Efficient reduction over threads

PATRIK FALKMAN
falkman@kth.se

Master’s Thesis
Examiner: Berk Hess

TRITA-FYS 2011:57
ISSN 0280-316X
ISRN KTH/FYS/--11:57--SE
Abstract

The increasing number of cores in both desktops and servers leads to a demand for efficient parallel algorithms. This project focuses on the fundamental collective operation reduce, which merges several arrays into one by applying a binary operation element wise. Several reduce algorithms are evaluated in terms of performance and scalability and a novel algorithm is introduced that takes advantage of shared memory and exploits load imbalance. To do so, the concept of dynamic pair generation is introduced which implies constructing a binary reduce tree dynamically based on the order of thread arrival, where pairs are formed in a lock-free manner. We conclude that the dynamic algorithm, given enough spread in the arriving times, can outperform the reference algorithms for some or all array sizes.
Conclusions

General conclusions

Recommendations

Note on compilers

Future work

References
Introduction

The road to multicore processors

During the last years, we’ve seen a decrease in the speedup of the clock frequency on silicon chips. Several factors are setting the boundaries as to how far the frequency can be pushed, where top contributors include heat dissipation (power wall), high memory latency (memory wall) and limitations in instruction level parallelism (ILP Wall). By shrinking the transistors, the heat output can be reduced allowing for higher frequencies, and by increasing the clock frequency of the main memory along with increased cache size, we can overcome parts of the problem with the memory wall. The ILP wall is slightly more complicated to break and details related to that is out of the scope of this project, but with approaches like predicting conditional branching, out of order execution and pipelining we can overcome some parts of the problem. All these approaches allows a small bumps in CPU performance, but it’s not enough to satisfy Moore’s law which loosely speaking says that the number of transistors on an integrated chip doubles every two years.

To circumvent this problem and keep up with Moore’s law, manufacturers are now placing two or more independent processors on a single integrated circuit die, known as a multi core processor, where core refers to an independent processor on the die. This allows each core to simultaneously execute its own set of instructions, enabling a significant speedup over a single executing CPU. If we parallelize a program to several threads utilizing the different cores, we can overcome many of the limitations created by the ILP wall by distributing the instructions to several cores. However, using multiple cores doesn’t address the memory wall problem and if the memory is unable to keep up, either due to bandwidth limitations or latency, the CPU will starve for instructions. This can be mitigated by larger caches with deep hierarchies combined with smart caching algorithms, but still remains a critical problem.

The thermal design power (TDP) is a measurement of how much heat the cooling system of a CPU should be capable of handling, and when placing several cores on one die, the TDP is shared among the cores. Given that, reducing heat output is a crucial task for both single core and multi core CPUs.

Running two cores at the same clock speed will result in significantly more heat output. One approach to reduce the heat output in the multi core architectures is to control the frequency of the cores individually. When one core is idle, that core can have its frequency reduced or even be completely suspended, requiring less voltage and resulting in reduced average heat output.
This can also be utilized to temporarily boost the frequency of one core (or a fraction of them) to more than the frequency it would run at if all cores were utilized. This is good if the running application only uses one or a few threads (knows as Turbo Boost by Intel and Turbo Core by AMD)

The general idea with multi core processors is to scale by using more cores rather than higher frequencies.

**Basic architecture of multicore CPU’s**

A multicore CPU can be seen as a single physical unit with two or more complete execution cores. The cores may or may not be identical, meaning that in some systems cores are specialized on specific tasks, known as heterogeneous systems whereas in the more common approach, the homogenous, all cores are identical.

The above figure illustrates a common architectural layout which features a three level cache hierarchy the third is shared between the cores. Having core dedicated caches might give the benefit of less data contention than if it were shared with other core(s), however, sharing the
cache provides other benefits such as the possibility to allow on core to utilize a larger portion of the cache if needed. The memory controller is in this example placed outside of the CPU die, but newer CPU’s (like the Intel Nehalem series) usually have the controller integrated on the same die for increased bandwidth and reduced memory latency.

**Parallelization and reduction**

For one application to take advantage of several processing units, it needs to be *parallelized*, meaning the problem to be computed is divided smaller parts that can be computed simultaneously. This in itself is a very non-trivial task, and in some cases it might not even be possible. There are several ways of dividing a problem into smaller tasks that can be executed individually, but a key point is that code may not depend on data from another thread. For example, many recursive and dynamic algorithms depends on previously calculated data, and if that data is assigned to and calculated by another thread, we have no guarantee that it’s available to us when we need it and the algorithm may fail unless carefully synchronized. It’s also generally hard to parallelize algorithms that directly depend on a calculation from a previous step, as there are trivially no parts that can be calculated independently, typical example are iterative numerical computations. This is known as inherently serial problems. The opposite of this is known as an embarrassingly parallel problem which means that it’s trivial to divide the problem into smaller tasks, e.g. in animation, each frame can be calculated independently.

