A Study of Memory-Aware Scheduling in Message Driven Parallel Programs

Isaac Dooley, Chao Mei, Jonathan Lifflander, Laxmikant V. Kale

University of Illinois, Urbana IL 61801, USA

Abstract. This paper presents a simple, but powerful memory-aware scheduling mechanism that adaptively schedules tasks in a message driven parallel program. The scheduler adapts its behavior whenever memory usage exceeds a threshold by scheduling tasks known to reduce memory usage. The usefulness of the scheduler and its low overhead are demonstrated in the context of an LU matrix factorization program. In the LU program, only a single additional line of code is required to make use of the new general-purpose memory-aware scheduling mechanism. Without memory-aware scheduling, the LU program can only run with small problem sizes, but with the new memory-aware scheduling, the program scales to larger problem sizes.

1 Introduction

It is well known that some parallel algorithms require large quantities of memory. Unfortunately, parallel systems have limited amounts of memory, and hence parallel programs must use algorithms that do not exceed the available memory bounds.

This paper describes a general purpose memory-aware scheduling technique that can automatically restrict the memory usage for a class of parallel algorithms that would otherwise run out of memory. Because the scheduling technique is included in a general purpose parallel runtime system, the parallel program needs only minor changes to use the scheduler.

Often it is easier to implement a simple naïve algorithm instead of a more complicated explicitly memory-aware algorithm. The productivity of a programmer will likely be higher if a simpler memory-oblivious algorithm can be written while allowing the runtime system's scheduler to restrict memory consumption.

The memory-aware scheduling scheme described in this paper could be used for different programs. The scheduling scheme effectively reduces the memory requirements for an algorithm implemented in a manner that is mostly oblivious to its own memory usage. Specifically, this paper demonstrates the scheduler's utility in the context of an LU dense matrix factorization program.

All the scalable LU dense matrix factorization schemes frequently used today are written using algorithms that explicitly restrict the progress of tasks to ensure that there always is enough memory available to make forward progress. Some algorithms, such as the one used in the High Performance Linpack implementation use a fixed parameter that statically controls the *lookahead depth*, or number of algorithm stages that can be executed ahead of the oldest currently executing algorithm stage [2]. When the amount of lookahead permitted is small, the degree of concurrency is small and the required memory buffer overhead is small. Conversely, if the amount of lookahead permitted is high, the degree of concurrency is high but the required memory footprint becomes large. The expanded memory footprint stores more blocks of incoming data blocks sent to each processor before pairs of these blocks are consumed in a trailing update operations.

Other LU implementations use dynamic lookahead so they can fully exploit as much concurrency as will fit in the available memory [6]. Memory buffers are reserved for specific tasks in a certain order and sending and receiving processors coordinate the accesses to the reserved buffers to ensure that deadlock will not occur if memory is exhausted for some processor. Such implementations use an application specific scheduler with a user-level threading package to allow the program to proceed in a safe manner.

One key goal that all implementations share is to achieve high performance. This can be achieved by performing computation aggressively along the critical path so that the parallel machine achieves high utilization. A message driven style of programming such as Charm++ [8] allows this pattern of computation to be expressed naturally. The case study presented in this paper, an LU implementation, was written in Charm++.

2 Memory Aware Scheduling

This paper describes a memory-aware scheduler that can constrain the memory consumption of a class of naïve parallel algorithms that are oblivious to memory consumption. The memory usage is reduced by the scheduler as it chooses to schedule tasks known to reduce the memory footprint whenever available memory resources are low. This new scheduler was created by modifying the existing scheduler in the Charm++ Runtime System. The new scheduler can therefore be used by any Charm++ program, and hence it is general purpose. In order for the scheduler to know which tasks should be scheduled when memory resources are limited, the system requires only minor changes to the Charm++ program. The programmer simply needs to add an annotation for each of the tasks that reduce memory consumption. This section describes the existing Charm++ scheduling system and the modifications that result in a simple memory-aware scheduler.

2.1 Existing Charm++ Scheduler

The existing Charm++ Runtime System uses a flexible scheduling mechanism to execute tasks spawned locally and tasks associated with incoming messages from other processors. In a Charm++ program, the tasks are *Entry Method Invocations* on *Chare Objects*, *Chare Groups*, or *Chare Node Groups* [10]. The flow of control for a Charm++ parallel program proceeds as entry methods are invoked. These entry methods perform computations and asynchronously invoke other entry methods.

The existing scheduler in the Charm++ Runtime System, which runs on each processor, supports prioritized execution in both LIFO and FIFO modes. Priorities or LIFO/FIFO designations can be associated with each entry method invocation. If no priority is specified, a default medium priority is implicitly assumed. When an entry method is invoked, its designated queuing scheme is stored along with any parameters to the method inside a message. Each message is then delivered to the destination processor or processors. Each destination processor will enqueue the message using the queuing scheme specified in the message's header.

Although the primary Charm++ scheduler queue acts just like a priority queue, it is actually composed of three data structures: a high priority heap, a default (or zero) priority queue, and a low priority heap. Charm++ entry method invocations are stored in messages recorded in one of these three data structures. The reason that three separate structures are used instead of a single priority queue is that the double ended queue used for the frequent default priority case can be slightly faster than a more complicated heap data structure.

2.2 New Adaptive Charm++ Scheduler

The new adaptive scheduler is a simple variant of the existing scheduler. The new scheduler adapts its behavior whenever the current memory usage for the processor exceeds a threshold. The threshold can be specified at runtime as a command line argument.

As long as the current memory usage is below a threshold, the scheduler acts as it normally would, processing messages one at a time in prioritized order from the primary scheduler queue. When the current memory usage exceeds the specified threshold, certain types of tasks are scheduled immediately even though they might have priorities lower than other tasks in the queue. Specifically, tasks that potentially reduce memory usage will be scheduled ahead of all other tasks whenever a processor's memory usage exceeds the threshold.

To modify the behavior of the scheduler when the memory usage is high, a call is made to a function that modifies the scheduler queue just prior to determining which task ought to be executed next. The modification function simply performs a linear scan through the three priority queue data structures, searching for the first task known to reduce memory usage. Once such a task is found, the task's entry in the priority queue is removed, and the task is re-enqueued with maximum priority. Then the scheduler resumes its normal operations, resulting in that task being executed next.

