Runtime System Support for Running Applications with Dynamic and Asynchronous Task Parallelism in Software DSM Systems*

Rafael Mendes
COPPE-UFRJ
rmendes@lcp.coppe.ufrj.br

Lauro Whately
COPPE-UFRJ
whately@cos.ufrj.br

Maria Clicia de Castro
IME-UERJ
clicia@ime.uerj.br

Cristiana Bentes
DESC-UERJ
cris@eng.uerj.br

Claudio Luis Amorim
COPPE-UFRJ
amorim@cos.ufrj.br

Abstract

State-of-the-art software distributed shared-memory systems (SDSMs) provide a cost-effective solution to run single-program-multiple-data (SPMD) applications on clusters of distributed memory computers. However, SDSMs are unsuitable for running applications with dynamic, highly asynchronous task parallelism (ATP), such as graphics, simulators, and decision support systems. In ATP-based applications, the execution of tasks depends not only on the input data but also on the variable amount of data that each task produces at runtime, which generates high load imbalance and communication traffic that degrades performance of DSM systems drastically. In this work, we propose a new load balancing (LB) mechanism to enable SDSM systems to support dynamic task scheduling as required by ATP applications. To evaluate the benefits of our LB mechanism, we developed Clik a new multi-threaded SDSM system with automatic load balancing. Our preliminary performance results of Clik running on a 16-node Linux SMP cluster for five ATP applications showed that Clik attained significant speedups. For four of our five applications, the speedups varied from 7.2 up to 13.8 on 16 processors.

1 Introduction

Clusters of distributed memory computers have increasingly become an alternative high-performance computing platform for a broad range of scientific and engineering applications [1]. At the same time, many research efforts have been made to efficiently support shared-memory program-

*This work was supported in part by HP Brazil R&D, and the Brazilian agencies FINEP, CAPES and CNPq.

ming on clusters, known as software distributed shared-memory systems (SDSMs).

Currently, state-of-art DSM systems can attain considerable scalable performance only on single-program-multiple-data (SPMD) parallel applications that exhibit regular access patterns to coarse-grain shared data. In SPMD programming style, the programmer decomposes the application into parallel tasks with different data domains and distributes them over the processing nodes statically before the application starts. As a result, the use of DSM systems has been limited to run well-structured numerical applications that conform to the SPMD model.

In this work, we concentrate on developing a runtime system that allows DSM systems to cope with another important class of applications such as graphics, simulators, and decision support systems that have great potential for cluster computing. Such applications exhibit dynamic, highly asynchronous task parallelism (ATP) behavior and as expected, show disappointing performance when submitted to DSM systems. The root cause of DSM’s low performance is the fact that the execution of tasks in ATP applications depends not only on the input data but also on the variable amount of data that each task produces at runtime, which generates high load imbalance and communication traffic that ultimately affect performance dramatically. With this in mind, we introduce Clik, a new multi-threaded DSM system with load balancing support to run ATP-based applications. The main features of current Clik implementation are as follows: (i) support for ATP multi-threaded shared-memory programming; (ii) automatic and dynamic load balancing; and (iii) run on clusters of SMP processing nodes.

Clik programs are written in Cilk [6], a language for multithreaded parallel programming based on ANSI C, tailored for exploiting ATP applications. Clik allows the programmer to concentrate on exposing the parallelism of an
application while leaving the burden of task scheduling and load balancing to the runtime system. In regard to memory consistency, ATP applications require that the SDSM system guarantees memory consistency only between pairs of threads that synchronize. Our preliminary performance results of Clik running on a 16-node Linux SMP cluster for five ATP applications showed that Clik attained significant speedups. For four of our five applications, the speedups varied from 7.2 up to 13.8 on 16 processors.

The remainder of this paper is organized as follows. In the next section, we discuss related works. In Section 3, we briefly review the Cilk language. In Section 4, we describe the mechanisms that Clik implements in order to run efficiently ATP applications. Section 5 reports our experimental results. Finally, in Section 6 we present our final remarks and outline ongoing works.