When a problem has been divided and computed, each thread will possess a local result which might be a single value or a larger set – an array. For the result to make sense the local results must be collected to a global result, and usually when collecting data, a binary operation, such as addition is applied. This act is known as a reduction and is done by applying the binary operation for every element in the set to be reduced, yielding a set of the same size as the input buffers. In this paper, we assume the operation used to be associative and commutative.

**Problem statement**

Given a number of threads that hold a copy of an array of a fixed size, which is the most *efficient* algorithm to reduce these to one array? The term *efficient* above may be defined in two ways; the minimal amount of cycles from that the last thread hits the reduction until the data is located in the destination buffer, or, the total accumulated cycles spent in the reduction by all threads.
Theory

Shared memory in contrast to MPI

In larger computer clusters, where nodes don’t share memory, the tasks are divided over different processes rather than threads. This requires the nodes to have some means of communication in order to pass information that otherwise would be directly available to them via shared memory. An API has been developed to ease such communication, known as the Message Passing Interface (MPI). This is used both for signaling and passing of actual data and has, among many things, support for reduction. The choice of algorithm used for reduction in MPI is up to the implementation, i.e. it’s up to the developer to choose the most effective one. Several papers have been published evaluating and proposing algorithms for efficient reduction over MPI, most of which are based on a tree structure. When developing algorithms for MPI the most important factors to take into consideration is the latency and available bandwidth between nodes. Depending on the topology which may vary significantly, different algorithms might be more suitable than others, often depending on whether latency or throughput is prioritized.

Turning focus to the multi core and multi socket systems where each computing node (thread, in this case) has direct access to each other threads data, we can take advantage of that in several ways. The latency to read data generated on one core to another core is very low and depends on the architecture of the processor, for example, whether it has local interconnects between the cores or not. If the requested data in not in any of the processors cache’s, the main memory has to be consulted, causing a significant delay, known as a cache miss. Although this is largely avoided by using larger and smarter caches, some of those evaluated in this paper being up to 24 MB. For inter-socket communication the bottleneck is the interconnect between the sockets.

SMP and NUMA

The cores in a multicore CPU can be seen as independent processors in the sense that they are able to independently execute code. To access the main memory, the cores are connected to a memory controller which in older systems usually was located outside of the multicore CPU die, but in newer systems, e.g. the Intel Nehalem, they are integrated on the same die (1). The memory controller is connected to the cores in a symmetric way, treating all cores equally which means that their memory access is uniform. This can also scale beyond one CPU by connecting one or more additional CPUs to the same bus (though this would require the
memory controller to be placed outside a CPU). The problem with this approach is when adding more CPUs all sharing the same bus, they will start competing for memory and the contention hinders efficient scaling. The technique is commonly known as symmetric multiprocessing (SMP).

To achieve better scaling, the numbers of CPUs sharing a bus can be limited by utilizing several buses which then are linked using an interconnect. Each bus naturally has its own set of memory which the CPUs that shares the bus has quick access to. In modern implementation of this approach there is typically one multicore CPU sharing a bus and a set of memory. There are many alternatives as to how these buses are connected to each other, but common for all is that memory access is non-uniform. This means that when a CPU is trying to access any memory, the latency and bandwidth to that particular data depends on where it’s located, with the best timings being the memory that is connected to its bus. This is known as non-uniform memory access (NUMA) and for programs to perform efficiently on these systems, they must be aware of the NUMA by utilizing memory locality to the extent it’s possible.

**Trees**

After a computation using N workers (threads / processes), we have N sets of data that is to be reduced into one set (array). The most trivial way to do this would be to have one worker do the reduction sequentially, requiring as many iterations as there are workers. This is highly inefficient since we only utilize one of the workers, leaving the rest to idle. For optimal parallelization, we strive to keep all workers busy at all times without doing redundant or unnecessary calculations.

An attempt to parallelize the reduction can be to have each process/thread reduce 1/N of the array. This would be highly inefficient in MPI topologies for several reasons. Firstly, before the reduction can start each process would have to send its fraction of the array to every other, and then receive a fraction from every other worker (N*MPI_Scatter, or, MPI_Alltoall). Secondly, after the reduction every workers needs to send its fraction of the result array to the root node which gathers them to a global result (MPI_Gather). The overhead of sending fractions and that every workers needs to communicate with every other worker along with the latency makes this approach unsuitable for MPI, but when the workers shares the address space as they do with threads the latency of fetching data from other threads is greatly reduced, especially in SMP architectures.

A more common approach is to use a tree like reduction, typically a binary tree were threads reduce pairwise upwards towards the root node. For example, given four threads, one and two would reduce, three with four and lastly one with three. This allows us to run the first two in
parallel, resulting in a total of two iterations compared to three in the trivial case. More generally, using a tree style reduction, it takes \( \log_2 N \) iterations to complete a reduction, where iteration refers to a level in the thread.

## Thread synchronization and communication

### Mutual exclusion

Writing code that is to execute simultaneously raises many problems that don’t exist in traditional sequential programming, a common is known as a race condition and occurs when two or more threads compete for a shared resource. For example, it could be as simple as incrementing a variable that is globally shared between two threads. If it is initialized to zero, both threads could at the same time load the value zero to a register, increment it, and write the new value of one back while the programmer probably expected a value of two to be written back.

