Data Locality and Load Balancing in COOL

Rohit Chandra, Anoop Gupta, John L. Hennessy
Center for Integrated Systems
Stanford University, Stanford CA 94305
Email: {chandra, gupta, hennessy}@cs.stanford.edu

Abstract

Large-scale shared memory multiprocessors typically support a multilevel memory hierarchy consisting of per-processor caches, a local portion of shared memory, and remote shared memory. On such machines, the performance of parallel programs is often limited by the high latency of remote memory references. In this paper we explore how knowledge of the underlying memory hierarchy can be used to schedule computation and distribute data structures, and thereby improve data locality. Our study is done in the context of COOL, a concurrent object-oriented language developed at Stanford. We develop abstractions for the programmer to supply optional information about the data reference patterns of the program. This information is used by the runtime system to distribute tasks and objects so that the tasks execute close (in the memory hierarchy) to the objects they reference.

We demonstrate the effectiveness of these techniques by applying them to several applications chosen from the SPLASH parallel benchmark suite. Our experience suggests that improving data locality can be simple through a combination of programmer abstractions and smart runtime scheduling.

1 Introduction

With rapid improvements in processor speeds, the memory system is increasingly the limiting factor in the performance of computer systems. The latency of memory accesses can be especially high in large-scale shared memory multiprocessors that have deep memory hierarchies. For example, in the Stanford DASH [10], while the local cache access takes only a single clock cycle, a miss serviced by the local portion of shared memory takes about thirty clock cycles, and a remote miss takes over a hundred clock cycles. Improving data locality in parallel programs can reduce the time spent by a processor waiting for data, and is critical for achieving good performance.

Data locality in parallel programs can be improved by scheduling computation and distributing data structures with an awareness of the underlying memory hierarchy, so that tasks execute close to the objects they reference. Determining an appropriate distribution of tasks and objects, however, requires knowledge about the program that is often beyond the scope of the compiler, but that may be readily known to the programmer. In this paper we focus on the support provided for such programmer intervention in COOL [4], a parallel programming language designed to express task-level parallelism. In addition to support for expressing parallelism, COOL provides abstractions for the programmer to supply hints about the data objects referenced by parallel tasks. These hints are used by the runtime system to appropriately schedule tasks and migrate data, and thereby exploit locality in the memory hierarchy; they do not affect the semantics of the program. This approach provides a clear separation of functionality: the programmer can focus on exploiting parallelism and supplying hints about the object reference patterns, leaving the details of task creation and management to the implementation.

Since our approach depends on programmer participation, it is important that the abstractions be intuitive and easy to use. At the same time the abstractions should be powerful enough to exploit locality at each level in the memory hierarchy, without compromising performance. We address these goals through the following key elements of our design. First, the abstractions are integrated with the task and object structure of the underlying COOL program so that the hints are easily supplied. Second, the abstractions are structured as a hierarchy of optional hints so that simple optimizations are easily obtained as defaults while more complex ones require incremental amounts of programmer effort. Finally, the abstractions are designed to enable experimentation with different optimizations, and address locality at each level in the memory hierarchy.

We demonstrate that our approach is both effective and easy to use, by applying it to several applications chosen from the SPLASH [15] parallel benchmark suite. We show how information about the program can be easily supplied by the programmer, usually with just a few lines of optional code, and how it is used by the runtime system for scheduling optimizations that improve both cache and memory locality. Performance improvements with these hints range from 60-135%, and are especially high in programs where locality of data references is critical. The primary conclusion of this work, therefore, is that a combination of simple programmer hints and smart runtime strategies is a highly effective approach for improving data locality and load balance.

The rest of this paper is organized as follows. Section 2 presents a brief overview of COOL. Section 3 reviews the basic techniques for improving data locality. Section 4 describes our approach, the abstractions that we provide, and the runtime scheduling mechanisms. Section 5 describes the implementation of the runtime scheduler. Section 6 applies these techniques to various programs, and presents performance results. We present related work in Section 7, and conclude in Section 8.
2 COOL Language Overview

COOL is a parallel programming language designed for applications using task-level parallelism on shared memory multiprocessors. It is an extension of C++, chosen because of its support for the definition of abstract data types and its widespread use. Our approach emphasizes the integration of concurrency and synchronization with data abstraction. Programs structured around objects have the advantage that they are modular. In addition, the object structure of COOL programs makes it easier to address load balancing and data locality issues. First, the objects referenced by tasks are more easily identified. Second, programs in COOL dynamically create lightweight tasks, providing the opportunity to schedule a task (when it is created) close to the objects it references with very low overhead.