2 Related Work

The Cilk language was designed for general-purpose parallel programming, but it is especially effective for exploiting dynamic, highly asynchronous parallelism, which can be difficult to write in SPMD or message-passing style. The MIT Cilk project [13] released software of a runtime system to run Cilk programs only on a single SMP platform. Our work differs from MIT’s Cilk software in two important ways. First, though Clik has been built on the Cilk’s distribution of single SMP code, Clik implemented practically a complete new distributed multithreaded SDSM system on top of it with transparent load balancing across the cluster nodes. Second, Clik enables Cilk programs to run not only on a single SMP node but also on a variety of cluster platforms ranging from single-processor nodes, SMP nodes, or a combination of both.

We note that Clik is not the first distributed implementation of the Cilk language. Randall [14] proposed an implementation using the dag-consistency model [4], where memory consistency was enforced along the thread dependency sequence. The same work reported some theoretical evidences of potential performance of the implementation proposed, but few practical results were presented. SilkRoad II [12] implemented two memory consistency models: dag-consistency and lazy release consistency. Clik uses a home-based approach that allows home nodes to have up-to-date versions of the pages assigned to them, while SilkRoad II uses an invalidate homeless coherence protocol. None of those two distributed Cilk implementations have been released so we could not make any performance comparison between the distributed Cilk implementations. The MIT group also has a distributed version of Cilk available on the Web, which did not work as expected.

3 ATP Programming Model

Clik implements the ATP model whose dynamic task parallelism follows the model originally proposed by the Cilk language [2]. Cilk is particularly suitable for recursive programming, easing the implementation of the common divide-and-conquer model.

Clik extensions to the C language are very simple, as it includes only a few directives to implement the fork-join model and no other new data type. Indeed, once Clik directives are removed from the source code, the result is a valid sequential C program. Two directives are all that are needed to start using the parallel features of Cilk:

- spawn - to create a thread dynamically (similar to a fork invocation). This directive only indicates that the thread can execute in parallel, while leaving to the scheduler the decision whether the thread will actually run in parallel or not;

- sync - to indicate that the execution of the current thread cannot proceed until all previously spawned threads have been completed (similar to a join invocation).

The last version of the Cilk language also has lock/unlock primitives to protect shared data access in critical sections. In Figure 1, we show an example of a parallel code using Cilk language with spawn and sync primitives. The code is a recursive implementation of the Fibonacci problem, in which the divide-and-conquer programming model is used to find the $n$ Fibonacci numbers. In Figure 1, when $n > 2$ two new parallel threads are created using the spawn primitive to solve recursively $\text{fib}(n-1)$ and $\text{fib}(n-2)$. In contrast with an ordinary function call where the parent thread suspends itself until its child thread has returned, in a Cilk spawn, the parent thread can continue to execute in parallel with its child thread. Actually, the parent thread can continue to spawn other child threads, producing a high degree of task parallelism. A sync primitive should be placed before the statement $\text{return (x+y)}$, so that the parent thread is forced to wait for its child threads to complete before summing x and y.

4 The Clik Runtime System

In this section, we describe the mechanisms that Clik implements to guarantee memory consistency and dynamic load balancing. We also show the improvements we made to the preliminary Clik prototype [9] to overcome its major performance bottlenecks to run ATP applications.
cilk int fib (int n)
{
    if (n < 2) return n;
    else
    {
        int x, y;
        x = spawn fib (n-1);
        y = spawn fib (n-2);
        sync;
        return (x+y);
    }
}

Figure 1. Fibonacci in Cilk

4.1 Clik Overview

Clik is a multithreaded SDSM system that implements the concept of virtual processors, called the workers, where each worker is responsible for the sequential execution of a set of runnable threads. In addition, Clik allows the user to define the number of workers per node that the runtime system will use. Clik activates automatically multithreading when the user specifies multiple workers per node. In this way, users can manage the level of multithreading that is useful for a particular application. Clik starts the defined number of workers on each node, where each worker has its own thread queue. Initially, all thread queues are empty, except for the worker 0’s queue, which will hold the first thread to run. As soon as the first thread starts and creates dynamically new child threads, Clik runs a work-stealing algorithm to balance the load among the available workers as will be shown in Section 4.4.