Areas in the code dealing with shared resources are known as critical sections, and there are few ways of dealing with critical sections like the above mentioned example. Many code languages and libraries provide synchronization primitives that allows for mutual exclusion so that race conditions can be avoided.

Unfortunately, providing efficient mutual exclusion is and remains a relevant problem as of today. The first algorithm to provide mutual exclusion in software was invented by Dekker (2) which works by using two flags that protects entrance to the section and loops that spins to check for changes in the lock. When a thread is spinning in a loop waiting to acquire a lock, it is known as a busy-wait since the executing thread is busy, consuming CPU cycles while really accomplishing nothing.

### Atomicity

The best solution to the above mention problem would be if the CPU could increment the value of the counter without requiring mutual exclusion. Manufacturers have realized this and for many types of architectures there are atomic primitives implemented in hardware which allows for this.

### Compare and swap

Compare and swap (CAS) is a low level atomic CPU instruction that given a reference value, a new value and a destination, compares the reference value to the destination value, and if, and
only if they are the same the new value is written to the memory. All of this is done atomically provided there is hardware support for the instruction, which it has on x86 and Itanium, for example. This is crucial as it provides the functionality many other synchronization primitives rely on.

**Thread yield and suspension**

When a thread has reached a point where it’s unable to continue before a certain condition is met, one could suspend the execution of that thread, allowing for other threads/processes that might be waiting to execute. The operating system keeps a queue, known as a ready queue, where it keeps threads/processes that compete for CPU time. For a true thread suspension to take place, meaning the thread is pulled from the ready queue, the user must make a request to the OS to do the pull which naturally implies that the OS needs to support the suspension.

Another way of achieving a similar effect to complete suspension is to let a thread yield for as long as it needs to wait. By yielding, the current executing thread/process gives up its current *time slice* and is placed at the back of the ready queue (3).

**Spin locks**

Instead of suspending a thread that is waiting for a certain condition, it can continuously check the status of the condition by *spinning* around it. This counts as a busy waiting as the thread consumes as much CPU as it gets and never yields until the condition is satisfied. The main benefit of spinlocks is the very short delay between the time when condition is met and the thread acquires the lock, making them possibly faster than synchronization primitives provided by the OS (4). Spinlocks is generally appropriate in scenarios where the overhead of suspending a thread is greater than the time the spinlock waits.
Methodology

Dynamic pair generation

Motivation
When dividing a job to run into several threads which in turn runs on several cores, load balancing and scheduling becomes important aspects to avoid wasting CPU time. Load imbalance, where one thread is given more work than another, is usually an effect of the algorithm used to divide the work is not perfect, which rarely is the case. Other processes in the operating system are also competing for CPU time, and unless explicit prioritizing has been specified content switches will also contribute to imbalance. Most threaded libraries allows the developer to decide whether to explicitly pin threads to specific cores, or let the operating system do the scheduling (5). Explicit pinning may prove beneficial but requires deep understanding of the application behavior, environment and the architecture of the system that is to run the program (6). If it is done incorrectly it may instead have a negative impact on performance, but avoiding it may cause heavy inter-core content switches resulting in possibly unnecessary overhead.

All of the above mentioned factors contribute to a spread in the arrival at which threads reaches synchronization points. Many of these operations require a barrier before the synchronization operation can start and by doing that, we have to wait until the last thread reaches the barrier before the operation can start. However, some operations can begin without having to wait for all threads to arrive.

Idea
To optimize the reduction algorithm, we can try to take advantage of the idle time caused by the imbalance. If we can keep cores from idling while they wait for the last thread to arrive at the reduction, we could possibly use this CPU time to speed up the reduce operation.

When using a tree style reduction, one could avoid the problem with a global barrier by having local barriers for each pair, meaning two threads in a pair only has to wait for each other before they can start the first iteration, i.e. a local pair barrier. The pairs are typically statically assigned, i.e. thread with rank zero reduces with thread $2^i$ where $i$ increases by one for every iteration until it’s done. Even though we avoid waiting for all threads to arrive, each thread is still depended on a specific, statically assigned neighbor. The problem with that is if the thread
with rank zero reaches the reduction first and its neighbor does it last, it will still stall the reduction as much as it would with having a global barrier.

**Algorithm**

As a way to avoid this problem, this paper presents several algorithms that takes advantage of the concept of dynamic pair generation (DPG). This still relies on a pairwise tree like reduction, but instead of assigning the pairs statically, they are assigned in the order in which threads arrive at the reduce function. When a thread arrives at the reduce function, there are two possible scenarios:

1. No thread is waiting to be reduced and hence we are unable to form a pair. We have to wait for another thread to reach the reduction before we can act.
2. A thread is waiting with which we can form a pair. We alert our neighbor that either can help with the reduction or return from the reduction as soon as we have fetched its data.