Of course, the scheduler needs to know which tasks are candidates for rescheduling. The adaptive scheduler therefore contains a list of such tasks. The list is populated with tasks specified by the application programmer in an interface file. All Charm++ programs contain one or more simple interface files that specify the entry methods and other parallel constructs in the program. A simple translator parses the interface file and generates C++ code that is compiled into the program to support the specified entry methods and other constructs. A new tag called `[memcritical]` has been added to the interface file's grammar and parser. When this new tag is added as an annotation to any entry method, the entry method will be included in the scheduler's list.

3 LU Case study

To evaluate the usefulness of the memory adaptive scheduler described in section 2.2, an LU program was modified to enable the adaptive scheduler. This section describes the LU implementation as well as its performance characteristics both with and without the adaptive scheduler. The resulting memory consumption patterns for the program are analyzed to show that the memory-aware scheduling technique does indeed reduce memory usage in a useful manner. The section concludes with a set of insights gained from this case study.

3.1 Experimental Setup

All runs of the Charm++ LU implementation are performed on 64 nodes of an IBM Bluegene/P system at the Argonne Leadership Computing Facility. Only one core per node is used, because the memory utilization patterns are the focus of this paper. Each node contains four processor cores running at 850MHz and 2GB of memory. All measurements of GFlop/s are counted in a manner consistent with the standard HPL benchmark. All visualizations of processor timelines are generated from actual application traces analyzed using the Projections performance analysis toolkit [9].

For the performance critical numerical kernels, when using an IBM Bluegene/P system, the Charm++ LU program uses the `dgemm` and `dtrsm` routines from the Engineering and Scientific Subroutine Library (ESSL).

For performance comparisons, the well-known High Performance Linpack Benchmark (HPL) version 2.0 was run on the same system with identical block sizes and matrix sizes.

3.2 Charm++ LU Implementation

To write a dense LU algorithm, there are many implementation choices to be made. This section describes some of the design decisions made when developing a Charm++ implementation of dense square LU matrix factorization. The LU program was written as simply as possible, without any explicit memory-awareness in the parallel program's code. This implementation does not perform pivoting. Hence some numerical stability is lost, but the same number of floating point operations are still performed when compared to an LU program that implements pivoting [4].

The program uses a 2-D *chare array* to decompose the 2-D matrix into $b \times b$ square blocks. Each matrix block is stored in one of the chare array elements. The mapping of the chare array elements to processors is flexible. The default

Charm++ mapping is a block mapping, but the program can easily specify other mappings, and for this LU program two custom mappings were developed. Section 3.3 describes the advantages and tradeoffs for these mappings.

The main communication pattern that occurs in an LU matrix factorization is a multicast of a data block from a source block to all subsequent blocks in the same row, and a downward multicast of a data block from its source to all blocks below it in the same column. The Charm++ language natively supports chare array section sends, which are a mechanism for sending a single message to a set of destination chare array elements. The programmer can choose one of many predefined algorithms for each section send [10]. The Charm++ LU implementation can therefore easily represent the pattern of communication that needs to occur. The multicast algorithm that appears to perform well for the cases described below uses a simple processor spanning tree of degree 4.

The main computations performed in a dense LU algorithm are matrix-matrix multiplications that update the values in a block. This update operation is referred to as a trailing update. For block (i, j), the block LU algorithm performs $min\,(i, j)$ trailing updates. The closer a block is to the bottom right corner of the overall matrix, the more computation is performed for it. Other computationally intensive portions of the algorithm involve local single-block LU factorizations to be performed for blocks along the diagonal, and updates along the topmost active row and leftmost active column.

To factorize an $n \times n$ matrix, approximately $\frac{2n^3}{3}$ floating point operations are required. Assuming the matrix is decomposed into $b \times b$ square blocks, the fraction of the floating point operations spent inside the matrix-matrix multiply operation approaches $1 - \frac{1}{b^2}$ as b increases [4]. Thus for large LU factorizations, almost all floating point operations occur in the context of matrix multiplication. Therefore, a performance of a good LU implementation should approach the performance achieved by the double precision matrix-matrix multiply.

3.3 Mapping Blocks to Processors

There are two mapping schemes implemented in the LU program. The mapping schemes define the processor that creates and perform operations on each chare array element and its corresponding matrix block. The first is a traditional block-cyclic mapping. A second mapping is proposed in this paper, as it achieves better performance than the block-cyclic mapping scheme for certain problem sizes and numbers of processors. The mapping schemes are static, so the blocks do not migrate between processors. All work associated with a block will be performed on the processor owning the block.

Block-Cyclic Mapping The block-cyclic mapping scheme is the traditional method used by many parallel LU implementations [7]. The advantages of a block-cyclic mapping are its simplicity and its relatively low communication volume. Each row or column of blocks spans only $\sqrt{p}$ of the p processors. Thus all

