, Julian Dolby, Bishwaroop Ganguly, Vijay Karamcheti, and Xingbin Zhang. Supporting high level programming with high performance: The Illinois Concert system. In *Proceedings of the Second International Workshop on High-level Parallel Programming Models and Supportive Environments*, pages 15–24, April 1997.

[9] Andrew Chien, Julian Dolby, Bishwaroop Ganguly, Vijay Karamcheti, and Xingbin Zhang. Evaluating high level parallel programming support for irregular applications in ICC++. *Software–Practice and Engineering*, 1998.

[10] Andrew Chien and Uday Reddy. ICC++ language definition. Concurrent Systems Architecture Group Memo, Also available from `http://www-csag.cs.uiuc.edu`, February 1995.

[11] Andrew A. Chien, Uday S. Reddy, John Plevyak, and Julian Dolby. ICC++ – a C++ dialect for high-performance parallel computation. In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software*, pages 76–95, March 1996.

[12] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *J. Parallel and Distributed Computing*, 37(1):70–82, 1996.

[13] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 107–121, 1996.

[14] Ananth Y. Grama, Vipin Kumar, and Ahmed Sameh. Scalable parallel formulations of the Barnes-Hut method for N-body simulations. In *Proceedings of Supercomputing Conference*, pages 439–448, November 1994.

[15] L. Greengard and V Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73:325–48, 1987.

[16] A. Grimshaw. Easy-to-use object-oriented parallel processing with Mentat. *IEEE Computer*, 5(26):39–51, May 1993.

[17] Matthew Haines, David Cronk, and Piyush Mehrotra. On the design of Chant: A talking threads package. In *Proceedings of Supercomputing'94*, pages 350–359, November 1994.

[18] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling FORTRAN D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[19] Jeff A. Jones. Parallelizing the phylogeny problem. Master's thesis, University of California, Berkeley, CA, December 1994.

[20] L. V. Kale and Sanjeev Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of OOPSLA'93*, pages 91–108, 1993.

[21] Laxmikant V. Kalé, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua M. Yelon. Converse: an interoperable framework for parallel programming. In *Proceedings of the International Parallel Processing Symposium*, pages 212–217, 1996. Available from `http://charm.cs.uiuc.edu/papers/converse-ipps96.ps`.

[22] Vijay Karamcheti. *Run-Time Techniques for Dynamic Multithreaded Computations*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1998.

[23] R. Keller and F. C. H. Lin. Simulated performance of a reduction based multiprocessor. *IEEE Computer*, 17(7), July 1984.

[24] E. Mohr, D. Kranz, and R. Halstead Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[25] Scott Pakin, Vijay Karamcheti, and Andrew A. Chien. Fast Messages: Efficient, portable communication for workstation clusters and mpps. *IEEE Concurrency*, 5(2):60–73, April-June 1997. Available from `http://www-csag.cs.uiuc.edu/papers/fm-pdt.ps`.

[26] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995.

[27] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[28] Jaswinder Pal Singh. *Parallel Hierarchical N-Body Methods and Their Implications For Multiprocessors*. PhD thesis, Stanford University, Stanford, CA, February 1993.

[29] Jaswinder Pal Singh, Anoop Gupta, and Marc Levoy. Parallel visualization algorithms: Performance and architectural implications. *IEEE Computer*, 27(7):45–56, July 1994.

[30] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: a mechanism for integrated communication and computation. In *Proceedings of the International Symposium on Computer Architecture*, pages 256–266, 1992.

[31] C. Waldspurger and W. Weihl. Lottery scheduling: Flexible, proportional share resource management. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 1994.

[32] M. Warren and J. Salmon. A parallel hashed oct-tree N-body algorithm. In *Proceedings of Supercomputing Conference*, pages 12–21, 1993.

[33] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the International Symposium on Computer Architecture*, pages 24–36, 1995.

[34] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System*. MIT Press, Cambridge, MA, 1990. ISBN 0-262-24029-7.

[35] Xingbin Zhang, Vijay Karamcheti, Tony Ng, and Andrew Chien. Optimizing COOP languages: Study of a protein dynamics program. In *IPPS'96*, pages 235–240, 1996.