To act of dynamically pairing up neighbor is subject to heavy race conditioning, especially if the thread arrives close to simultaneously. The fundamental approach to deal with this problem is by using a flag and the atomic compare and swap operation. The flag either is set to a static value implying that no thread is waiting, or it is set to the rank of the thread that is waiting. When the threads arrive in a close to simultaneously manner, there will be some cache contention for the flag, but as no thread spins around the flag there it won’t have any significant impact on the results.
static unsigned int get_neighbor(unsigned int flag, myrank)
{
    unsigned int nbr;
    do {
        /* Assume there is no thread waiting. Try to set the flag */
        if(Atomic_cas((global_flag), REDUCE_MISS, flag))
        {
            /* no other thread was waiting. flag set and we can break */
            return REDUCE_MISS;
        }
        /* we don't want anything that happens after this point to
         * propagate before the barrier */
        Atomic_memory_barrier();

        /* arriving here means a thread could be ready for reduction,
        * or, a thread was ready but someone else picked it before us */
        nbr = Atomic_get(global_flag);

        /* We loop here cause if both CASes returns false, it means we had
        * a race condition where another thread snatched the waiting thread
        * before we could exclusively acquire it */
        } while(nbr == REDUCE_MISS ||
            ! Atomic_cas(global_flag, nbr, REDUCE_MISS));

        /* arriving here means we exclusively acquired the neighbor
        * and should start the reducing with it. */
        return nbr;
    }

    A key point to note here is that no locking mechanism is used, providing a non-blocking
    function which is a crucial property for efficient scaling. As we can see, there is a loop around
    the code that sets the flag for those cases where there has been a race condition. Although the
    loop is necessary to protect from these cases, they are fairly uncommon and occurs roughly only
    every 500:th run. The cost to run this function averages to a few hundred cycles depending on
    the system and in rare cases peaks to a few thousand.
Quick return

Motivation

In large simulations, the work that is parallelized and assigned to threads is often built as a chain of different steps separated by synchronization points. This chain can then be run repeatedly in iterations with different input data. In that chain, the step after a synchronization point may or may not be dependent on a result generated in that point. For example, if the synchronization point is a reduction, the following step may use the result of the reduction as input data and hence rendering it unable to continue until the reduction is complete. In that case, it would be in the developers’ interest to minimize the time from the last thread to hit the reduction until it’s available to all threads.

A hybrid parallelization using threads for communication within a socket and MPI for external socket ditto is not uncommon and proved to be an efficient approach, especially in topologies with limited bandwidth as the passing of data between sockets gets explicit (7). For this to work one thread handles the MPI communication and before it does some communication there typically lies a synchronization point between the threads, e.g. a reduction. If the purpose of the reduction is to gather data which is a dependency for another node, chances are that the threads are not dependent on the result of the reduction and can continue working as soon as they return from the reduction. In that case, we are more interested in minimizing the total amount spent in the reduction by all threads. The scenario where the threads are independent of the reduction result is not exclusive for hybrid parallelization and may also occur in a threaded only situation.

Idea

By letting threads returns as soon as possible they can continue with other work, given that work is independent on the result of the reduction. The key to minimizing the total time spent in the reduction is elimination of idle time due to threads waiting to synchronize. This coincide with the dynamic pair generation, as its wait time only consists of the time every other thread that arrives has to wait until the next thread arrives. In contrast, a static assigned tree algorithm needs to wait for a specific thread, which could arrive significantly later than its neighbor. Worst case being the linear approach (detailed in next chapter) which has to wait until all threads arrive before the reduction can start and hence before any can return.
Algorithms

Linear

This algorithm works by letting each of the N threads reduce 1/N of the array. Assuming each thread contains an array of data to be reduced, and then each thread reduces N fractions of the array.

A major drawback of this algorithm is that it requires all thread to rendezvous, i.e. reach the reduce function before the reduction can start as all the data need to be available to all threads. Implementation wise this means a barrier is placed early in the function to provide this guarantee. Also, if the program cannot guarantee the input buffers to remain untouched when threads return from the reduction, a barrier is also required at the end of the algorithm. Another drawback is that each thread needs data from every other thread before the reduction can start, causing heavy contention.

As the threads are assigned a fraction of the array and handle that part independently, there is no need for thread communication or synchronization during the actual reducing and hence we avoid the overhead associated with this. We also achieve optimal load balancing where each thread is assigned an equal amount allowing each core to work constantly and never starve for data.

The algorithm basically consists of two loops, one iterating over the index in the array and the other one over the threads. There is obviously a fundamental choice to make related to this with to options, which of the loops should be the outer and which should be the inner. Trying to predict this is very hard as the answer depends mostly on the caching of the targeted architecture. Experiments showed that it was more beneficial to place the thread loop as the outer loop and have the array indexing being the inner.

The Linear algorithm
Static

As mentioned in the theory section, tree like reductions are common in implementation of MPI reductions, but many of the benefits of trees applies to SMP environments as well and this is a tree style reduction implementation. The word static naming this algorithm comes from the fact that the pairs are statically assigned in a predefined manner. The pair assignment is based on the rank of the threads, starting with an adjacent rank and increasing the distance exponentially for each iteration.