In COOL, parallelism is expressed by identifying a function to be parallel; such functions execute asynchronously when invoked. Parallel functions communicate through program data in the shared address space and synchronize through monitors and condition variables [7]. Operations on shared objects may be declared mutex, signifying that they must acquire exclusive access to the object before executing. Event synchronization is expressed through operations on condition variables that allow a thread to wait for an event, or to signal a waiting thread when the appropriate event occurs. Finally, for convenience, we also provide a construct called waitfor to express fork-join style synchronization at the task level. The keyword waitfor can be attached to a scope in the program; upon reaching the end of that scope, the executing thread blocks until all tasks created within the scope of the waitfor have completed. We will see examples of COOL programs later in the paper, and a detailed description of COOL can be found in [4].

3 Exploiting Memory Hierarchy

![Memory Hierarchy Diagram]

Figure 1: Multiprocessor memory hierarchy.

In this paper we assume a three-level memory hierarchy (see Figure 1) consisting of (a) per processor caches, (b) local portion of shared memory, and (c) remote shared memory. We believe that this is a reasonable abstraction for many large scale multiprocessors (e.g. the Stanford DASH, MIT Alewife [1]), since it captures the essence of the memory hierarchy for such machines. (In these multiprocessors the ratio of the latencies of local to remote references is usually much more significant than variations in the latencies to different remote processing elements).

For such architectures, the primary mechanisms to improve data locality in a parallel application are task scheduling and object distribution. The latency of references to an object can be reduced by scheduling computation and distributing objects so that a task executes on a processor that is close (in the memory hierarchy) to the objects referenced by the task. However, scheduling tasks to improve cache and memory locality often conflicts with the simultaneous goal of scheduling to achieve good load balance. Therefore, it is important to weigh the tradeoff between locality and load balance when determining a task schedule.

Given a specific task decomposition for an application, we can exploit cache locality (i.e. reuse in the cache) by scheduling tasks that reference the same objects on the same processor. Cache locality can be further enhanced by scheduling these tasks back to back to reduce possible cache interference caused by the intervening execution of other unrelated tasks. When cache locality alone is insufficient, we can exploit memory locality by identifying the primary object(s) referenced by a task and executing the task on the processor that contains the object(s) in its local memory. Thus, references to the object that miss in the cache will be serviced in local rather than remote memory, resulting in lower latency. However, ensuring a good load balance while exploiting memory locality requires both task scheduling and an appropriate distribution of objects across the local memories of processors. Finally, the impact of long-latency memory references can be further reduced by dynamically migrating the object, or by prefetching the object closer to the processor.

4 Language Abstractions and Runtime Support

Determining an efficient task and object distribution requires knowledge about the program that is often beyond the scope of a compiler, but may be known to the programmer. In COOL, therefore, we provide abstractions for the programmer to supply hints about the data reference pattern of the program. The COOL implementation has runtime mechanisms that use these hints to exploit locality at different levels in the underlying memory hierarchy. The abstractions are designed to be easy to use, and enable the programmer to supply hints naturally in terms of the objects referenced by a task. In addition, the hints are optional since they only affect the performance of the program.

We provide a hierarchy of control, ranging from smart defaults to simple hints to very specific hints. Smart default scheduling policies provide the basic optimizations without any programmer effort. When programmer participation becomes necessary, we have abstractions for providing simple hints about object usage. These hints are sufficient for most programs. For programs that require more complex optimizations, we have abstractions that provide greater control over task scheduling and data placement. These latter abstractions require a greater degree of understanding of the application and the memory hierarchy, but are within the same framework (as the simple abstractions) and still relatively easy to use.

For simplicity, we currently leave the burden of distributing and migrating objects across processors’ memories to the programmer, and simply provide constructs to allow this to be expressed in the language. Ongoing compiler [8, 16] and operating systems [2, 9] research has enjoyed some success in automatically distributing objects, and could reduce this burden on the programmer.

4.1 The Abstractions

In COOL, information about the program is provided by identifying the objects that are important for locality for a task. Along with a parallel function, the programmer can specify a block of code that contains affinity hints (see Figure 2). This block of code is executed when the parallel function is invoked and a corresponding task is created. The affinity hints themselves are simply evaluated by the runtime system to determine their effect on the scheduling of the task; they do not affect the semantics of the program. We present the hierarchy of the affinity abstractions in this section.