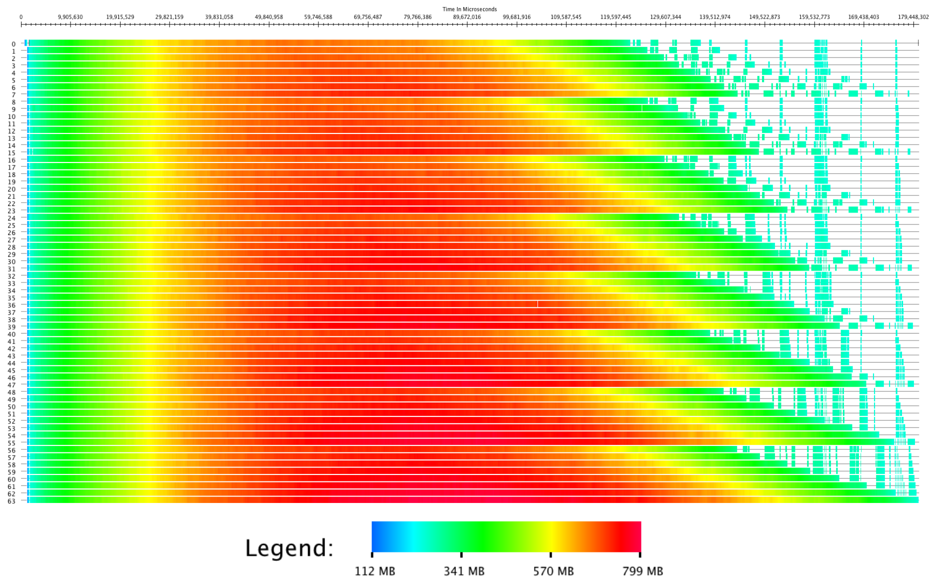

Fig. 1. A timeline view, colored by memory usage, of an LU program run on 64 processors using a traditional Block-Cyclic Mapping for a $N = 32768$ sized matrix with 512×512 sized blocks. The traditional block-cyclic mapping suffers from limited concurrency at the end (the right portion of this plot).

of the multicasts have at most $\sqrt{p}$ destination processors. However, the disadvantage is that the work is unevenly balanced near the end of the computation. Figure 1 visualizes the entire computation for a run of the LU program on 64 processors. In an attempt to fix the imbalance near the end of the computation, a second mapping scheme was developed.

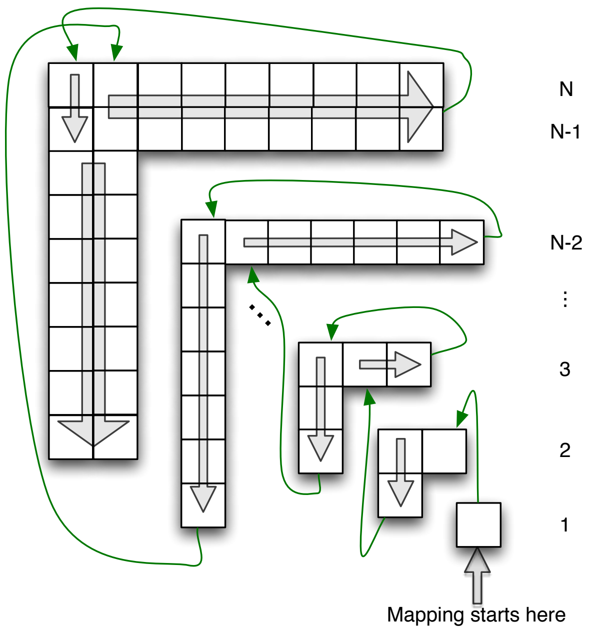

Fig. 2. The traversal order for the *balanced snake mapping*.

Balanced Snake Mapping In order to balance the amount of work that is performed on each processor, a new mapping scheme was developed called a *balanced snake mapping*. Figure 2 helps illustrate the order in which blocks are mapped in this scheme. The blocks are traversed in the order shown by the arrows. This traversal order visits the blocks in roughly decreasing order of the amount of work expected to be performed by each block. As each block is visited, it is assigned to the processor which has been assigned the smallest amount of work so far. Thus the first p heaviest blocks will be assigned in a round robin manner to the processors, and the remaining blocks will be assigned in a manner that attempts to balance the load across the processors. The assignment function also forces subsequent blocks in traversal order to be on different processors.

It is expected that the number of processors spanning each row of blocks is larger than $\sqrt{p}$. In the case of 64 processors, with a matrix partitioned into

64×64 blocks, there are on average 43 unique processors spanning each column of the matrix and 49 unique processors spanning each row of the matrix. So in this case, the average number of unique processors on each row and column is much higher than $\sqrt{p} = \sqrt{64} = 8$. Thus the multicast of a block along a row or column will involve more processors than the traditional block-cyclic scheme, and the multicasts will therefore incur a higher overhead. For large numbers of processors, the block-cyclic mapping performs better than this newly proposed balanced snake mapping.

Comparison of the Two Mapping Schemes Although the balanced snake mapping does a much better job of evenly distributing the workload, the increased overhead for communication results in small delays between many of the matrix-matrix multiplications when compared to the block-cyclic mapping. Figure 1 shows that there the block-cyclic mapping exhibits a load imbalance near the end of the computation, while the balanced snake mapping for the same problem exhibits a much better load balanced, as seen in figure 3. When the $N = 32768$ problem with 512×512 block sizes is run on 64 processor cores the balanced snake mapping performs better, achieving 138 GFlop/s, whereas the block-cyclic mapping yields 131 GFlop/s. A theoretical analysis of the computation and communication properties of the block-cyclic mapping and some other matrix decomposition schemes are provided elsewhere [5].

Automatically Determining The Optimal Mapping Scheme Although it is clear that the block cyclic scheme has benefits for large numbers of processors, and the balanced snake mapping exhibits a better load balance for small matrix sizes, the decision of which scheme to use for a specific problem size and machine depends upon the performance characteristics of the machine as well as the problem size. Thus it is advantageous for the choice to be made automatically. This section describes one such method for choosing between the mapping schemes at runtime. It is possible to automate the choice between the two mapping schemes. The automatic decision can utilize the fact that the block-cyclic mapping scheme produces larger amounts of idle time for some of the processors toward the end of the factorization.

To automatically determine which scheme to use, the LU program is adapted to use the measurement based steering framework provided by the Charm++ runtime system [10]. The program provides to the steering framework a tunable knob and some information about the effects of the knob. In this case the program specifies that one mapping scheme can possibly reduce the amount of idle time wasted by the processors. The steering framework will therefore be able to turn the application provided knob when a large amount of idle time is detected. Figure 4 shows a performance visualization of an execution of the program performing 10 consecutive LU factorizations. The initial LU factorizations result in a large amount of idle time because the matrix blocks are not well distributed across the processors. Hence after a period of observation for the first 3 factorizations, the steering framework decides to turn the appli-

