Implementation of Parallel Set Intersection for Keyword Search using RapidMind

Fady Samuel, Jérémy Barbay, and Michael McCool

David R. Cheriton School of Computer Science,
University of Waterloo, Canada

University of Waterloo Technical Report CS-2007-12

Abstract. The intersection of large ordered sets is a common problem in the context of the evaluation of boolean queries to a search engine. In this paper we propose several parallel algorithms for computing the intersection of sorted arrays, taking advantage of the parallelism provided by the new generation of multicore processors and of the programming library Rapidmind. We perform an experimental comparison of performance on a excerpt of web index and queries provided by Google, and show an improvement of performance proportional to the parallelism when using up to four processors. Our results confirm the intuition that the intersection problem is implicitly parallelisable at the set level, and not only at the query level as previously considered.

Keywords: Conjunctive Queries, Intersection, parallel algorithm, RapidMind, multicore.

1 Introduction

In the past two years, the microchip industry has taken a decided turn towards thread-level parallelism and away from instruction-level parallelism and higher clock frequencies. This means that in the upcoming years, single threaded performance will hit a wall. and new hardware may not provide significantly improved performance for a single thread.

The intersection of large ordered sets is a common problem in the evaluation of queries entered into a search engine. Much research has been done to improve performance in theory in the comparison model [1, 4, 8] or experimentally in a single processor environment [2, 5, 9]. To date, the focus in search engines has been to run multiple queries in parallel efficiently but there has been little work done on optimising the performance of a single query to take advantage of multi-core and multi-processor systems. We show here that the intersection problem lends itself to data parallelism at a lower level than query parallelism.

A naive approach consists in searching each set in parallel for each potential element of the intersection. This type of solution requires a lot of communication between threads, which is too costly to be efficient. We consider another solution which consists in dividing the sets into smaller sets, which are passed on to multiple threads, each of which operates independently to produce the intersection of its small chunk.

While our work is still preliminary, we claim that it raises issues which will prove important in the future: as the Internet grows, and the number of references associated to each keyword grows, it will eventually become impractical to perform an entire set intersection on a single processor without impacting perceivable response time.

The rest of the paper is organised as follows: We describe previous work which we either use or improve upon in Section 2: the architecture (Section 2.1), the problem (Section 2.2) and the programming library (Section 2.3). We describe the algorithms we developed and how we tested them in Section 3, we describe our measures in Section 4, and discuss them more in depth in Section 5. We conclude with a perspective for future work in Section 6.

2 Background

2.1 General Purpose Graphic Processing Units

As graphic processors become cheaper and more powerful, engineers have pushed the concept of general-purpose computation on graphics processors (GPGPU) [13], for diverse applications and in particular for databases [10].

2.2 Conjunctive Queries and Intersection Algorithms

The intersection of large ordered sets is a common problem in the context of the evaluation of relational queries to databases as well as boolean queries to a search engine. The worst case complexity of this problem has long been well understood, dating back to the algorithm by Hwang and Lin from over three decades ago [11, 12], and the average case has been studied in the case of the intersection of two sets, when the elements are uniformly distributed [7].

In 2000, Demaine *et al.* [8] introduced a new intersection algorithm, termed `Adaptive`, which intersects all the sets in parallel to find the shortest proof of the result. In a subsequent study [9], they compared its performance in practice, relative to a straightforward implementation of an intersection algorithm, and proposed a new and better adaptive algorithm which outperformed both in practice. They measured the number of comparisons performed, on the index of a collection of plain text from web pages. In 2002, Barbay and Kenyon [4] introduced another intersection algorithm, which adapts to the correlation between the terms of the query, and one year later Barbay [3] introduced a randomised variant. To the best of our knowledge, neither of these algorithms were implemented before our study. In 2004, Baeza-Yates [1] introduced an intersection algorithm, based on an alternative technique. Baeza-Yates and Salinger [2] measured the performance of the algorithm in terms of CPU time, on pairs of random arrays.

Only a few of those works consider the cpu-time performance, and none of them take advantage of the parallelism available on modern architectures and intrinsic in the intersection problem.