Defaults: Parallel functions in COOL have a natural association with the object that they are invoked on. So by default, tasks created by the invocations of a parallel function are scheduled on
the processor that contains the corresponding object in its local memory. The task is therefore likely to reference the object in local rather than remote memory. In addition, if there are several tasks that operate on that object, then only the first task will need to fetch the object from memory; the rest are likely to find the object in the cache. The runtime system executes such tasks back to back on that processor to reduce the cache interference caused by the execution of unrelated intervening tasks. This further improves cache locality.

Simple Affinity: When a parallel function would benefit from locality on an object other than the base object, the programmer can override the default by explicitly identifying that object through an affinity specification for the function. When affinity for an object is explicitly identified, the runtime system schedules the task in a manner similar to the default described above, except that the scheduling is based on the specified object rather than the default object. As a result, the object is likely to be referenced in the cache, and references to the object that miss in the cache are serviced in local rather than remote memory. In addition, if there are several tasks that operate on that object, then only the first task will need to fetch the object from memory; the rest are likely to find the object in the cache. The runtime system executes such tasks back to back on that processor to reduce the cache interference caused by the execution of unrelated intervening tasks. This further improves cache locality.

Task and Object Affinity: We have shown how to exploit both cache and memory locality by simply specifying the objects referenced by the parallel function. It is often useful to simultaneously exploit memory locality on one object and cache locality on a different object.

We illustrate this using Gaussian elimination on a matrix. In our algorithm, each column of the matrix is updated by the columns to its left to zero out the entries above the diagonal element. Once the column has received all such updates (i.e. all entries above the diagonal element are zero), it is used to update other columns to its right in the matrix. A task in this algorithm is an invocation of the parallel function update (see Figure 3), which updates a destination column by a given source column. A desirable execution schedule and object distribution for this column-oriented algorithm are as follows (see Rothberg [13]). The algorithm exploits memory locality on the destination column by the task where the destination column is allocated (the number of columns per processor is so large that they are not expected to fit in the cache). Distributing the columns across processors in a round-robin fashion results in good load distribution. In addition, the algorithm exploits cache locality on the source column since each processor executes multiple updates that involve the same source column. However, it is important that each processor execute the tasks involving the same source column back to back; this avoids the execution of other tasks that might flush the source column from the cache.

The above example clearly shows that we wish to exploit cache locality on the source column and memory locality on the destination column. We therefore allow the keywords OBJECT and TASK to be specified with an affinity statement. The TASK affinity statement identifies tasks that reference a common object as a task-affinity set; these tasks are executed back to back to increase cache reuse. The OBJECT affinity statement identifies the object for memory locality; the task is collocated with the object. As shown in Figure 3, we can simultaneously exploit cache locality through task affinity on the source column, as well as memory locality through object affinity on the destination column. This exactly captures the way the algorithm was hand-coded using ANL macros [3] to run on the Stanford DASH multiprocessor; the same scheduling is very simply expressed in Cool.

Processor Affinity: Finally, for load balancing reasons it often becomes necessary to directly schedule a task on a particular processor (in practice the corresponding server process), rather than indirectly through the objects it references. We therefore provide processor affinity through the PROCESSOR keyword that can be specified for an affinity declaration. An integer argument is supplied instead of an object address, and its value (modulo the number of server processes) is used as the server number on which the task is scheduled.

Object Distribution: In addition to affinity hints for a task, we also allow the programmer to distribute objects across memory modules, both when they are allocated, as well as by dynamically migrating them to another processor’s local memory. Since tasks with object affinity are executed where the object is allocated, distributing objects across memory modules can improve the load balance of the program.
By default, memory is allocated from the local memory of the requesting processor. To allocate memory from within the local memory of a particular processor, a processor number can be supplied as an additional argument to the new operator. For dynamic object distribution we provide a migrate function that takes a pointer to an object and a processor number, and migrates the object to the local memory of the specified processor (modulo the number of server processes). An optional third argument that specifies the number of objects to be migrated is useful to migrate an array of objects.

Finally, the home function returns the number of the processor that contains the given object allocated in its local memory.

The various affinity hints are summarized in Table 1. If affinity is specified for multiple objects then we currently schedule the task based on the first object. There are obvious better heuristics that would determine the relative importance of objects based on their size and schedule the task on the processor that has the most objects in its local memory, while prefetching the remaining objects. We plan to study such tradeoffs in the future.