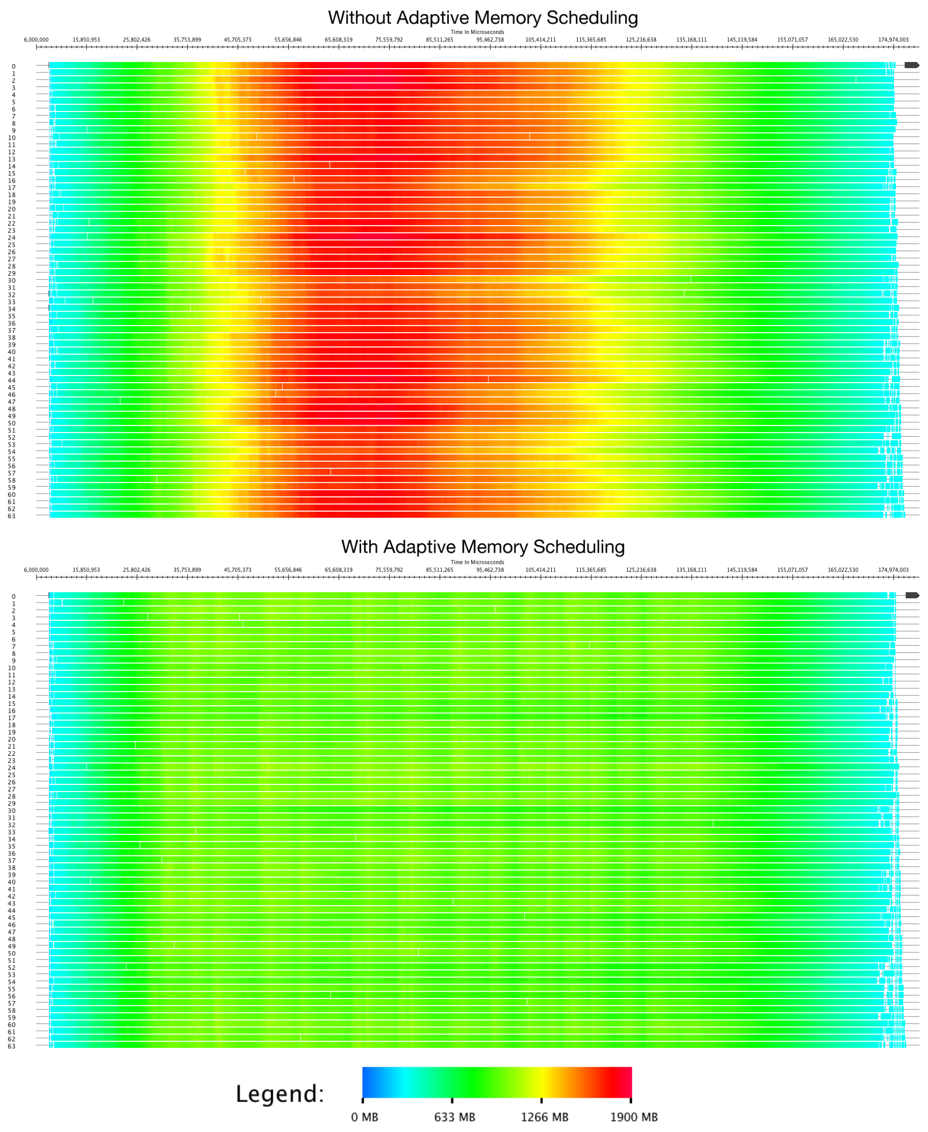

Fig. 3. Plot of memory usage on each processor over time, both without and with adaptive scheduling using a 1000MB threshold.

cation provided knob that switches from the block-cyclic mapping scheme to the balanced-snake scheme. The subsequent LU factorizations complete more quickly, although a larger amount of overhead time is present. The increased overhead time is caused by the increased multicast communication volume inherent in the mapping scheme.

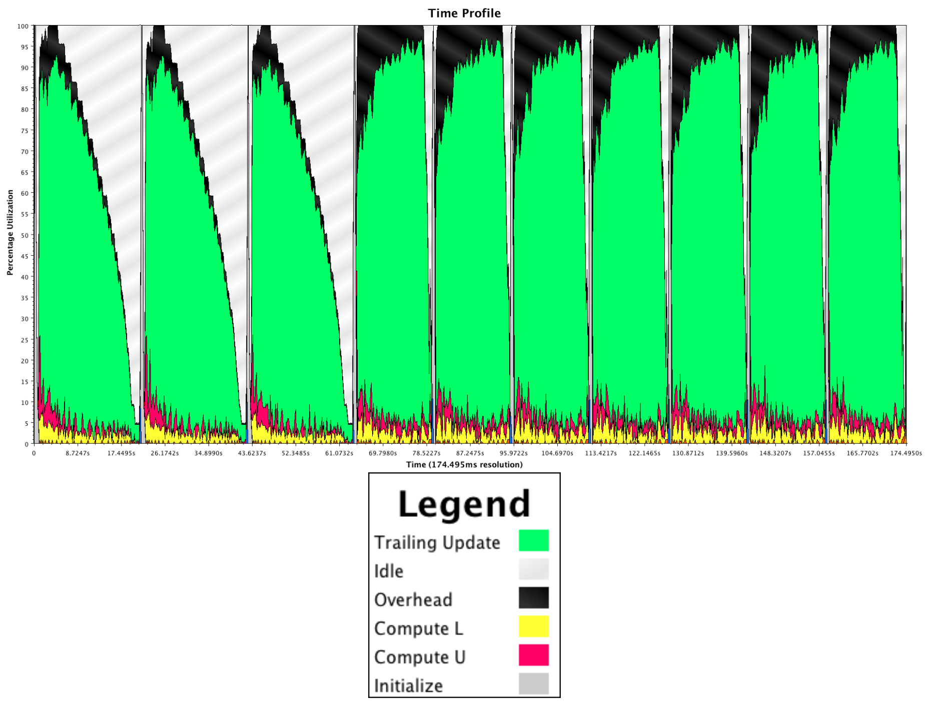

Fig. 4. Visualization of a program performing 10 LU factorizations. After the third LU factorization, the measurement based automatic steering framework instructs the program to use the snake-mapping instead of the block-cyclic mapping. This adaptation reduces the amount of idle time found in the subsequent 7 factorizations.