2.3 Rapidmind

RapidMind is macro-programming language library for C++. Commands are recorded and compiled and executed at runtime to the appropriate platform. The RapidMind platform is centred on the notion of data parallelism. It was designed to allow the rapid development of data parallel applications without concern for common concurrency complexities such as mutual exclusion and synchronisation. Under the RapidMind platform, a single "kernel" program is written which performs the same task on multiple pieces of data. Each instance of the kernel program operates in parallel if run on a multi-core or multi-processor system. To avoid race conditions, RapidMind avoids inter-thread communication. Furthermore, RapidMind dedicates only a few areas of memory as writable output for a kernel instance. No two instances share the same output space. This, again, avoids race conditions.

The RapidMind platform is designed to operate on multiple processor architectures. It has support for running data parallel kernel programs on the x86 platform, GPU platform (operating as a GLSL front-end), and the IBM Cell BE platform. When running under the IBM Cell BE platform, current Beta revisions of RapidMind expose some unique architectural features that provide opportunities for significant performance improvement such as a 256KB high-speed local store per Synergistic Processing Unit (SPU) [3]. Running on the PlayStation3 on a 7-core Cell processor, this report demonstrates a preliminary performance evaluation of the Parallel Adaptive algorithm.

3 Experimental set-up

The set intersection problem lends itself to the RapidMind programming model. The proposed Parallel Adaptive algorithm does this by taking the minimum sized set and dividing it up into equal chunks. From there, an instance of the parallel adaptive kernel is created for each chunk of the minimum set. Each kernel instance performs a variation of the sequential Small Adaptive[2] algorithm on the sets. As the output is constrained to a finite space, each kernel outputs a bit array of n bits, where n is the size of the chunk of the minimum set used as input for the kernel instance. Each bit marked in the output indicates whether the given element in the minimum set was found in the intersection set or not, where "1" indicates found and "0" indicates not found.

3.1 Parallel Intersection algorithm

We implemented Algorithm 1, and some variants of it. Sets are input in ascending order of their respective sizes. As with the single-threaded Adaptive and Small Adaptive, Parallel Adaptive begins by picking an eliminator [2]. From there it looks for the eliminator in the next set. If the eliminator is found, the next set becomes the "current" set and the same eliminator is checked in the next set. Unlike Small Adaptive or Adaptive, if not found, a new eliminator is

picked in the next set which is then checked in the current set. If the eliminator is still not found, a new eliminator is picked in the current set and checked in the next set. This process is repeated until a common set element is found between the two sets. At that point we move on to look for the eliminator in the set subsequent to the next set.

Algorithm 1 `Parallel Adaptive Kernel`

Given a set of `numset` sorted arrays along with their sizes, the algorithm (...).

while !done **do**
  e $\leftarrow$ pick eliminator from the set of smallest size;
  i $\leftarrow$ elimset;
  while !done and matches < `numset` **do**
    lastmatches $\leftarrow$ matches;
    curr $\leftarrow (i + 1)$ mod `numset`;
    $l \leftarrow i$;
    while !done and lastmatches == `numset` **do**
      $p \leftarrow$ adaptive search for e in set[curr] (galloping + binary search);
      if $p > sizeofsets[curr]$ **then**
        $done \leftarrow true$;
        break;
      end if
      if found **then**
        matches++;
      else
        $matches \leftarrow 1$;
        $lastmatches \leftarrow 1$;
        $e \leftarrow sets[curr][p]$;
        swap(curr, l);
      end if
    end while
    $i \leftarrow (i + 1)$ mod `numset`;
  end while
  if !done **then**
    output e;
  end if
end while

We study the performance of two variations of Parallel Adaptive. The first variation of Parallel Adaptive performs no pre-processing prior to firing up the kernel program instances. Each instance identifies itself and its segment of the work by a unique identifier that specifies which subset group of the first set to consider. The second variation of Parallel Adaptive divides the entire data set.

3.2 Technical Optimisations

The current implementation of this algorithm takes advantage of some architectural features found in modern systems.

Single Instruction Multiple Data (SIMD) enhancements. One significant feature found in virtually all modern processor architecture is SIMD support [4]. SIMD is a technology that allows a single instruction to act on a single register as if it is a vector of components. Typically, SIMD-enabled architectures support 128-bit registers, each divided into four 32-bit fields (floating point or integer). A single instruction would perform the same operation on all four components simultaneously. Asymptotically, this does nothing, but this provides a notable performance difference in practice. To demonstrate this algorithm in the best light, the algorithm takes advantage of SIMD. SIMD is used in searching (effectively reducing the search space to a quarter the size by evaluating four parallel set elements simultaneously). SIMD is also used to implement a 4-way set associative cache as described in the next section.