### 4.2 The Runtime Scheduler

The runtime scheduler uses the affinity hints to schedule tasks as described above. To maintain good load balance an idle processor steals tasks from other processors. We are also experimenting with other optimizations that the runtime system may perform based on the hints about object usage. For instance, tasks in a task-affinity set should preferably be executed on the same processor, but the particular processor can be chosen based on load balancing considerations. In addition, tasks scheduled with task-affinity can be stolen as a set by an idle processor to improve load balance and still benefit from cache locality. In contrast, tasks scheduled with object-affinity should preferably not be stolen; moving such tasks away from the memory containing the objects they reference may result in an increase in effective memory latency arising from remote memory references to the objects.

### 5 Implementation

In this section we give a brief overview of the runtime system and the task queue structure used to support the task scheduling optimizations.

When a COOL program begins execution several server processes are created, usually one per available processor. Ideally a server process runs on the same processor throughout its lifetime. It continually fetches a task from the task queue and executes it (without preemption) until the task completes or blocks awaiting some synchronization event. There are two kinds of task queues per server

---

2The operating system on DASH supports data allocation at the page level only, therefore the migrate call on these machines is implemented through the migration of entire pages spanned by the object, rather than the object alone.

---

3An operating system call is available on DASH which, given a virtual address, returns the number of the processor that contains the physical page corresponding to that address. To avoid the cost of the system call, the runtime can maintain the location of an object in a special variable within the object.
thereby identifying tasks in that task affinity set. This allows us to simultaneously exploit both cache and memory locality on different objects, as illustrated by the example in Section 4.1. Tasks in the same task-affinity set therefore get mapped onto the same queue, and can be serviced back to back. Collisions of different task-affinity sets on the same queue can be minimized by choosing a suitably large array size.

This task queue structure is implemented efficiently. Determining where to schedule a task simply requires two modulo operations. Within the task queue structure of each processor, the non-empty queues in the array are linked together to form a doubly-linked list and provide fast enqueue and dequeue operations.

6 Case Studies

In this section we apply our approach to several scientific and engineering applications chosen from the SPLASH parallel benchmark suite. For each application we describe both the structure of the program and how the concurrency is expressed in terms of the results due to space limitations.

We present results from five SPLASH applications. The first application (Ocean) has a simple structure, and simple affinity between tasks and objects is sufficient to improve both cache and memory performance. The next application (LocusRoute) illustrates the benefits of scheduling related tasks on the same processor to enhance reuse in the cache. The third application (Panel Cholesky) is more complex, and requires careful placement of objects and tasks to improve both cache and memory locality. For the final two applications, Barnes-Hut and Block Cholesky, we often present a summary of the results due to space limitations.

We present performance results running on a prototype of the Stanford DASH multiprocessor, consisting of thirty-two 33 MHz R3000 processors. The processors are organized into (eight) clusters, with each cluster containing four processors and some physical memory (28 MB each). Each processor has a 64 KB first level cache and a 256 KB second-level cache. References that are satisfied in the first-level cache take a single processor cycle, while hits in the second-level cache take about 14 cycles. Memory references to data in the local cluster memory take nearly 30 cycles, while references to the remote memory of another cluster take about 100-150 cycles. We measure the time spent in the parallel portion of the code, and plot its speedup with respect to the time taken by a serial version of the code running on one processor. We also use the hardware performance monitor on DASH [11], that enables us to monitor the bus and network activity in a non-intrusive manner.

```c
class region { 
  // data declarations ...
  public:
    parallel void laplace (region_c*); 
    ... 
  
  class grid { 
    public:
      // Grid composed of regions.
      region_c* region;
      void laplace (grid_c* p) { 
        int i;
        waitfor { 
          for (i=0; i<N; i++)
            // Process all the regions in parallel.
            region[i].laplace (&(p–>region[i]));
        } // 'waitfor' the operations to complete for all regions.
      }

      // Distribute the regions across processors.
      void distribute () {
        int i;
        for (i=0; i<N; i++)
          migrate (region+i, i);
      } 
    A, B;

    main () {
      A.distribute ();
      B.distribute ();
      ... 
      A.laplace (B);
    }
  }

  void laplace (grid_c* p) { 
    int i;
    waitfor { 
      for (i=0; i<N; i++)
        // Process all the regions in parallel.
        region[i].laplace (&(p–>region[i]));
    } // 'waitfor' the operations to complete for all regions.
  }
}
```

Figure 5: Concurrence and affinity in the Ocean code.