3.4 Priority Based Dynamic Lookahead

One general goal when writing parallel programs is to expose as much concurrency as possible to provide for the greatest opportunities to fully exploit the available processors and obtain high application performance. In common parallel LU algorithms, there are important tasks along the critical path of the computation, namely the block LU factorizations and the following topmost active row and leftmost active column block updates. Scheduling such tasks as early as possible results in greater exposed concurrency earlier in the program. The other tasks, namely block trailing updates, can sometimes be delayed relative to the other tasks. If the trailing updates are executed with high priority, the program will not expose enough concurrency to keep all processors busy because the other critical path tasks are delayed in time. Alternatively, if the trailing updates are executed with low priority, then the critical path tasks will execute sooner, causing an avalanche of enqueued block trailing updates across all processors. The enqueued block trailing updates necessitate the buffering of two incoming data blocks. These blocks will occupy space in memory, and an increase in delayed trailing updates will directly relate to increase in memory usage.

When writing an LU program, there are a few options regarding how much lookahead to support. High degrees of lookahead cause more trailing updates to be delayed, increasing memory usage. Low degrees of lookahead ensure that trailing updates cannot be buffered for too long, and hence the memory usage will not be as high.

The simplest LU implementations ignore the issue of lookahead and allow the program to proceed without regard to how far ahead one processor can compute relative to tasks buffered on itself or other processors [3]. Such an unlimited lookahead scheme is not scalable because memory usage can grow as the problem size is scaled up. At some point the program cannot run because memory is exhausted and the program will deadlock. Other algorithms, such as the one used in the High Performance Linpack implementation include a static parameter specifying the allowed degree of lookahead [2]. Other implementations support dynamic lookahead, but restrict some tasks so that deadlock will not occur when memory is exhausted [6].

Dynamic lookahead is important because better performance can be achieved due to the greater amount of available concurrency than is found in a static lookahead algorithm. Hence dynamic lookahead is typically preferred over static lookahead [6]. However, existing dynamic lookahead schemes require applications to include specific code that explicitly coordinates between sending and receiving processors to ensure memory is not exhausted.

The Charm++ LU implementation described in this paper is written to provide unlimited lookahead, with no code attempting to reduce concurrency. Priorities are assigned to tasks with higher priorities for block LU operations occurring in the upper-leftmost active blocks and lower priorities for the trailing updates, with priorities decreasing for each type of event from top left to bottom right. The priority scheme should provide as much concurrency as is available at any point in time.

This section shows that although the LU program itself is written with unlimited lookahead and hence a high level of available concurrency, a general purpose memory-aware scheduling technique provides a sufficient mechanism to reduce the memory consumption of the simple LU program. This scheduling technique will dynamically vary the lookahead in the case of LU, but could also be used to control the memory usage patterns of other Charm++ programs.

3.5 Enabling Memory-Aware Scheduling

To enable the new memory-aware Charm++ scheduler, the user is only required to modify the Charm++ interface file (.ci file) for the program by adding one annotation to each entry method that could be used for reducing memory usage. The reason this current implementation uses annotations is that the user has knowledge of the program behavior, particularly which entry methods will decrease memory usage. In the LU implementation, the trailing update entry method is the sole method that is annotated for possible rescheduling when the memory threshold is reached.

Figure 3 visualizes the memory usage over time for two runs of the LU program. The first run does not use the memory adaptive scheduler, while the second one does. The figure demonstrates that the memory utilization is reduced once the memory adaptive scheduling is enabled for the trailing updates. Because for this case, the memory usage approaches the physical memory on a BG/P node, this is the largest problem size, N=32K, that can be run on 64 nodes of BG/P with the traditional Charm++ scheduler. The maximum memory usage is 1873MB without memory-aware scheduling, and 1122MB with memory adaptive scheduling.

3.6 Analysis of Resulting LU memory patterns

To analyze the effects of the memory-aware scheduler, the LU program was run with varying thresholds. Figure 5 displays the memory utilization over time with varying thresholds, for an $N = 32768$ sized matrix with 512×512 sized blocks. The horizontal red line displays the corresponding memory threshold for each run. This figure shows that adapting the scheduler queue does constrain the memory that is used on each processor. It appears that 300MB was the minimum effective threshold for this problem size, which is evidenced in figure 5 where the actual memory usage for all the processors is mostly above memory threshold. In the runs where the threshold is higher (600MB and 1200MB), the range of memory footprints for all processors mostly straddles the threshold. In all three cases where a threshold is applied, the memory usage is reduced from the original version where no adaptation was performed in the scheduler.

3.7 Analysis of Performance

When running with the N=32768 matrix problem size and 512×512 block size, the Charm++ LU implementation using the balanced snake mapping performs at 138 GFlop/s. The same implementation using a block-cyclic mapping performs at 131 GFlop/s. Both of these configurations perform better than the standard HPL benchmark [2]. Figure 8 shows the resulting performance of 93 different configurations for the HPL benchmark. All of these configurations use the same N=32768 matrix problem size and 512×512 block size, but the other

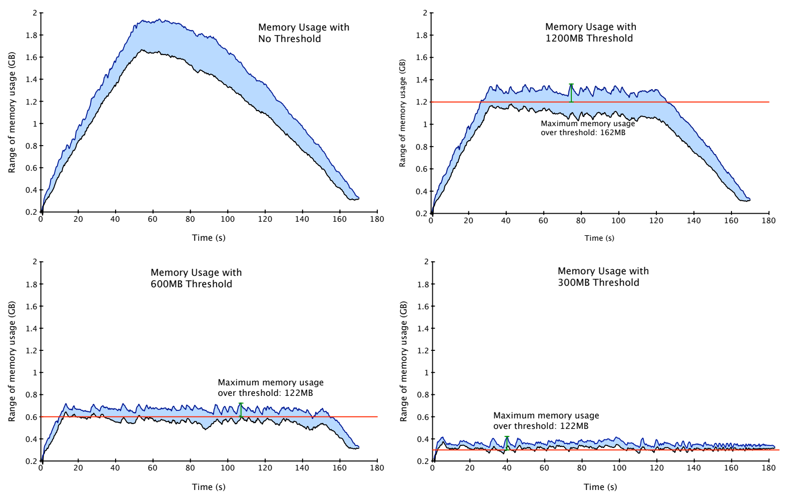

Fig. 5. Ranges of maximal memory utilization across all processors over time for different thresholds. The adapting scheduler causes the memory usage to remain close to the threshold for this LU factorization.

configurable parameters are varied. The broadcast method, processor grid arrangement, depth of lookahead, panels in recursion, and recursive stopping criterion were all varied. The maximal observed performance for HPL among these 93 different configurations is only 111 GFlop/s.

