Memory Hierarchy Aware Parallel Priority Based Data Structures

Dinesh Agarwal (3rd year) Ph.D. student
Adviser: Sushil K. Prasad
Georgia State University
Department of Computer Science
Atlanta, Georgia 30303

Abstract—With the proliferation of multicore and manycore architectures, memory hierarchy plays an important role in realizing expected performance of memory intensive scientific applications. A vast majority of scientific applications require a priority based data structure to discriminate among available data elements; for instance, a priority based data structure is imperative for extracting the earliest events in a discrete event simulation, identifying urgent tasks in a parallel scheduler, or exploring most promising sub-problems in a state-space search. Traditional priority based data structures are tree-based that makes them cache unfriendly due to the exponentially increasing distance between parent and child nodes. Fine grained large-scale applications further cause excessive contention among competing processors due to frequent updates. In this dissertation we propose a priority based data structure that adapts to the memory hierarchy of the underlying computer system. The top priority items are aggregated and kept into a working subset of data items that is readily accessible for processing.

I. Problem Description

With the proliferation of multicore and manycore architectures, memory hierarchy plays an important role in realizing expected performance of memory intensive scientific applications. Intel hosted a workshop a few months ago to brainstorm how to tackle the irregular, non-numeric, discrete computations on the emerging multicore architectures. The premise was that the parallel processing community excels in solving regular, numeric, scientific computations. However, there is equally large if not larger class of ill-structured problems, such as those related to graphs and networks, state-space search, tree-based data structures, unordered multisets and other scattered or unstructured data, which are poorly understood. What is fundamentally so difficult about these irregular discrete problems? Consider an example of discrete event simulation of a large complex network (communication network, logic circuit, immune system, or social network). Although the network may have hundreds of thousands or even millions of nodes (or agents), activity may only be in a few nodes, say 1% of them, at any given step. What is worse, this set of active nodes can be arbitrarily scattered across the network. Likewise, in a state space search of an exponential space, say for a SAT solver or a game of Chess, the set of high-potential nodes (possibly closer to the goals) in any given step is a tiny set, again scattered dynamically in space as the computation unfolds.

A fair amount of work on design, implementation, and application of some parallel data structures (like stack, queues, deque, linked list, heaps) on traditional shared-memory machines has been reported in the literature in the past 25 years [1]–[6]. To the best of our knowledge, no significant result has come out on priority based data structures for multicore (or manycore) architecture because of the inherent difficulties. Priority based data structures have been implemented in concurrent setting in different ways, including concurrent heaps, calendar queues, array-based, tree-based, hash-table-based priority queues; but these are suited for the traditional shared-memory multiprocessors [1], [7] or for distributed architectures [3], [4]. The recent data structure related work on multicore platforms such as on formalization of language primitives to manipulate hierarchical place trees [8] highlight the broad importance of this problem; however these have focused primarily on language issues and not on performance. In this dissertation we are going to study the behavior of such irregular, non-numeric, discrete computations in order to come up with a data structure that adapts to the memory hierarchy of underlying system to reduce the effective memory latency while increasing the effective memory bandwidth at the same time.

II. Related Work

Priority based data structures are not a new area of research. Many variations of priority queues were proposed during 90s to cater to the needs of almost all scientific applications that require priority based data structures. These data structures can be categorized by the following two approaches reported in literature to parallelize them. The first one is to speed up the individual queue operations known from the sequential setting using a small number of processors [5], [9]–[12]. The parallel priority queue of [9] for instance, reports a time and work optimal priority queue for the CREW PRAM which supports FindMin in constant time with one processor and MakeQueue, Insert, Meld, FindMin, ExtractMin, Delete and DecreaseKey in constant time with $O(\log n)$ processors. The other approach is to support insertion and deletion of $k$ items where $k$ is a constant [2], [11], [13], [14] reported two new data structures, the $n$-Bandwidth-Heap and $n$-Bandwidth-Leftist-Heap which support insertion of $n$ new items in a priority queue of $m$ elements in parallel time $O(h + \log n)$ and deletion of the $n$
smallest items in time $O(h + \log \log n)$, where $h = \log \frac{m}{n}$ on a CREW-PRAM. The data structures represented by them are modified and augmented by [11] to give an EREW-PRAM PPQ which support insertion and deletion in $O(\log h + \log n)$
time. [15] presented constant time operations on a priority queue. Their main result is that any sequence of $n$ queue operations involving $m$ elements in total can be performed in $O(n)$ time using $O(m \log n)$ operations on the EREW PRAM. [16] surveys concurrent priority queues to analyze various methods of synchronization such as locking, blocking and a software transactional memory based approach. Implementation of parallel priority queue on practical models was reported by Das et al. [13] and [14]. The authors present a scheme to map a priority queue on the hypercube architecture in a load-balanced manner with no additional communication overhead. [17], [18] gave an efficient implementation of Parallel Heap on bus-based shared memory machines such as sequent balance and SGI power and later on SGI Origin systems, which are NUMA, shared machines. With the advent of multicore and manycore architectures in CPU design it has become essential to revisit these data structures.