4-way set associative cache. The current implementation of the set intersection algorithm runs on the Cell BE microprocessor. The Synergistic Processing Units (SPUs) of the Cell processor contain 256KB of high-speed local store each. This local store does not have any hardware caching mechanism. Instead, it provides an interface for the developer to load and store to and from the local store. For the purpose of this report, a software 4-way set associative cache was developed. Using the SIMD capabilities of the Cell processor, four index slots are checked simultaneously. Each slot corresponds to a 128byte-block. In the current implementation, the cache can load up to 32 blocks. This provides a 4KB cache. Test results have shown there is a two to four fold improvement in performance by using the cache over consistently loading from main memory.

3.3 Experiments

A single un-timed run was first executed to ensure that RapidMind had fully compiled the kernel program code. Subsequently, data was collected by taking the average time of five runs of Parallel Adaptive on the same query. In the pre-processed version of the algorithm, pre-processing time was factored in the total time to execute the algorithm. The algorithm was run on Google data discussed by Barbay *et al.* [5].

4 Results

4.1 Number of processors available

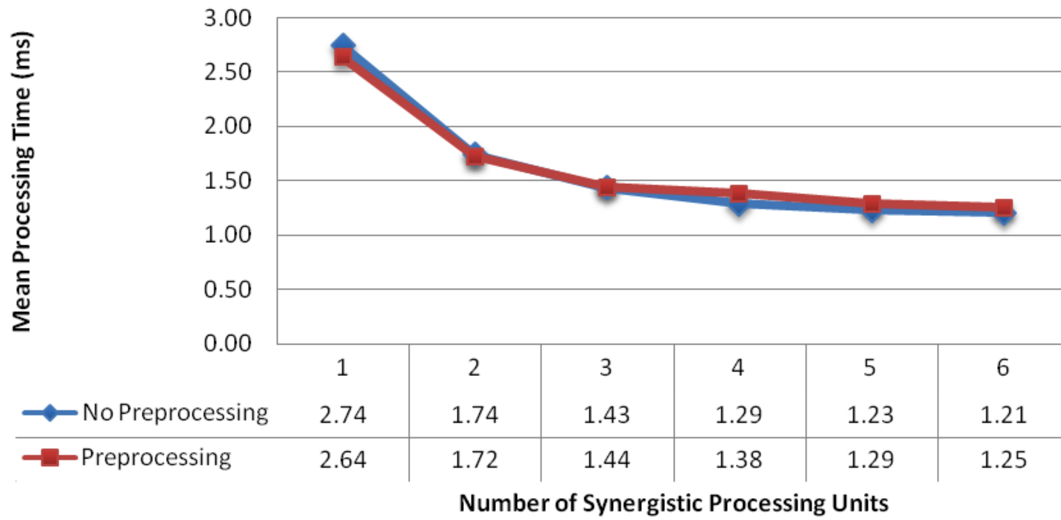
The first graph demonstrates the performance of the two variations of the Parallel Adaptive algorithm discussed earlier. Using the Playstation3 and the RapidMind platform, the number of SPUs enabled was limited to a range of 1 to 6 measuring the performance of all queries on each configuration. An average processing time was taken for each run. The graph demonstrates the average processing time for the "No Pre-processing" version of the algorithm verses the average processing time of the "Pre-processing" version. It is clear here that for one or two processors, the time spent pre-processing data so as to reduce the work done per thread is less than the time wasted to do the same work over and over again per processor. However, as the number of processors increases, threads are spread across more processors and so each processor is doing less and less repetitive work, and so the overhead of the repetitive work also decreases. This is significant because it indicates that if the number of threads is far greater than the number of processors then as the number of processors increases the total overhead per processor decreases to a point where pre-processing of this sort becomes inefficient at best. From the first graph it is clear that beyond two processors, pre-processing does little to improve performance.