Clik associates a new frame with every thread the application creates. A frame is a data structure that holds the state of a thread, e.g., global variables and parameters. Clik stores the frames in the distributed shared memory so that when a thread migrates to a new node the memory coherence protocol guarantees that all updates occurred in the old node will be propagated to the new one. In this way, Clik avoids to checkpoint before a thread could migrate.

4.2 Memory Consistency

Clik implements the Direct-Acyclic-Graph(DAG)-consistency model proposed by the Clik language [3]. DAG-consistency is defined on the DAG of threads that make up a parallel computation. Intuitively, a read can ”see” a write in the DAG-consistency model only if there is some serial execution order consistent with the DAG in which the read sees the write. However, Clik implements its own memory coherence algorithm to enforce the DAG-consistency memory model. This algorithm comes from the authors past experience [16, 10] with Home-based Lazy Release Consistency (HLRC) protocols. HLRC employs a relaxed memory consistency model stronger than DAG-consistency as well as a multiple-writer protocol to reduce the overheads due to both remote communication and maintaining memory coherence in a distributed system. In Clik, the modifications that each worker makes locally to a shared page are propagated to its home node when either one of the following events occurs: (i) a new thread is created, and (ii) a thread performs a join operation. Future accesses to the page require that a copy of the page be retrieved from the home node. The advantage of using home nodes is to improve scalability of a software DSM system since they tend to reduce the number of messages and memory space overheads when compared to homeless SDSM systems, such as TreadMarks [7].

The Clik coherence protocol propagates invalidations using write-notices while using synchronization intervals between pairs of fork and join operations on threads to establish a partial order of writes to the shared data. Modifications to a certain page are sent to its home node in the form of diffs (the resulting difference between the actual page and its original contents), which allow for multiple concurrent writes to the same page to occur in different processing nodes. When a page-fault occurs, the page will be requested to its home node only if no other worker in the same node has already requested the page neither it has a valid copy of it.

Clik implements a dynamic thread scheduler responsible for assigning threads to the cluster nodes and redistributing them for load balancing purposes. Clik relies on its memory coherence protocol to keep shared data updated in order to support thread migration. Clik coherence protocol allows a thread a after completing its execution to resume execution of another thread b in a different node that was blocked waiting for a to finish. This is possible because the Clik protocol uses new coherence operations to keep the shared data valid between threads a and b.

4.3 Synchronization

Clik supports only the sync primitive for thread synchronization. When thread t in node j executes a sync, Clik performs coherence operations by sending the diffs related to the modified pages to their respective homes. After that, either one of the following three cases can occur: (i) if t has other child threads, then t is suspended; (ii) if t was created in node k and after that t migrated to j, then thread t is sent back to k (its frame is set valid only in k); and (iii) if t does not have child threads or it has not migrated, t continues running.
4.4 Load Balancing in Clik

Clik integrates a load balancing mechanism with the runtime system, so that load balancing in Clik is completely transparent to the user and it is key to Clik’s performance, which is discussed in the next section. In ATP applications the amount of threads to be created is unpredictable, thus an efficient load balancing runtime system like the one we present next is most needed.

4.4.1 Work Stealing

Clik implements a distributed work-stealing (WS) algorithm where a worker whose thread queue becomes empty will try to steal threads from the thread queues of other workers. The WS algorithm first tries to steal threads from workers that execute in the same processing node, thus avoiding unnecessary message traffic across the network. Otherwise, the WS algorithm starts a remote work-stealing. In this case, the idle worker \( w \) sends a message to the victim node \( v \) to request work. If the \( v \) node has threads to be stolen, it sends back to \( w \) not only the stolen thread, but also the write-notices of all shared data modifications that \( w \) has not seen. Upon receiving the new thread, \( w \) inserts it in its thread queue to be scheduled afterwards.