Research has also reported effective techniques to make use of multicore computers and memory hierarchy for memory intensive scientific applications such as sparse matrix vector multiplication [19], [20], Matrix Reduction [21], [22], and divide and conquer algorithms [23] among others. Nevertheless, these problems are regular in nature thereby exploiting data level blocking and/or thread level blocking are sufficient to exploit memory hierarchy. To the best of our knowledge, no prior work has concentrated on a solution for irregular, non-numeric, discrete computations.

III. SIGNIFICANCE OF THE RESEARCH

Over the past several years multicore technology has become mainstream in CPU design, with potential to transform each laptop and PC into a true high-performance platform for the masses. Research in the area of multicore architecture has addressed the issue of optimizing computationally intensive, regular, numerical applications. Work has also been conducted on multicores to improve performance by concentrating on memory architecture of multicore chips for individual applications. One distinguishing feature of the non-numeric problem domain is their use of non-array data structures to achieve algorithmic optimality. Consider, for example, stacks and queues in graph traversals or priority queues in Huffman’s prefix code construction. Probably the most widely used complex data structures in non-numeric computation are the ones which are priority based, such as priority queue. Additionally, coarse grained applications are easier to parallelize by employing the methodologies similar to those used for regular numeric applications. However, for fine-to-medium grained applications these methodologies are ineffective and can even cause memory contention among competing processors. While there are hosts of legacy applications such as discrete event simulation, branch and bound, and other non-numeric applications that require priority based data structures, there are no known general frameworks that can efficiently accelerate these large classes of applications. In this dissertation we intend to address this gap for medium-to-fine-grained priority-based applications.

IV. GOALS AND APPROACHES

In the modern multicore architectures each processor has multiple cores on the same chip. Additionally, each core has its small fast private cache as well as there is a larger cache shared among all of the cores on a chip. The latest chips such as Intel Nehalem and AMD Phenom have 3 levels of cache hierarchy. As correctly predicted by Yale Patt [24] in 2001, modern CPUs were supposed to have three or more levels of caches which means we can safely expect chip manufactures to add one or more levels of caches in the future architectures. However, unlike some manycore architectures such as GPUs multicores do not provide researchers with explicit software control over data allocation to these different levels of available cache memories. An application that performs ideally on a multicore simulator may not show even half as good performance on a real multicore chip due to factors that cannot be controlled by the application. For instance, in order to control the allocation of data to a specific level of cache hierarchy, the only way is to exploit the memory replacement policy. For instance, on a computer system that uses LRU policy the thread affinity to a processor can be controlled in such a manner that cpu stalls due to memory latency can be reduced drastically. Nevertheless, it is challenging because the cache coherency protocols [25] vary among different manufacturers.

Therefore, the goals of our research are

1) To study existing priority based data structures to assess their capabilities to leverage memory hierarchy on multicore processors,
2) To create a framework to control thread and memory affinity,
3) To employ prefetching to help reduce memory latency, and
4) To implement memory hierarchy aware, parallel, priority-based data structures

To achieve these goals we have explored following approaches that can be applied to a priority based data structure to leverage multiple levels of memory hierarchy.

A. Task Allocation for Cache Locality

Tree based data structures are difficult to optimize for cache hierarchy based architectures since the parent-child nodes are kept far from each other. In order to access both parent and children nodes enormous amount of cache misses are required. The cache misses increase with the distance of the current nodes from root as the size of the tree grows exponentially going down from root. For instance, In case of PPQ to update the $k$ items at any node, without destroying the heap property, it is essential to access the $2k$ items at its children. This scenario can be worsen if the processor/core operating at the current node does not have the nodes stored in its cache. In this case it needs to read the information either from the
shared cache, another core’s private cache, or main memory. Moreover, once the current node is updated it might have destroyed the heap property at one of the children nodes and thus the update process should repeat at that child node in the next step and so on. In order to alleviate this problem we tried to allocate tasks to the cores in such a manner that the core that operates at a node in step $i$ is the one that operated at that node in step $i - 1$. Therefore, the current node was most probably left in the last level cache during step $i - 1$ and does not cause cache misses during step $i$. We call this scheme $RR_{1st}/1$ because the initial allocation of tasks (levels of a tree) to cores is based on round robin fashion and after first step the cores move down one level in a cyclic shift.