6.1 Ocean

Ocean is a program to study the influence of eddy and boundary currents on large-scale ocean currents. It simulates flows in a cuboidal basin over a series of time steps, solving a set of partial differential equations within each step. In this case study we illustrate how a simple object distribution together with the default affinity works well in improving data locality.

The main data structures are twenty-five double precision floating point grids. Each grid is a two-dimensional array, representing the value of a state variable at different locations in the ocean basin. The computation involves very regular intra-grid (e.g. nearest neighbor) and inter-grid (e.g. adding corresponding elements of two grids) operations. To exploit concurrency, both intra-grid and inter-grid operations we partition each grid into regions, and process different regions of the grid concurrently. This results in a uniform amount of work for each region, as well as minimal communication between processes working on different regions.

Figure 5 shows abstracted segments of the Ocean program expressed in Cool. The primary objects are regions and grids, with a grid composed of several regions. We chose to partition a grid into a single array of regions, although rectangular block decompositions are also possible. Concurrency within a grid operation is exploited by invoking a parallel function on each region within the grid. Synchronization for the completion of a grid operation is easily expressed by invoking the operations on the regions within a waitfor.

As part of initialization, the Cool programmer explicitly dis-
attributes the regions of the grids across the memories of the processors so that corresponding regions of different grids are allocated within the same local memory (see the function distribute). Actually, although we impose a region and grid structure in the program, for simplicity the underlying grid elements are still maintained as a two dimensional array. Regions are distributed by migrating the corresponding elements of the two dimensional array. Thus object distribution can also be easily incorporated into existing codes without requiring a major restructuring of the data structures, and illustrates the flexibility of the basic mechanism.

An ANL version of this code on DASH is still under development, hence we cannot compare the performance of the Cool program with the ANL version. However, we expect their performance to be similar.

LocusRoute is a parallel algorithm for standard cell placement and routing in integrated circuits. Given a circuit consisting of modules and connections, and a placement of the circuit modules, the program does routing to determine the paths of the connecting wires. The objective is to find a route that minimizes the area of the circuit.

6.2 LocusRoute

LocusRoute is a parallel algorithm for standard cell placement and routing in integrated circuits. Given a circuit consisting of modules and connections, and a placement of the circuit modules, the program does routing to determine the paths of the connecting wires. The objective is to find a route that minimizes the area of the circuit.
the wire, tries to find a better route, and updates the CostArray with the sum of the CostArray values for all the routing cells that it traverses.

portion of the CostArray spanned by the geographical end-points of concurrency is shown in Figure 9, and consists of the parallel function to route a wire.

main () {
  for (i=0; i<Number; i++) {
    waitfor {
      for (all wires 'W' do W->Route ());
      // Route the wires in parallel.
    } // Wait for all wires to be routed.
  }
}

Figure 9: Specifying affinity hints in LocusRoute.

The two main data structures are Wires and the CostArray. A wire object contains the list of pin locations to be joined, where a pin location corresponds to a routing cell in the circuit. The CostArray keeps track of the number of wires running through each routing cell, and contains two values: the number of wires that pass horizontally and vertically through the routing cell. These values are updated as wires are routed. The cost of a route is given by the sum of the CostArray values for all the routing cells that it traverses.

Figure 10: Performance improvements in LocusRoute with affinity hints.

The program iteratively converges to a route for each wire. Within each iteration, for every wire, LocusRoute rips out the previous route of the wire and tries to find a better route by reading the CostArray and examining the cost of alternate routes. If it finds a better route then it updates the CostArray with the new route. Synchronization for the completion of an iteration is expressed by wrapping a waitfor around an iteration.

The algorithm spends nearly all of its time in evaluating the cost of different routes for a wire, so locality on the CostArray is important. We can express this by viewing the CostArray as partitioned into geographical regions (see Figure 8), and identifying wires that lie within the same region of the CostArray as a task-affinity set. Executing these tasks back to back on a processor will hopefully reuse that region of the CostArray in the cache. To simultaneously distribute the load across processors as well, we actually exploit this locality using processor affinity: each region of the CostArray is (conceptually) assigned to a processor, and tasks within that region are directly scheduled on the corresponding processor. Besides reusing the same region in the cache, each region is likely to be referenced by only one processor, thereby reducing the invalidations of the CostArray in caches of other processors. Finally, depending on the degree of reuse, physically distributing the regions of the CostArray across the memories of different processors may further improve performance, since the misses to the CostArray will be to local rather than remote memory.