4.4.2 Choosing the Victim node

A key aspect of the WS algorithm is the best choice of the victim nodes since that depending on the chosen nodes the number of messages to request extra work that workers exchange can increase greatly. To investigate the best WS algorithm, we implemented in Clik three different policies for selecting the victim node, namely Random, Most Overloaded, and Affinity.

The Random policy is the simplest one and was previously implemented in other Clik-based systems [6, 12], and also in a preliminary version of the Clik system [9]. In the random policy, the stealing worker chooses randomly the victim node to which it sends a message to request work. In case of the chosen node has no work available, the stealer repeats the procedure until it finds an appropriate victim node. Since the random choice does not take into account the load level in each node, so even when nodes have emptied their thread queues, they keep receiving messages from other nodes unnecessarily. As a result, the random policy often wastes network bandwidth.

The Most Overloaded policy tries to improve the random policy by spreading information on the load level of every node across the network. The central idea is to piggyback each regular message that Clik transmits between processing nodes (e.g., to send diffs, write-notices, or sync messages) with load level information on the nodes involved. One advantage of using regular messages is to avoid the high cost of spreading load level information through periodic broadcasts using extra messages. In the most overloaded policy the stealing worker chooses the victim node by looking in its load-level information list for the most overloaded node. Although such a list may be outdated, in practice it provides a good hint for choosing the best victim node. Another advantage of this policy is that it allows Clik to block the stealing worker when there is no work left on any other node. In contrast, the random policy cannot block a stealer since nodes share no load level information among themselves.

The Affinity policy is similar to the most overloaded one. The only difference is that the affinity policy tries to choose first the same victim node it has chosen on the last work-stealing attempt. The idea behind this policy is to create groups of nodes with stealing affinity. In this way, Clik can improve the shared data locality, since the stealing node and the victim node would tend to share similar page working sets.

5 Experimental Results

In this section, we evaluate performance of Clik running on a cluster of 16 SMP nodes interconnected by a Gigabit Ethernet switch, where each node consisted of two 2 GHz AMD Opteron processors and 1 GB RAM. Clik was implemented in C and runs on Linux kernel 2.6.7, using sockets and UDP protocol.

5.1 Workload

We used in our experiments five different ATP applications: Matmul, MergeSort, LU, PZSweep and Knapsack. Matmul, MergeSort and LU are well-known kernels widely used to evaluate performance of distributed shared-memory systems. PZSweep is a graphics application for rendering volumetric image data. Knapsack is a combinatorial optimization application. Next, we give a brief description of each application and show in Table 1 the input sizes used.

5.1.1 Matmul

Matmul computes the product of two \( n \times n \) matrices, \( A \) and \( B \), and writes the result in another matrix \( C \). This application uses the divide-and-conquer programming paradigm. The \( A \) and \( B \) matrices are recursively divided until they reach the minimum trivial size. In each recursion step, two \( A' \) and \( B' \) (smaller than the original) matrices are generated and the multiplication is performed in \( A' \) and \( B' \) in parallel.

5.1.2 MergeSort

MergeSort is a sorting algorithm that orders an array of \( N \) keys. MergeSort is a recursive sorting algorithm that uses
the divide-and-conquer programming paradigm in the following way. It splits the list to be sorted into two equal halves, and places them in separate arrays. Each array is recursively sorted, and then merged back together to form the final sorted list.

5.1.3 LU

LU is an algorithm for decomposing an $N \times N$ matrix $A$ into a product of a lower triangular matrix $L$ and an upper triangular matrix $U$, $LU = A$. We used a divide-and-conquer algorithm that can be easily expressed in the ATP programming model. The matrix to be decomposed is divided into four blocks, $B(0,0)$, $B(0,1)$, $B(1,0)$ and $B(1,1)$. Initially a partial LU decomposition is performed recursively for $B(0,0)$. When $B(0,0)$ computation finishes, two partial LU decompositions are performed recursively in parallel for $B(1,0)$ and $B(0,1)$ blocks. Having obtained the results for these decompositions, another LU decomposition is performed recursively on $B(1,1)$ block.