3.8 Costs of Modifying the Scheduler Queue

The overhead of adapting the scheduler queue for the LU factorization program is small. To measure the overhead, timer calls were added around the code that adapts the scheduler queue. Included in this code is the function that determines the current memory usage and compares it to a threshold. When the LU program is run with an $N = 32768$ sized matrix and a 512×512 block size, the average time spent in the scheduler modification code on each of the 64 processors was 0.0239 seconds while the whole LU factorization takes 168.4 seconds. This corresponds to a negligible overhead of 0.014%.

3.9 Insights Gained From This LU Implementation

The naïvely written LU program exhibits a simple memory usage pattern: memory usage changes slowly, and is relatively uniform across processors at each point in time. The memory usage generally grows to a single maximum value on each processor and then shrinks back down to the minimum required to store the matrix. The memory patterns are different however when a memory-aware adaptive scheduler is used, or when lookahead is restricted by other means. Hence,

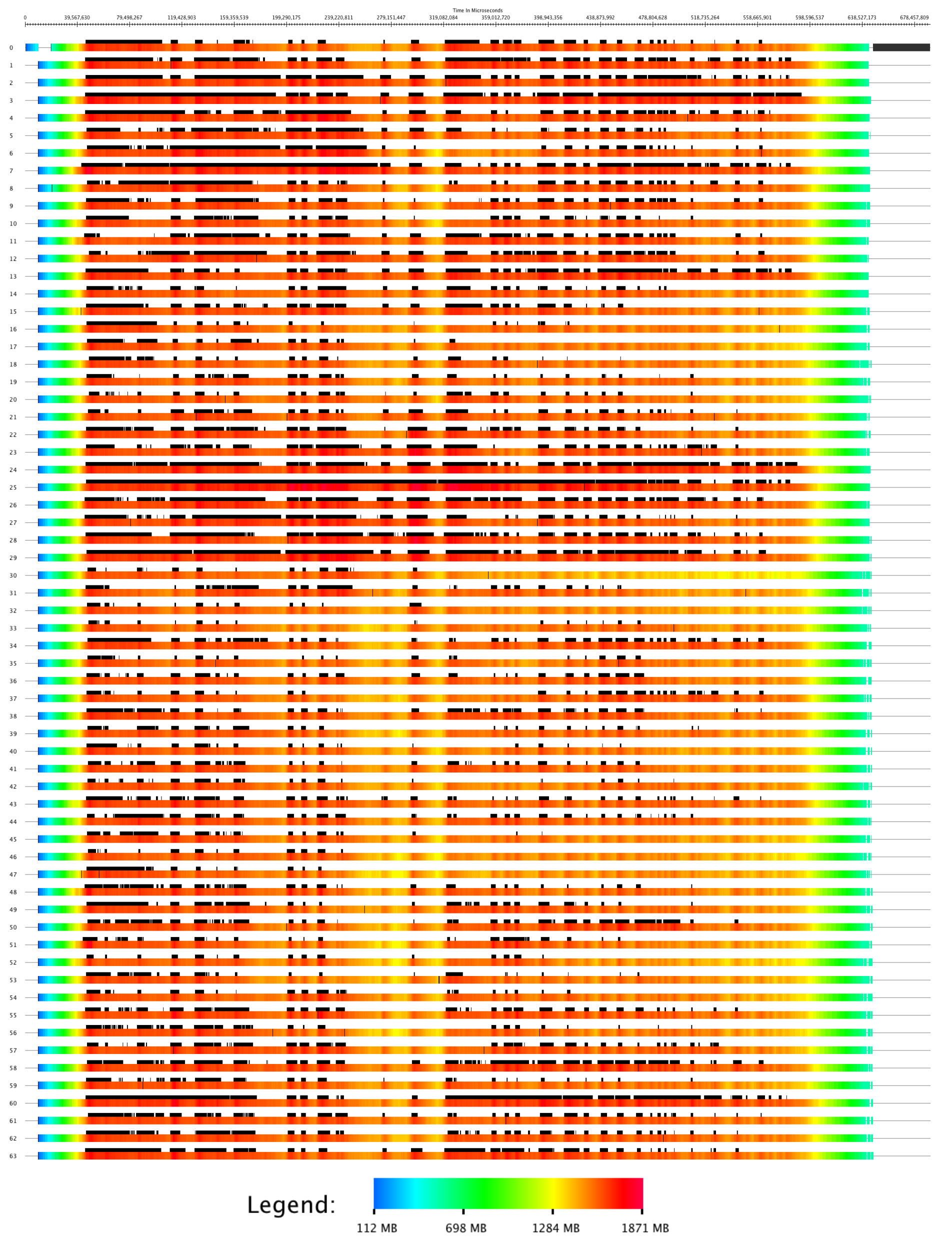

Fig. 6. A timeline view of an execution of LU on 64 processors for a larger matrix N=51,200 using the adaptive scheduler. This same program dies when it runs out of memory when not using the adaptive scheduler. Each row in the figure corresponds to one of the processors, with colors indicating memory usage. Black tick marks on the top of each row indicate a point where a trailing update is immediately executed because the memory usage is over the specified threshold.