When reducing pairwise, the threads start off with two arrays that are to be reduced to one, meaning one thread, called master thread, will have both threads buffers reduced to its buffer, leaving the other thread, the slave, obsolete after the pair has run its operation. That thread can return once it’s certain the neighbor thread has reduced all data from its buffer.

As both the master and the slave have access to one another’s buffer, they split the work having the master take the first half and the slave the second half. This way the operations on the arrays are parallelized between the threads providing a potential speedup. An important point to make here is that both threads need to be guaranteed that the data in its neighbor’s buffer remains available and unchanged throughout the operation. This implies that the slave can’t return from the function and the master cannot proceed to the next iteration until they both are certain that its neighbor is finished. To solve this dependency the threads needs to synchronize once it’s done reducing, and since the operation should take about an equal amount of time, the synchronization is implemented using spinning waits. This is due to the overhead of suspending a thread takes longer than the time the thread usually has to wait.

Dynamic

This is a first proposal to an algorithm that takes advantage of the irregularities in thread arrivals and utilizes the ideas mentioned in dynamic pair generation under methodology. Upon arrival at
the reduction, the DPG algorithm described previously is called, deciding if a thread is to wait for another to arrive or if it should reduce with an already waiting thread.

When a pair is formed, the thread that arrived later becomes the master in the reduction and will continue onto the next iteration. The other, which arrived earlier will break and return once its data has been reduced and placed in the buffer at the master thread. To provide a potential speedup of the actual operation, the buffers are split between the pair in the same way as the static algorithm. After the master has reduced with the slave it will, assuming there are more threads waiting to reduce, go for another iteration and will remain a master or become a slave depending on if there is a thread waiting or not.

A problem with this idea is determining when the reduction is complete and the master thread should return instead of searching for a new neighbor. Using a global counter could be an option, but an obvious side effect is another potential bottleneck and a point for cache contention. As forming pairs already pass a point of communication, the flag used to signal if a thread is waiting or not, it is preferable to use for this to pass the information of how many thread the slave has reduced. The flag used for communication is a 32 bit word, so splitting that giving 16 bits for the rank and 16 bits for the reduction count, yielding an upper limit of \(2^{16}\) threads.

Analyzing how this algorithm would behave when threads arrive with a certain spread leads to a hypothesis. If the spread is large enough to allow pairs to complete one or more iteration before the last thread arrive, it is likely that the last arriving thread can skip one or more iterations in the tree. For example, let’s say there are 4 threads reducing, requiring a total of 2 iterations to complete, let’s say one iteration takes time \(S\). Now assume threads 1, 2 and 3 arrives at time \(t=0\), and thread 4 at \(t=1.5S\). Thread 1,2 can start reducing at \(T=0\), and at time \(T=S\) the master thread (let’s say it’s thread 1) in the pair can pair up with thread 3 and start reducing. When thread 4 arrives, 0.5\(S\) later, it has to wait another 0.5\(S\) before it can start reducing with thread 1. This means that in total, the time it takes from the point of the last thread to hit the reduction until the reduction is complete, a total time of 1.5\(S\) has passed instead of 2\(S\) which it would have taken if thread 3 had to wait for thread 4 in a non-dynamic fashion.

Theoretically, if the spread is large enough, it will only take one iteration from that the last thread reaches the reduction until it’s complete, instead of \(\log_2(N)\) steps in the trivial tree reduction.
Dynamic Group

The group version is an extension to the dynamic algorithm. Its fundamental outline is the same, with one exception. In the normal dynamic, when the slave in a pair has had its data reduced, it returns from the reduce function and unless the threads next task is independent from the reduction result, it will go to idle, wasting CPU. To avoid this wasting while still keeping the dynamic pair generation, the threads that are to return will instead help other pair with its reduction. This involves introducing two additional functions:

- **void join_workforce().** This is called by threads that in the normal version would return. Instead, they enter a loop where they wait until they are woken up by the master thread in a new pair that is to start reducing. When woken it will, along with the other workers, help the pair by reducing a fraction of the target array. They all split the buffer equally, giving each thread $1/(2 + \text{nr. workers})$ of the array.

- **int alert_workers().** This function is called by the master thread in a pair just before it is to start reducing. It takes as arguments pointers to the buffers.

The workforce shares a struct containing pointers to the buffers and a counter keeping track of how many workers are available. To avoid the race condition that might occur if threads try to join the workforce while another thread is trying to alert and wake them, a lock is protecting the entrance to the workforce and the calls to `alert_workers()`. When workers are being woken up, they’ll need to fetch the information from the struct, and in order to keep the data consistent, the thread that is to alert the workers acquires the lock, but only the last worker to awake releases it. To achieve this consistency, the technique of passing the baton is used (8).
Evaluation

To determine the efficiency of the algorithms there are several factors that needs to be taken into consideration:

- The spread in arrival times. Threads in real situations reach the reduction function at different times, so to simulate this threads are stalled. This is done by a function that stalls threads in a linear distribution of up to a mean of 1M cycles with a standard deviation of 200000.
- The number of threads. Each algorithm may scale differently with a growing number of threads, and to evaluate this algorithms are run with 4-32 threads.
- Array size. To investigate the impact the size of the array has on the algorithms sizes from 1000-200000 floats are tested.
- The architecture.
- Compilers.