5.1.4 PZSweep

PZSweep is an out-of-core volume rendering application. The algorithm is an image-space parallelization of the sequential ZSweep algorithm [5]. PZSweep breaks the screen into tiles and each tile represents a computational unit of work. There is one thread for the computation of each tile. The rendering is done using the sweeping plane paradigm, where a plane is swept in order of increasing $z$. Each time a plane intersects a vertex, all of its incident faces are projected onto the screen. We used the out-of-core version of PZSweep, that allows the rendering of very big datasets that do not fit in main memory. The out-of-core version performs frequent I/O operations, in order to bring the dataset portion to be computed. The images generated have $512 \times 512$ pixels.

5.1.5 Knapsack

Knapsack is a NP-complete problem in combinatorial optimization, usually used in business, combinatorics, complexity theory, cryptography, and applied mathematics. It derives its name from the maximization problem of choosing as much as possible essentials that can fit into one bag (of maximum weight) carried on a trip. Given a set of items, each with a cost and a value, then determine the number of each item to include in a collection so that the total cost is less than some given cost and the total value is as large as possible. The application provided by Cilk suite is known as 0-1 knapsack problem or binary knapsack problem, using a branch-and-bound technique. The 0-1 knapsack problem is a special instance in which all the lower bounds are equal to zero and all the upper bounds are equal to 1.

<table>
<thead>
<tr>
<th>Application</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matmul</td>
<td>2048 $\times$ 2048</td>
</tr>
<tr>
<td>MergeSort</td>
<td>20M keys</td>
</tr>
<tr>
<td>LU</td>
<td>4096 $\times$ 4096</td>
</tr>
<tr>
<td>Knapsack</td>
<td>80 objects/1000 bag capacity</td>
</tr>
<tr>
<td>PZSweep</td>
<td>SPXI/103K cells</td>
</tr>
</tbody>
</table>

5.2 Clik Speedups

Figures 2 and 3 show the speedups relative to a single processor execution of our workload running under Clik with 2 workers in each SMP node. Clik speedups were plotted only for the load balancing strategy that produced the best result for each application. A main advantage of Clik is to allow the programmer to choose the load balancing strategy that fits best his/her application characteristics. This is the reason why we consider the best load balancing policy on analyzing Clik performance. In Section 5.4 we will evaluate the effects of the load balancing strategies on the performance of Clik for each application tested.

As we can observe in Figure 2, PZSweep, MergeSort and Matmul achieved good speedups in Clik. The speedups for 16 processors were 13.8 for PZSweep, 9.4 for Matmul and 11.0 for MergeSort. PZSweep is the application that presented the lowest communication overhead, because the computation of each tile is independently performed and the shared data is updated only in the end of the tile computation. This application, however, is very sensitive to load balancing, since each tile presents very different computational load. The speedups showed that the Clik load balancing scheme performed very well for PZSweep. This result is in turn quite significant since that in graphics applications the programmer usually has to implement its own load balanc-
ing scheme. Therefore, Clik offers an efficient and transparent load balancing scheme that alleviates the burden of the graphics application programmer. MergeSort also showed good speedups. This result comes from the fact that the recursive division of the array to be sorted provided a fair distribution of the computational load among Clik workers. The merge phase, however, included communication and some load imbalance, since after each merge step, half of the active workers became idle. Matmul is a simple application, that performed well in Clik, but it transferred a lot more data than both PZSweep and MergeSort.