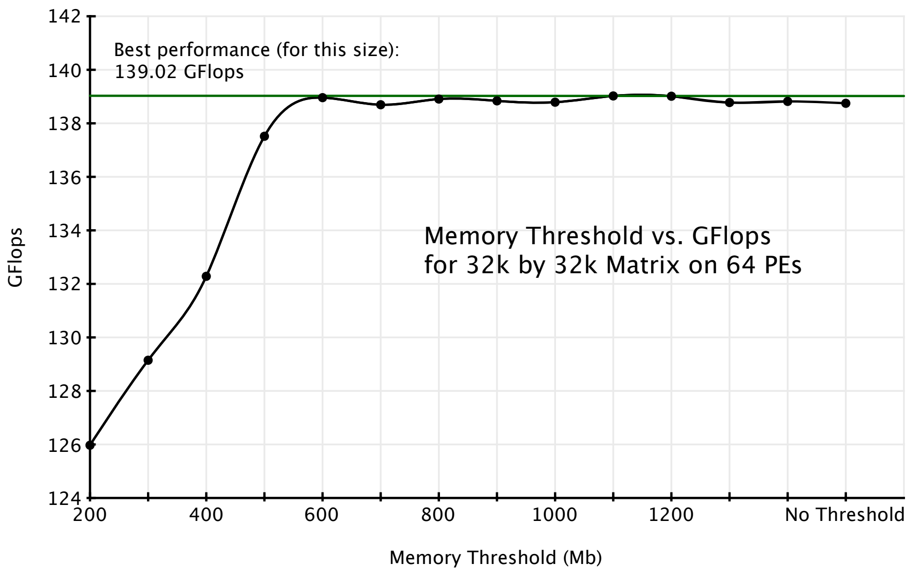

Fig. 7. Performance of LU program for various memory thresholds. The problem is a factorization of an $N = 32768$ sized matrix with 512×512 sized blocks run on 64 processor cores of BG/P.

using memory adaptive scheduling on each processor can constrain the memory usage in a useful manner.

The performance of the LU program over a range of memory thresholds shows two performance regimes. The first exhibits decreasing performance when lower thresholds are used, while the second regime is a large plateau of sufficiently large thresholds. Figure 7 shows that these two performance regimes meet at some point, namely the knee in the plotted curve knee in the curve.

A simple straightforward implementation of LU in the Charm++ language can achieve reasonable performance, while remaining flexible and not requiring complicated application-specific schedulers or static limitations on lookahead. Charm++ makes it easy to specify the mapping of blocks to processors and to specify the priorities of each task. When developing the LU program, we found that a non-standard mapping outperformed the traditional block cyclic mapping.

Finally, the new adaptive scheduling technique enables larger LU factorizations to be performed, even ones that previously would have failed by depleting all available memory. Figure 6 shows a timeline visualization of one such larger factorization of an $N = 51200$ matrix size.

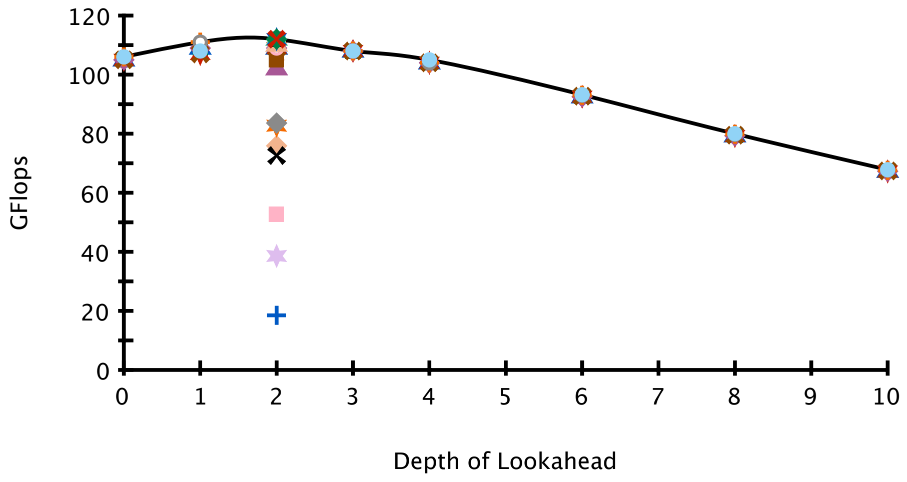

Fig. 8. HPL performance on 64 processors for 93 different configurations for a $N = 32768$ sized matrix with a 512×512 block size. The configurations were tested in two phases. The first phase varied some parameters to find a good lookahead value. Then the best lookahead depth of 2 was fixed and more configurations were evaluated. The best observed HPL performance is 111 GFlop/s, which is .

4 Automatically Finding An Optimal Memory Threshold

Although the scheduling scheme presented earlier in this paper can reduce memory consumption for a certain class of programs, the memory aware scheduling scheme does not provide hard upper limits on the amount of memory used by a program. Thus a reasonable threshold needs to be chosen for a run of the program. The simplest scheme would be to set the threshold to a fixed fraction of the system's memory. A safer, and better solution is to automatically find the threshold that yields the best performance. This section describes an automatic scheme that slowly increases the threshold while observing memory consumption measurements across all processors.

The proposed scheme is simple. The memory threshold is initially set to a safe low value, but it is automatically increased when previously observed memory usage measurements are low enough. After the threshold has been increased to a level where further increases are likely to exceed the desirable limits, the tuning framework [10] scans through its recorded history to find the best known configuration. The best known configuration can then be used for all future factorizations. This automatic tuning system can find a configuration providing good performance while restraining the actual memory consumption even when it exceeds the specified threshold. Figure 9 displays the actual memory usage over successive LU factorizations for a program using the automatic threshold determination scheme described in this section.