The data type used for the evaluation is 32 bit floating points and the reduce-operation is addition. The Y-axis is divided with the size to provide a better overview as the amount of cycles scale in a linear order with the size. The label size on the X-axis is the number of floats in the arrays to be reduced.

Implementation

All code is implemented in the C language and compiled with various compilers. To achieve atomicity, all operations that require so are implemented using inline assembly instructions to ensure correctness and efficiency. Other critical parts, such as spin locks, are also implemented using inline assembly according to recommendations from the manufacturer. The threads are managed using the POSIX thread library.
Algorithm overhead

To measure the overhead associated with each algorithm they can be run with arrays of size of only one value, leaving the total time being practically only synchronization overhead. As the amount of synchronization is independent of the array size, the timings measured here do not grow with the size of the arrays. Many synchronization bottlenecks are due to heavy contention of a shared resource, e.g. the switching flag in the dynamic algorithm. The worst case scenario in those cases are when all threads compete at the very same time, which is what happens to the reduction flag, among other, when all thread reaches the reduction simultaneously. This happens rarely in practice, the overhead is evaluated using two cases one being no spread at all, and the other with a spread of max 128000, mean of 37000 and standard deviation of 18500. The latter case with the stall measures the overhead from the point at the last thread hits the reduction until the data is completely reduced, omitting overhead prior to this point. This means that it could even exclude everything but the overhead for the last iteration in the dynamic case. However, these measurements may still be of interest since much of the synchronization, and even several iterations, can take place prior to the arrival of the last thread in practice when running with larger sets of data. In these cases, the synchronization costs that occur prior to the last thread arrival can take place when the thread otherwise would be idle, waiting for a new neighbor.

SMP

<table>
<thead>
<tr>
<th>Threads</th>
<th>Dynamic Xeon</th>
<th>Dynamic AMD</th>
<th>Static Xeon</th>
<th>Static AMD</th>
<th>Linear Xeon</th>
<th>Linear AMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1577</td>
<td>4168</td>
<td>878</td>
<td>2195</td>
<td>802</td>
<td>2382</td>
</tr>
<tr>
<td>4</td>
<td>2722</td>
<td>8492</td>
<td>1927</td>
<td>3974</td>
<td>1541</td>
<td>3443</td>
</tr>
<tr>
<td>6</td>
<td>3896</td>
<td>12379</td>
<td>3401</td>
<td>5649</td>
<td>2664</td>
<td>4499</td>
</tr>
</tbody>
</table>

*Intel Xeon E5645 vs AMD Opteron 8425, Intra-socket overhead (cycles)*

We can see that the Intel CPU outperforms the AMD in every algorithm, with the difference being most apparent at the dynamic algorithm where it’s more than 3 times more on the AMD at 6 threads. No further analysis has been made in this work to understand the differences in these results, but factors that are suspected to affect these are cache coherency protocols and latency between levels of caches.
When the threads arrive with some inconsistency, or *spread*, we can see that the dynamic version benefits the most from it, followed by the static whereas the linear performs worse. This is to be expected since the dynamic version can progress as soon as *any two* threads have reached the reduction, whilst the static can progress once *two specific* threads have reached the reduction. The linear version is unable to progress until *all threads* have arrived.

**NUMA**

For these tests the scheduling is handled by the operating system, meaning the threads are likely to be distributed across sockets as soon as possible. An alternative would be to manually set the affinity and only put threads on a new socket once the previous is full. The latter test would mix the scaling of NUMA with intra-socket (SMP) performance. Instead, to see how the algorithms scales over sockets, two architectures are evaluated, one quad socket 8 core Xeon and one quad socket 6 core AMD.
We can first note that there is no significant bump at the between 6 and 8 threads (or earlier) indicating that the threads were already distributed among both sockets at 2 threads. Comparing how the algorithms scale with several sockets we can see that there is a clear difference between Intel and AMD in several aspects. First, if looking at the case of 24 threads which means all cores are utilized on the AMD system and 6/8 cores are used on the Intel, none of the algorithm’s overhead stretches above 25000 cycles on AMD, whereas on the Intel system, none takes less than 60000. Looking specifically at the linear algorithm, it scales considerably worse on Intel where it lands on just below 100000 cycles compared to 25000 on AMD.

What characterize the linear algorithm is that it fetches smaller chunks of data than what the other algorithms do, but it fetches these chunks from every other thread causing contention on the socket interconnects. Also when fetching more but smaller pieces, the latency becomes increasingly important and one possible explanation could be that AMD’s interconnect HyperTransport has lower latency than Intel’s QuickPath Interconnect, although this is not a conclusion that can be concluded from this data.
The last two benchmarks is the same as the first two, but with the same spread as in the SMP tests. The pattern in the differences between the two reassembles of that in the SMP architecture, and this is to be expected for the same reasons mentioned in that section.
Notes on dynamic group algorithm

The dynamic group algorithm has been excluded from the benchmarks here as it its timings are so different from the others. The table below contains a summary of the scaling on different systems, both SMP and NUMA.