The speedups of LU and Knapsack, shown in Figure 3, are smaller than the ones obtained for the other applications. For 16 processors, Clik produced speedups of 7.2 for Knapsack and 4.3 for LU. Knapsack showed good speedups for a small number of processors, but attained more limited scalability because the problem was divided into small pieces, where the computational load at each process was not large enough to amortize the communication load. LU is the application that presented the lowest speedups. This can be explained by the poor data locality that LU exhibited, which increased the amount of remote pages and associate communication traffic to the home nodes. In addition, LU had inherent load imbalance because the threads had to wait for the pivot computation to finish.

5.3 Multithreading Effect

The effects of Clik multithreading had positive performance impact on some applications. In Figure 4, we show Clik speedups for PZSweep running on a cluster of 16 processors, where each processor was configured with 2 and 4 workers. In the former, each thread was assigned to one processor whereas in the latter, there were 2 threads competing for each processor. The latter usually generated more overhead due to competition. However, when the competing threads overlapped I/O with computation, the application executed faster, as one thread was always running in each processor. This is the case of PZSweep for small number of processors. For large number of processors, however, given that PZSweep is an out-of-core application, then doubling the number of workers increased the load imbalance, because there were less tiles to be stolen.

For Knapsack, Clik performance with 4 workers was almost the same as that of with 2 workers. This could be explained by the unpredictable characteristic of the application. The workload of such an application depends on how it is scheduled, as it often requires that the result of a sub-problem to be recalculated. In this case, by using more threads we increased the probability of a thread to prune a sub-problem recalculation.

5.4 Load Balancing

Table 2 shows the percentage (%) of load imbalance obtained for each application running on Clik using the three different load balancing strategies: Random, Most Overloaded and Affinity. The percentage of load imbalance was calculated according to [8]: 1 - \( \frac{t_{\text{avg}}}{t_{\text{max}}} \), where \( t_{\text{avg}} \) is the average computation time and \( t_{\text{max}} \) is the maximum computation time for all processors. As we can observe in the table, the results for Clik’s load balancing strategies were quite different for each application. For the Most Overloaded policy the lowest load imbalance results were obtained in MergeSort and LU whereas for the Random policy the lowest load imbalance results were obtained in PZSweep, Matmul, and Knapsack. Also, as shown in table Table 2 Clik achieved the lowest load imbalance rate at around 10% for the best strategy of each application, except for LU in which the pivot computation phase was responsible for the load imbalance.

Figures 5 to 9 show the speedups for the five applications running on Clik, using the three load balancing strategies. Those results revealed the impact of the load imbal-
Figure 5. Matmul speedups for three load balancing policies.

Figure 6. MergeSort speedups for three load balancing policies.

Figure 7. LU speedups for three load balancing policies.

Figure 5. Matmul speedups for three load balancing policies.

Figure 6. MergeSort speedups for three load balancing policies.

Figure 7. LU speedups for three load balancing policies.

6 Conclusions

In this work, we proposed and evaluated a new load balancing (LB) mechanism that enables SDSM systems to support applications with dynamic, highly asynchronous task parallelism (ATP). We evaluated the benefits of our LB mechanism in Clik, a multithreaded SDSM system that we developed. Clik programs are written in Cilk, a language for multithreaded parallel programming based on ANSI C that allows programmers to easily describe parallel recursive applications that use the divide-and-conquer paradigm. Our preliminary performance results using different load balancing policies showed that Clik attained significant speedups from 4.3 up to 13.8 on 16 processors, for 4 out of 5 ATP-based applications we run on a 16-node SMP cluster.

We will continue work on Clik to efficiently support the traditional SPMD programming model as well. Although SPMD applications may be rewritten to the ATP model, our
experience showed that conversion between the two models was not trivial. A preliminary analysis of Clik showed that it also can support SPMD applications to which load balancing is unnecessary and distributed synchronization primitives such as distributed locks and barriers suffice. The reasons are that Clik allows to switch off its load balancing mechanism as well as to easily extend the coherence mechanism to support the lazy release consistency that state-of-art SDSMs adopt.

References


Figure 8. PZSweep speedups for three load balancing policies.

Figure 9. Knapsack speedups for three load balancing policies.