A finer analysis of the results show that the algorithm performs even better on *large* instances. There is a large number of queries with very small minimum set sizes, in which the current algorithm does nothing to improve performance when the number of processors increases, as each thread operates on 128 elements from the minimum set. Filtering out queries whose Minimum set contains less than 768 elements (the minimum size required to use all 6 SPUs), the average time for a given number of processors from 1 to 6 shows a much better performance (non-preprocessed version):

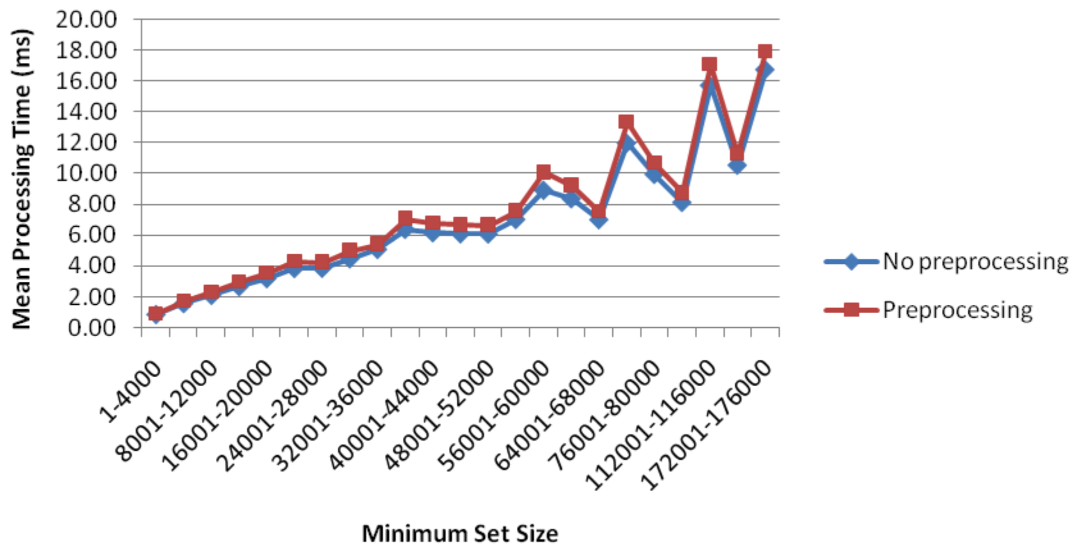
Number of SPUs:	1	2	3	4	5	6
Mean Processing Time(ms):	5.76	3.36	2.59	2.20	2.00	1.88

4.2 Size of smallest set

Next, we consider the processing time of the algorithm as the minimum set size varies. From the second graph, it is clear that there is a general tread towards increasing processing time as the minimum set size increases, which is what was expected. For the algorithm to complete, it must take a search pass at the entire minimum set. Under 6 SPUs, the "No pre-processing" version of the algorithm consistently performs better than the pre-processed version. This graph illustrates a nearly linear relationship between minimum set size and processing time. However, as minimum set size gets large, there are several dips in the graph. As in the Google data set used there are very few queries of that particular size so as the sample size is small, and the accuracy of the data is less reliable.



	1	2	3	4	5	6
No Preprocessing	2.74	1.74	1.43	1.29	1.23	1.21
Preprocessing	2.64	1.72	1.44	1.38	1.29	1.25



4.3 Size of largest set

The third graph illustrates that the processing time varies greatly with maximum set size. There doesn't appear to be any strong correlation between maximum set size and processing time in the Google data set. This is somewhat unexpected. This behaviour suggests that the algorithm isn't consistently hitting the "Extended Search Problem" described in the previous section. This further suggests that there is a near uniform distribution of elements across sets in the queries given. It is not clear where this is true in most data sets in practice. Analysis on other sources of data needs to be done to see whether this behaviour is consistent or not. The variation in certain ranges might be attributed to the limited sample size in those ranges. There is a strong correlation between minimum set size and processing time so if a given range of maximum set size only tests a particular minimum set size or a limited range then the processing time might be skewed as a result. Once again here, pre-processing produces worse results no pre-processing. This suggests that contention and the extended search problem are insignificant here.

4.4 Output size

The fourth graph examines the behaviour of processing time as intersection set size increases. Obviously, there is a strong correlation between intersection set size and minimum set size as intersection set size must be less than or equal to minimum set size. Discarding a few spikes and drops in the higher range, which can be attributed to fewer samples in that range and thus higher variation, there is a roughly linear relationship between intersection set size and processing time as expected.