B. Data Dependent Prefetching

Data Prefetching is an effective technique to hide data access latency by predicting future data access patterns of a process. Prefetching helps to mask CPU stalls caused by cache misses and to bridge the performance gap between processor and memory. Prefetching essentially requires a careful observation of what to prefetch and when to prefetch [26]. Given the limited size of such cache memories it is imperative to bring only as much data that can stay there in addition to the data currently needed. If one prefetches too frequently there might not be enough space in the cache to store the data needed for prefetching; or if one prefetches too infrequently then the probability of the process encountering a raw cache miss is still high. Thus, unless software prefetching is tuned properly, implicit hardware prefetching can be difficult to beat. A key challenge in predicting what to prefetch in non-numeric irregular computation such as tree-like data structures and discrete event simulations is that access pattern is data-dependent. In addition to the question of what and when multicore architectures also impose additional concern of where to prefetch. If the data is prefetched by a core other than the one that requires that data we will end up in the problem described in subsection IV-A.

V. CURRENT PROGRESS

We have successfully engineered Parallel Priority Queue [17] (PPQ) data structure for multicore architecture to understand the behavior of traditional priority based data structures on newer chips with multiple levels of memory hierarchy. A priority queue is a multiset of elements where each element has an associated priority. The traditional heap data structure yields only one item at a time and any update along its tree structure is cache unfriendly due to exponentially increasing distance between parent and child nodes. On the contrary, a parallel priority queue facilitates fast access to $k$ top-priority elements, where $k$ is an application dependent constant. An effective parallelization of priority queue data structures would play a critical role in the design of efficient and scalable parallel algorithms for irregularly structured, non-numeric problems such as discrete event simulations and branch and bound problems. For this study we chose PPQ over other similar priority queues because its wide-node structure makes it easier to exploit the spatial locality in a cache memory. Based on the amount of concurrency afforded by a parallel application, it can yield hundreds to thousands of top-priority items at a time. Even for extremely fine-grained applications, it is 2-3 times faster. For larger grain, it scales quickly to linear speedups. Our Research on Parallel Priority Queues also showcases (i) how parallel data structures can employ the synergy of multiple concurrent operations on wide-nodes to increase cache locality, (ii) how pipelined allocation of threads to update tasks at various tree levels can be dynamically shifted to closely follow the cached data, and (iii) how data-dependent cache prefetching can work by splitting a synchronous pipeline into odd and even phases. A challenge in implementing a PPQ on multicore architectures is the significant memory hierarchy with several levels of caches, requiring effective cache and locality management. Three key ideas have been explored in this context, each providing good insights. First, in the usual allocation of processing cores to a pipeline of tasks (along the tree branches), the data moves among the processing cores, thus disrupting cache locality. It is better that cores reallocate themselves to closely follow the cached data IV-A. Second, if one can predict the memory access pattern, a challenge in dynamic systems such tree-based priority queues, the future data can be prefetched in Level-1 caches of specific processing cores IV-B. Prefetching of node data into cache hierarchy as the delete and insert update processes flow in PPQ from root down to leaves was the primary challenge we faced. However, the pipelined access pattern did allow for some prefetching, even though such predictions are data dependent. Third, for fine-grained system, it is extremely wasteful for the processing cores to extract only a few tasks, quickly finish them, and then go back to the task/event queue creating a severe bottleneck competing for the shared data structure. While basic speedup is achieved for coarse grained tasks, concerted effort was needed to bring PPQ’s operational range down to medium to finer grain.

Fig 1 represents the absolute speedup for PPQ. The size of

\[ \text{heapsize} = 2^{26}, k = 2^{13} \]

![Fig 1. Overall performance of PPQ for varying granularity, heapsize = 2^{26}, k = 2^{13}]
heap was $2^{20}$ and the value of $r$ was $2^{13}$. We had a total of 256 cycles of insertion and deletion processes. For large grain sizes, it can be seen that we achieve an absolute speedup of 12.65 using 16 processors. Interestingly, for $p = 7$ and $p = 8$ processors we have superb linear speed up. This is due to the better cache locality afforded by PPQ over serial heap.

VI. REMAINING OBJECTIVES AND CHALLENGES

We have explored both regular and irregular scientific applications to analyze their memory behavior. Although the memory throughput depends on the application employing our data structure, we haven’t seen satisfying results with PPQ. Even for very fine grained execution, PPQ demonstrates poor performance in terms of memory throughput when compared with the results of the memory benchmarking tools on the underlying system. The reason for this is the inherent difficulty of load balancing among available processors. Since PPQ is a tree-based data structure, the cores handling the levels away from root have considerably more load than those handling the levels closer to the root. Moreover, the number of levels do not increase linearly with increasing heap size. PPQ currently does not support allocation of a level to multiple processor therefore it is difficult to employ a large number of processors. We are going to explore and extend other priority based data structures on multicores to analyze their behavior on these newer chips. Learning from this experience we will know the fundamental properties of priority based data structures that adapts well to multicore. For instance, the wide-node data structures appear to work well on multicores compared to single node priority queues. Such properties will guide the design of a new parallel priority based data structure. Our implementation code for PPQ is available online.¹

REFERENCES


¹http://code.google.com/p/parallelpriorityqueues/