The code to express these hints is shown in Figure 9. The function Region(CurrentWire) computes the mid-point of the wire and returns the region number that the mid-point lies within, which is then used as the server number in the processor-affinity hint. Partitioning the CostArray into a few large regions (say one per processor) will have better locality but perhaps poorer load balance, while larger numbers of smaller regions will have better load balance at the expense of data locality. These tradeoffs can be easily explored in the Cool program by varying the Region function.

Since we had only small input circuits available to us, we demonstrate our technique using a synthetically constructed input consisting of a dense network of wires within regions of the circuit (we believe our results apply to larger realistic circuits as well). The performance results in Figure 10 show (a) the Base version of the program in which the wire tasks are scheduled across processors in a round-robin fashion without regard for locality, (b) the Affinity version in which processor affinity hint is supplied so that wires in a geographical region are likely to be routed by the associated processor, and (c) the Affinity+ObjectDistr version in which the regions of the CostArray are physically distributed across processors' memories as well. Overall speedups are small due to the high degree of communication of shared data. However, with the affinity hints most of the wire tasks (over 80%) in a region are routed on the corresponding processor, resulting in significant performance improvements. As shown by the cache miss statistics in Figure 11, affinity scheduling nearly halves the number of cache misses. Distributing the CostArray improves performance further, although the gains are smaller. The number of cache misses remain unchanged but more of them are serviced in local rather than remote memory.

In contrast to the Cool program that simply requires affinity hints, the ANL version requires the programmer to explicitly maintain task queues per processor, and manage task creation and scheduling based on the geographical locality described above. The Cool program outperforms the ANL code due to some additional optimizations for dynamic memory allocation. To factor out the effect of these optimizations, we present self-relative speedup for the ANL code in Figure 10. Even so, the Cool program exhibits better speedup, especially for larger number of processors.

While the actual affinity hints supplied are simple, it is important to realize that the hints are based on insights about the semantics of the application, that would be impossible for a compiler to deduce. For instance, a compiler simply cannot infer that wire objects have end-points that correspond to a region in the CostArray and have affinity for that region. Programmer intervention is therefore necessary for such programs.
6.3 Panel Cholesky

This program performs parallel Cholesky factorization of a sparse positive definite matrix [13]. That is, given a sparse matrix $A$, the program finds a lower-triangular matrix $L$, such that $A = LL^T$. The primary operation in a column-oriented sparse Cholesky factorization is the addition of a multiple of one column of $A$ into another column to its right in order to cancel a non-zero in the upper triangle. This operation is typically referred to as a column modification. Rothberg and Gupta [13] have suggested a matrix representation in which columns with identical non-zero structure are organized into panels, as shown in Figure 12. Operations are performed between panels — each panel has updates performed to it by relevant panels to its left, and once all the updates to a panel have been performed, the panel becomes ‘ready’, and can be used to update other panels to its right.

The Cool code expressing this computation is shown in Figure 13. The main procedure initially starts the computation by calling CompletePanel on the panels that are initially ready and do not require any modifications. This function performs some internal modifications, and then calls the parallel function UpdatePanel to update those panels to its right that are modified by this panel. Besides being a parallel function, UpdatePanel is also a mutex function since it requires exclusive access to the panel being updated. This ensures that multiple updates to a particular panel are serialized. Once all the modifications to a panel have been performed, the panel becomes ready, and CompletePanel is called on it.

```c
class panel { 
  ...
  public:
    parallel mutex void UpdatePanel (panel* src) 
      [ affinity (src, TASK); affinity (this, OBJECT)]; 
      { 
        ... Update this panel by the given src panel ...
        if (all updates to this panel have been performed) 
          // This panel is now ready.
          CompletePanel ();
      }

    parallel void CompletePanel () 
      { 
        ... Perform internal completion ...
        // Produce updates that need this panel,
        for (all panels ‘p’ modified by this panel) 
          panel[p].UpdatePanel (this);
      }
  } *panel;

  main () 
  { 
    // Distribute panels across processors memories,
    // in a round robin fashion.
    for (p=0; p<MaxPanels; p++) migrate (panel+p, p);
    waitfor 
      // Start with the initially ready panels.
      for (all panels ‘p’ that are initially ready) 
        panel[p].CompletePanel ();
  } // Wait for all updates to complete.
```

Figure 13: Expressing concurrency and affinity in Panel Cholesky.