<table>
<thead>
<tr>
<th>Threads</th>
<th>Single AMD</th>
<th>Single Xeon</th>
<th>Quad AMD</th>
<th>Quad Xeon</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7655</td>
<td>2066</td>
<td>8346</td>
<td>6541</td>
</tr>
<tr>
<td>4</td>
<td>21510</td>
<td>8580</td>
<td>21867</td>
<td>36600</td>
</tr>
<tr>
<td>6</td>
<td>38489</td>
<td>18653</td>
<td>43479</td>
<td>73522</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
<td>-</td>
<td>367637</td>
<td>338129</td>
</tr>
<tr>
<td>24</td>
<td>-</td>
<td>-</td>
<td>1.61953e+06</td>
<td>1.60352e+06</td>
</tr>
<tr>
<td>32</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2.68088e+06</td>
</tr>
</tbody>
</table>

*Overhead scaling of Dynamic group algorithm*

The reason as to why it scales so bad has to do with the workforce which act as slaves and helps other pairs with their reduction. For every thread that joins the workforce, additional synchronization is required at each step (pairwise reduction) and given enough spread, for the last iteration, all threads need to synchronize with each other. This obviously gives a lot more synchronization overhead, but also as more workers joins, the chunks each thread reduces gets smaller and the algorithm starts to suffer from the same problem as the linear algorithm, especially in the NUMA architectures.
Optimal scenario: no spread

This test aim to see how the algorithms would perform in the optimal case where each thread reaches the reduce function at the same time (± 200 cycles). This is likely to have a negative impact on the algorithms with heavier communication overhead.

Intel Xeon X5650, 6cores, ICC 12.0.2
Evaluation

AMD 1090T, 6 cores, gcc 4.5.1

The sudden dip at the 64k size mark is due to the cache being unable to keep all the required data causing some cache misses.

Dual Xeon E5645, 2x6 cores, gcc 4.5.2
Evaluation

Dual AMD Opteron, 2x12 cores

![Graph showing performance comparison](image1)

Quad Xeon E7-4830, 4x8 cores, ICC Version 11.1

![Graph showing performance comparison](image2)

**General case analysis**

We can see that the linear algorithms performs very well which is to be expected since it requires no synchronization during the run and utilizes all cores at all times, achieving good load balancing. It does, however, start to lose to the tree algorithms on the 24-core NUMA
AMD. A theory as to why the linear isn’t the fastest throughout these benchmarks is the latency penalty associated with fetching small chunks of data from many different locations.

Spread of max 15000, mean 8000, stddev 3900

Intel Xeon X5650, 6cores, ICC 12.0.2
It’s worth noting the drastic improvement of both the static and the dynamic version. The dynamic algorithm goes from being significantly slower at all sizes to being faster for sizes > 5000.

Dual Xeon E5645, 2x6 cores, gcc 4.5.2
Dual AMD Opteron, 2x12 cores

Quad Xeon E7-4830, 4x8 cores, ICC Version 11.1

General case analysis

We can see that the linear algorithm remains unchanged which is to be expected as it requires all threads to reach the reduction before it can start.
Spread of max 128000, mean 37000, stddev 18500

Intel Xeon X5650, 6 cores, ICC 12.0.2
Evaluation

AMD 1090T, 6 cores, gcc 4.5.1

Dual Xeon E5645, 2x6 cores, gcc 4.5.2
Dual AMD Opteron, 2x12 cores

Quad Xeon E7-4830, 4x8 cores, ICC Version 11.1

General case analysis

We can see that the dynamic algorithm improves further and is, with one exception, the fastest for sizes up to 5k floats.
Evaluation

Spread of max 555k, mean 184k, stddev 102k

Intel Xeon X5650, 6cores, ICC 12.0.2
AMD 1090T, 6 cores, gcc 4.5.1

![Graph for AMD 1090T, 6 cores, gcc 4.5.1]

Dual Xeon E5645, 2x6 cores, gcc 4.5.2

![Graph for Dual Xeon E5645, 2x6 cores, gcc 4.5.2]
Dual AMD Opteron, 2x12 cores

Quad Xeon E7-4830, 4x8 cores, ICC Version 11.1

General case analysis

In this test with the greatest spread, the dynamic algorithm outperforms all other for sizes up to at least 10k floats on all architectures. In other words, the dynamic algorithm is dominating for arrays of twice the size than in the previous tests.
Total time spent in reduction

These tests are done on a system running quad Xeon E7-4830, 4x8 cores, ICC Version 11.1.