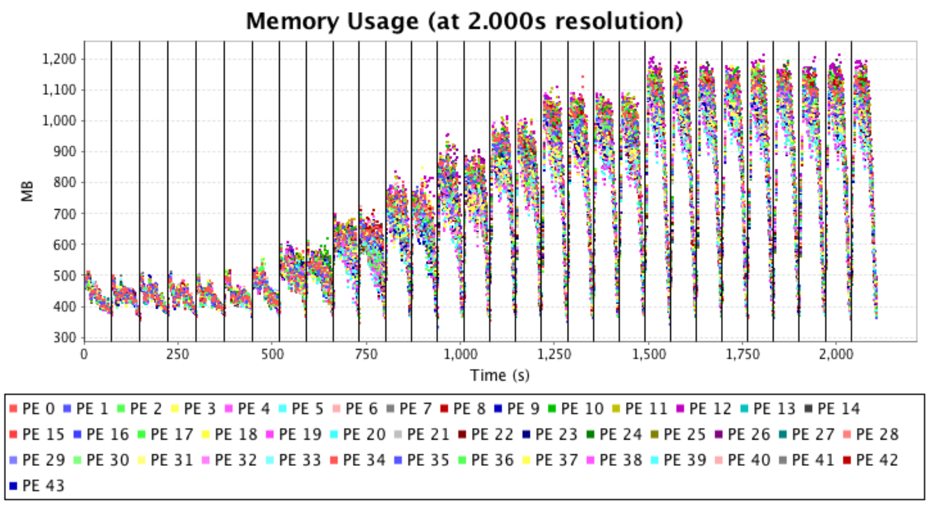

Fig. 9. Actual memory usage for each of 44 processors while the LU program performs 30 successive factorizations. The memory threshold is increased by an automatic tuning mechanism whenever memory usage measurements from previous factorizations are still low.

5 Related Work

One implementation of LU uses virtualized descriptors for the blocks allowing an arbitrary mapping to processors, specifically to enable work stealing and adaptation to newly available processors in a grid context [3]. That implementation achieves 21% of peak on 128 Xeon processors connected via Gigabit Ethernet with matrix width of $N = 46,080$. That implementation is similar in many ways to the asynchronous implementation in this paper. It does not perform pivoting, and it uses asynchronous unlimited lookahead and virtualized locations of the blocks. It however only uses a ring multicast strategy. Their implementation suffers from the memory constraints described in this paper, and it is specifically noted that the implementation cannot run the N=46,080 problem for the infinite lookahead priority configuration.

Memory-aware scheduling has also been used in other contexts. The constructive algorithm [12] considers data and memory constraints in addition to other constraints for scheduling tasks in real-time systems. The scheme described in [11] collects memory performance related data online and then feeds those to an analytical model of cache and memory behavior, trying to minimize the overall miss-rate in scheduling a collection of processes time-sharing or space-sharing a cache. In contrast, our memory-aware scheduling limits the amount of memory usage for a running parallel application on clusters, which indirectly controls the amount of concurrency (the lookahead depth in the LU case) during the execution.

The hybrid scheduling [1] scheme, applied in the parallel solution of linear systems, first estimates the memory usage besides the workload on a static optimistic scenario during the analysis phase, and then such estimations are used during the factorization phase to constrain the dynamic decisions to better balance the workload. In comparison, our scheme does not perform any static analysis but instead depends on the dynamic memory footprint for adaptive decisions and trades off the amount of concurrency for reducing the maximum memory usage.

It may appear at first glance that the LU implementation described in this paper would benefit from dynamic load balancing, however, there is no dynamic load imbalance within a single LU factorization, if all processors are identical. Hence techniques such as work-stealing are irrelevant for dense LU factorization implementations that run on many homogeneous processors as is the norm in large scale parallel machines. Other potential uses for work stealing techniques would include stealing tasks to balance the memory consumption across processors. Again, such techniques would be irrelevant because the memory usage across processors is nearly uniform, and hence there is no obvious benefit to work-stealing for that purpose.

6 Conclusion

In this paper, we introduced a new method for constraining memory usage dynamically over the lifetime of an application. We showed that this method can be utilized by a programmer who simply annotates methods that reduce memory usage. Furthermore, the utility of this new scheduling mechanism was demonstrated by showing that an LU factorization algorithm can be scaled beyond the $N = 32768$ problem size, without any other modifications to the program. Typically, there is a tradeoff between implementing dynamic lookahead, which introduces many problems and increases the complexity of the program significantly, and using static lookahead, which constrains the concurrency. We showed that the best of these extremes can be realized in Charm++ using a simple LU factorization program, which implicitly allows for infinite lookahead but is constrained by our memory-aware scheduler so it can scale to large problem sizes.

In the future, we may explore the possibility of automatically selecting tasks to be rescheduled to eliminate the use of annotations by the programmer.

References

1. P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Comput.*, 32(2):136–156, 2006.

2. J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, Jan 2003.
3. T. Endo, K. Kaneda, K. Taura, and A. Yonezawa. High performance LU factorization for non-dedicated clusters. In *Proceedings of the 4th International Symposium on Cluster Computing and the Grid (CCGrid 04)*, pages 678–685, 2004.
4. G. H. Golub and C. F. Van Loan. *Matrix Computations (Johns Hopkins Studies in Mathematical Sciences)*. The Johns Hopkins University Press, October 1996.
5. B. Hendrickson and E. Womble. The torus–wrap mapping for dense matrix calculations on massively parallel computers. *Siam J Sci Stat Comp*, Jan 1994.
6. P. Husbands and K. Yelick. Multi-threading and one-sided communication in parallel LU factorization. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–10, New York, NY, USA, 2007. ACM.
7. P. Husbands and K. Yelick. Multi-threading and one-sided communication in parallel lu factorization. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–10, New York, NY, USA, 2007. ACM.
8. L. V. Kale, E. Bohm, C. L. Mendes, T. Wilmarth, and G. Zheng. Programming Petascale Applications with Charm++ and AMPI. In D. Bader, editor, *Petascale Computing: Algorithms and Applications*, pages 421–441. Chapman & Hall / CRC Press, 2008.
9. L. V. Kale, G. Zheng, C. W. Lee, and S. Kumar. Scaling applications to massively parallel machines using projections performance analysis tool. In *Future Generation Computer Systems Special Issue on: Large-Scale System Performance Modeling and Analysis*, volume 22, pages 347–358, February 2006.
10. Parallel Programming Laboratory, Department of Computer Science, University of Illinois, Urbana, IL. *The Charm++ Programming Language Manual, (Version 6.1.3)*, 2010.
11. G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 117, Washington, DC, USA, 2002. IEEE Computer Society.
12. R. Szymanek and K. Kuchcinski. A constructive algorithm for memory-aware task assignment and scheduling. In *CODES '01: Proceedings of the ninth international symposium on Hardware/software codesign*, pages 147–152, New York, NY, USA, 2001. ACM.