Most of the work is done in UpdatePanel, which reads the source panel and modifies the destination panel. By default, tasks corresponding to UpdatePanel have affinity for the panel that they are invoked on (the destination panel), and are automatically scheduled to exploit both cache reuse and memory locality on the destination panel. In addition, by distributing the panels across processors’ memories we can both distribute the associated work and distribute the memory bandwidth requirements.

This scheme should perform well. Since the destination panel is being modified, it is preferable that modifications to a panel be performed on the processor that contains the destination panel in its local memory; otherwise the destination panel will continually be
due to a better distribution of the memory bandwidth requirements. As shown in the figure, the performance of the final code is less than 10% slower than a hand-coded version of the code written using the ANL macros. Figure 15 displays the effect of the optimizations on the cache miss behavior of the program. Simply distributing the panels improves performance due to better utilization of the available memory bandwidth without affecting the cache performance. However, affinity scheduling and cluster scheduling significantly reduce the number of cache misses; in addition, since the tasks are collocated with the panel, more of the misses are serviced in local rather than remote memory.

We now discuss an experiment we did with cluster-based scheduling. This experiment demonstrates the benefits of exploiting specific architectural features in improving data locality; the necessary optimizations, however, may require additional programmer abstractions and/or runtime support. Recall that although tasks are collocated with the destination panel on a particular processor, all processors within that cluster (in DASH) share the local memory containing the destination panel. We therefore ran the program with an idle processor allowed to steal tasks only from other processors within the same cluster; the stolen tasks would therefore continue to reference the destination panel in local rather than remote memory. This effect is controlled through a runtime flag that can be dynamically manipulated by the programmer. As shown by the plot Distr+Aff+ClusterStealing, stealing only within a cluster further improves performance.

Although in our experiments cluster scheduling was explicitly manipulated by the programmer, perhaps this effect can be automated within the runtime system. For instance, a runtime scheduler could always try to steal tasks from processors within the same cluster before trying to steal tasks from remote processors. Or, the scheduler may decide to steal tasks from remote processors based on the relative latencies of local and remote memory references. However, more experience is required before good defaults or automatic strategies can be developed; this example simply demonstrates the usefulness of such mechanisms.

As shown in the figure, the performance of the final COOL code is less than 10% slower than a hand-coded version of the code written using the ANL macros. Figure 15 displays the effect of the optimizations on the cache miss behavior of the program. Simply distributing the panels improves performance due to better utilization of the available memory bandwidth without affecting the cache performance. However, affinity scheduling and cluster scheduling significantly reduce the number of cache misses; in addition, since the tasks are collocated with the panel, more of the misses are serviced in local rather than remote memory.

6.4 Summary of other Applications

We have obtained similar good results with other SPLASH applications. Here we briefly summarize our experience with two applications, Barnes-Hut and Block Cholesky [14] (see Figure 16). Barnes-Hut is an N-body application, while the Block Cholesky program performs Cholesky factorization by representing the matrix as a set of blocks (instead of panels as studied in Section 6.3). Both

---

6We present results on up to 24 processors due to limitations in the amount of physical memory on the machine.
these applications illustrate how the abstractions in Cool allow the programmer to easily explore several alternatives that tradeoff between locality and load balance. By simply changing the affinity hints the programmer could try different optimizations for locality, and depend on the runtime to dynamically correct any load imbalance. The Cool version of Barnes-Hut performs close to the hand-coded ANL version, while Block Cholesky in Cool gets even higher performance than the ANL program due to better load balancing.

7 Related Work

In this section we begin with a brief outline of automatic techniques to address data locality in programs. We then describe other approaches based on explicit programmer support, and contrast them with Cool.

Automatic Techniques: Compiler based approaches [8, 16] primarily address loop-level parallelism in regular numerical algorithms, and include optimizations such as loop transformations to improve reuse in the cache, and array alignment and distribution for better memory locality. However, these approaches focus on affine array accesses within a loop nest, while our target is a much broader class of programs with complex data reference patterns.

Operating system techniques [2, 9] address data locality in NUMA machines through smart page placement policies such as allocating private data in the local memory of a processor, allocating modified data in common global memory if any, and replicating those pages that are referenced but not modified (replicating pages in softwares requires that the system keep the multiple copies of a page consistent). Although such schemes don’t require programmer intervention, their primary drawback is the limited amount of information available based on which pages must be distributed. In addition, these schemes are based only on page distribution to improve memory locality, and do not consider techniques such as task scheduling that can improve cache locality.