4.5 Number of sets

In the fifth graph we consider the relationship between the processing time and the number of keys in a query. There seems to be little correlation between the number of keys per query and the mean processing time. The spike in processing time when the number of keys is 10 and 11 can be attributed to there being only one query in the data set with that number of keys in each case. In each of those cases, the minimum set size is quite large, and thus impacting the result here. Thus, if we only consider queries with 2-9 keys, we can see that the number of queries has virtually no effect on processing time. This is a direct consequence of the design of the algorithm. The algorithm as it is now is heavily reliant on the minimum set size. Every time a new eliminator search is initiated, the eliminator is picked from the minimum set. Yet again, the performance without pre-processing is notably better than that without.

5 Discussion

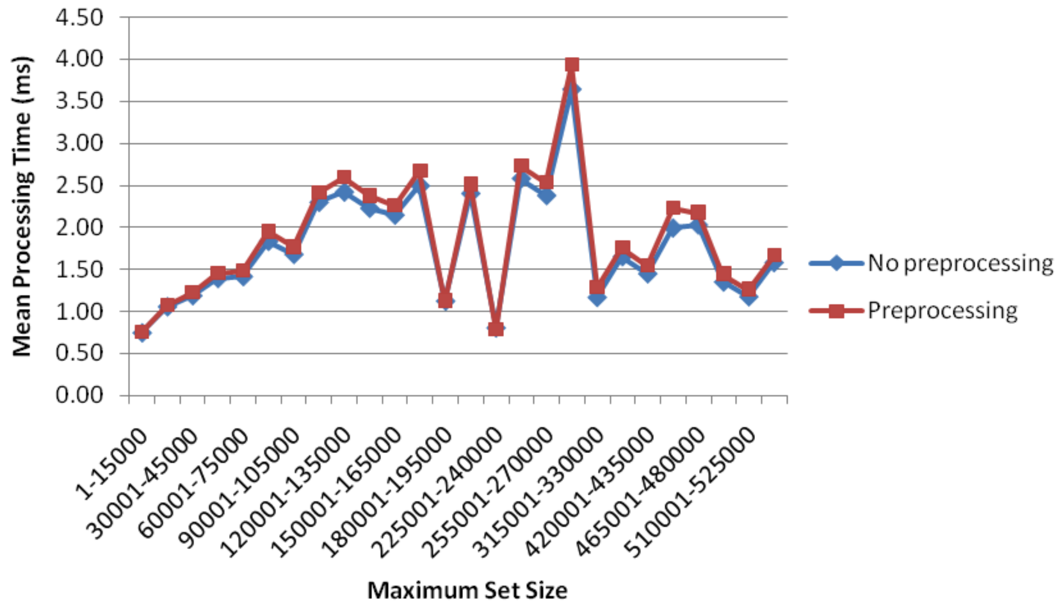
5.1 Main Memory Contention

This issue is more prominent in the first variation of Parallel Adaptive without pre-processing. All threads start their search for eliminators from the beginning of each set. They all try to access the same data in memory (and thus the same memory module) at about the same time. Thus, they all end up sending a read request to the same memory module. As a result, they all stall except the one that got there first as a module can only service one request or at best one row at a time [5]. The pre-processed version of the algorithm provides a slightly better scenario. Every thread has a different starting read address for the sets. This reduces the probability of contention. This does not eliminate the issue however as there is no guarantee each subset will start on a different module or a different row within the same module.