If we compare this to the previous benchmark on the same system, where the measurement is the time from the last thread to hit the reduction until completion to this, there’s one interesting observation to make. In that case, the linear algorithm becomes the fastest algorithm for size slightly above 10k whereas here, the linear is slower in the whole range, and considerably slower for sizes < 5k. This is however to be expected and can be explained by the tree algorithms ability to return earlier, whereas the linear has to wait until the reduction is complete. The tree algorithms, the static and dynamic in this case, can let one thread in each pair return in every level of the binary tree, meaning after the first iteration, half of the threads can return, and on the second iteration half of the remaining, and so on. The timings for the linear algorithm here is essentially the number of thread times the time for the last thread to hit the reduction till completion.
The above graph shows a slight improvement for the dynamic algorithm whereas the static and the linear show a significant declination. The dynamic proved to outperform all other with spread as low as 3500 cycles (standard deviation).
Conclusions

General conclusions

By evaluating the different algorithms in various systems with varying spread we can establish that neither proved to be universal preeminent, yet there are reoccurring patterns which are of interest.

The practical results support the theory behind the dynamic pair generation; with increasing spread, the algorithm based on the dynamic theory performs better overall, but especially in relation to the other, non-dynamic algorithms. When looking at time from last thread to completion, the linear algorithm is often superior for larger sets of data (starting from 15k floats), even with a significant spread. NUMA system running AMD seems to deviate from this pattern, and the linear is being outperformed by the static and occasionally the dynamic algorithm.

The static algorithm is generally faster than the dynamic with no spread in terms of last thread arrival to completion, but with just a little spread the dynamic becomes faster for small sizes and as the spread increases, the dynamic becomes faster for larger and larger sizes. When measuring efficiency as the minimal accumulated time spent in the reduction, the dynamic becomes faster very quickly, and is in some systems superior on all measured sizes with a spread of only 2-3k cycles. The dynamic group algorithm showed a decline when scaling due to the synchronization required with every step of the reduction. It did, however, outperform all algorithms in certain SMP systems.

Recommendations

As mentioned in the conclusion, no algorithm is superior throughout any given spread or size, hence giving a general algorithm recommendation is hard. But with a known architecture and array size, one could estimate or measure the thread spread and can by combining these factors narrow the available options. Below are two tables which summarizes the results from the evaluation, divided between SMP and NUMA and categorized in terms of spread and size. The recommendations are based on timings from last thread arrival until completion and the three spread option corresponds to those in the evaluation (minus from the zero spread).
Conclusions

<table>
<thead>
<tr>
<th></th>
<th>Small Array &lt;5k</th>
<th>Medium array &gt;5k</th>
<th>Large array &gt;24k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low spread</td>
<td>Intel: Dynamic</td>
<td>Linear</td>
<td>Linear</td>
</tr>
<tr>
<td></td>
<td>AMD: Static</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Medium spread</td>
<td>Dynamic</td>
<td>Intel: Dynamic</td>
<td>Linear</td>
</tr>
<tr>
<td></td>
<td>AMD: Linear</td>
<td></td>
<td></td>
</tr>
<tr>
<td>High spread</td>
<td>Dynamic</td>
<td>Dynamic</td>
<td>Intel: Dynamic group / Linear</td>
</tr>
<tr>
<td></td>
<td>AMD: Linear</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Recommended algorithms for SMP systems*

<table>
<thead>
<tr>
<th></th>
<th>Small Array &lt;5k</th>
<th>Medium array &gt;5k</th>
<th>Large array &gt;24k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low spread</td>
<td>Intel: Dynamic</td>
<td>Intel: Dynamic</td>
<td>Intel: Linear</td>
</tr>
<tr>
<td></td>
<td>AMD: Static</td>
<td>AMD: Static</td>
<td>AMD: Static</td>
</tr>
<tr>
<td>Medium spread</td>
<td>Dynamic</td>
<td>Intel: Dynamic</td>
<td>Intel: Linear</td>
</tr>
<tr>
<td></td>
<td>AMD: Static</td>
<td>AMD: Static</td>
<td>AMD: Static</td>
</tr>
<tr>
<td>High spread</td>
<td>Dynamic</td>
<td>Dynamic</td>
<td>Intel: Linear</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>AMD: Static</td>
</tr>
</tbody>
</table>

*Recommended algorithms for NUMA systems*

**Note on compilers**

The compilers used in the benchmarks were the one that yielded the best results, or, in some cases, the one that gave the best result of those available on the test system. Although several compilers were used in the evaluation, no side by side comparison is illustrated in the evaluation. This is mostly due to that there were no significant different between the algorithms, but rather a slight improvement on all algorithms when a better compiler was used. The highest difference noted was an improvement of around 10% when using an Intel compiler rather than GCC on a Quad Xeon system.
Future work

Although synchronization overhead were taken into consideration when implementing the dynamic algorithm, such as choosing spin locks rather than yielding or suspension when a pair waits for its neighbor after reducing half the buffer, there is still room for improvement.

In NUMA systems, there many cycles to save by having one reduce flag per socket instead of one shared among them, allowing for each socket to work in an SMP-manner which would reduce latency and contention to the memory and reduce flag. It’s only when all threads are reduced within one socket that inter-socket communication begins.
References