Explicit Programmer Hints: Along with other researchers, both Fortran-D [6], and Rogers and Pingali [12] have taken a data-centric view towards writing parallel programs in machines with distributed physical memory. The programmer writes a serial shared memory program and specifies a data distribution across processors; explicit constructs for predefined regular patterns as well as for irregular distributions are provided. The parallelism is automatically inferred by the compiler based on an owner-computes strategy. The compiler generates code for each processing node along with the necessary communication and synchronization.

This approach enables the programmer to write serial shared memory programs, and in particular relieves the programmer from having to specify the low level communication and synchronization. However, since the extraction of parallelism is driven by the data partitioning, the programmer needs to consider the effect of specified data partitioning on the automatic parallel decomposition. In particular, if the compiler makes some wrong decisions then the programmer has only limited and indirect control through the data decomposition specification. Finally, their research focuses on loop-level parallelism in programs with highly regular structure, while we are interested in task level parallelism in programs with irregular and complex structures.

Research done concurrently with us includes Fowler [5], who provide a parallel programming environment with light-weight threads and continuations (for synchronization). The programmer can influence the scheduling of threads through object-affinity scheduling: objects derived from a base class uses_OAS have a field that stores a processor number, with methods for reading and writing this field. Tasks on an object derived from uses_OAS are scheduled on the processor specified by the field. By default, tasks are scheduled on a global queue, and acquire affinity for the processor that steals them once they get stolen. In addition, tasks or groups of tasks may also be scheduled on the local queue of a processor.

Their work is done in the context of SGI multiprocessor workstations, that have per processor caches but a single global shared memory. Therefore they focus on optimizations for cache reuse, such as scheduling tasks on the same object on the same processor, and scheduling tasks in a template group (i.e. tasks related through common synchronization) together on a processor. However, they do not consider the temporal aspect of cache reuse. Each processor has a single task queue that is simply serviced in FIFO order. In contrast, temporal reuse is explicitly addressed in Cool by identifying related tasks through task affinity for back to back execution on a processor. In addition, Cool provides support for exploiting memory locality on NUMA machines.

8 Concluding Remarks

The high latency of memory references in multiprocessors significantly hinders the efficient execution of parallel programs. In this paper we have shown how scheduling computation and distributing objects, with an understanding of the underlying memory hierarchy, can improve data locality. We develop an abstraction of the mem-

Figure 16: Performance improvement with affinity hints for Barnes-Hut and Block Cholesky.
ory hierarchy of a shared-memory multiprocessor, that captures the essential components of the memory system. To exploit locality in this abstract memory hierarchy, we develop support in COOL for the programmer to supply hints about the program, and scheduling mechanisms in the runtime system that use these hints to distribute tasks and objects appropriately. Through several realistic examples we have shown how the hints can be naturally supplied within the task and object structure of the COOL program. The hierarchy of abstractions provides an increasing degree of control, while the flexibility of the abstractions makes it easy to experiment with different scheduling optimizations. The hints are highly effective, with performance improvements as high as 135%.

An important question that arises from this work concerns the feasibility of automating these techniques in a compiler. Are programmer supplied locality hints necessary in the long-term, or are they only a stop-gap measure that will be made obsolete with improvements in compiler technology? Our experience with the applications indicates the former. For instance, in the LocusRoute application, the locality hints are based on the semantic understanding of ‘wires’ (e.g. their 2-D spatial nature and that they are short), the ‘cost array’, and the relationship between the two. We do not believe that a compiler could automatically infer such semantics. At the same time, however, experience with programs such as Panel Cholesky and Ocean suggests that smart default strategies for object distribution and task scheduling can significantly reduce the burden on the programmer.

Our ongoing research focuses on experimenting and evaluating different runtime heuristics, supporting affinity for multiple objects, runtime support and abstractions for prefetching, and algorithms for automatic object distribution.

Acknowledgements: The Barnes-Hut code was developed by J. P. Singh and programmed in COOL by Anveesh Agrawal. Sparse Cholesky (panel and block) codes were developed by Edward Rothberg, and programmed in COOL by Maneesh Agrawal. This research has been supported by the Defense Advanced Research Projects Agency under DARPA contract #N00039-91-C-0138. Anoop Gupta is also supported by an NSF Presidential Young Investigator Award.

References