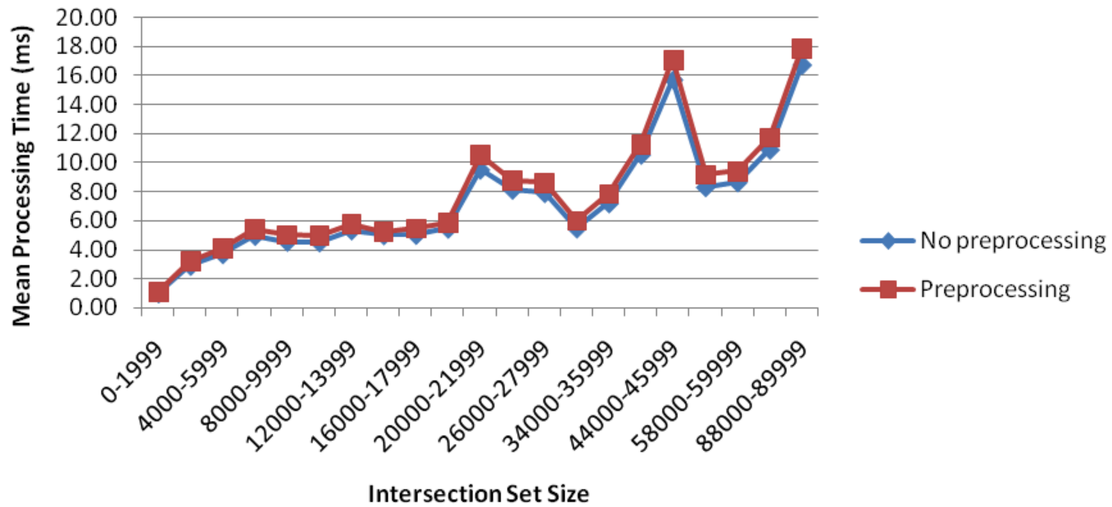
5.2 Extended Search

The inner loop in the algorithm presents a problem in certain data sets. Consider the following scenario:

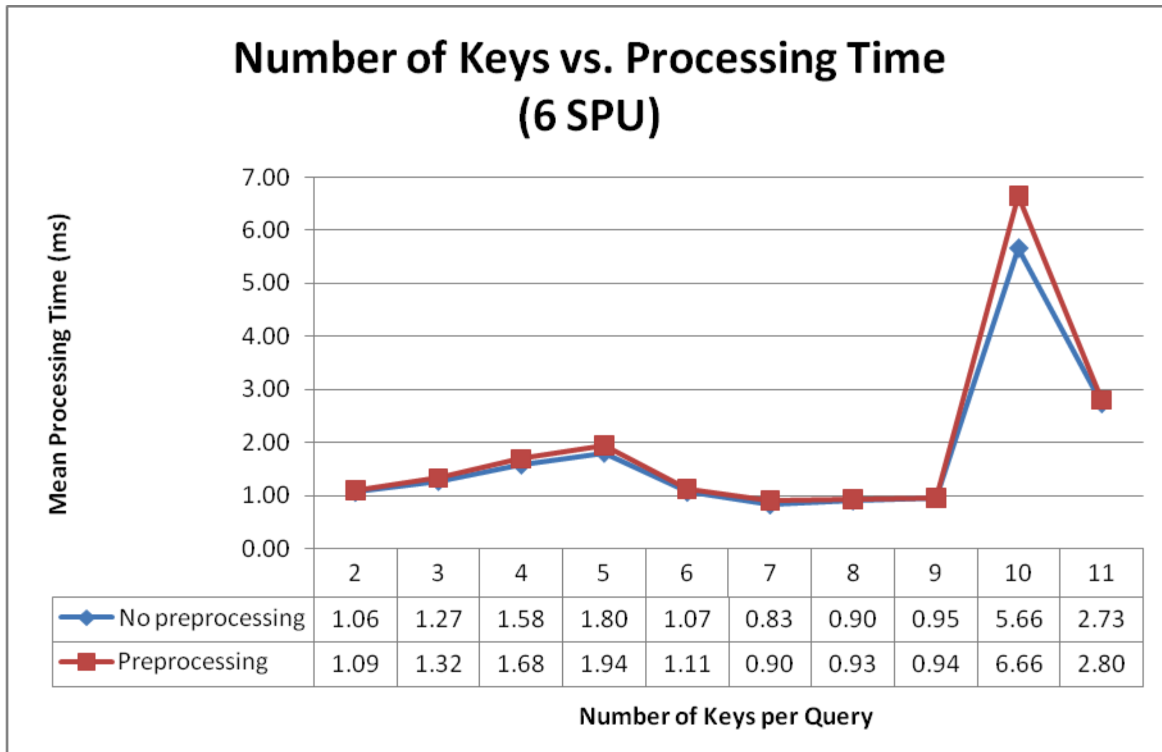
– There are three sets. One is set contains two elements, 1 and 1,000,000. The next set contains all the odd numbers between 1 and 1,000,000 as well as 1 and 1,000,000 (thus the first two sets share 2 elements). The third set contains all the even numbers between 1 and 1,000,000, in addition to 1,000,000. The only element shared across all three sets is 1,000,000.



Maximum Set Size vs. Processing Time
Mean Processing Time (ms)
4.50
4.00
3.50
3.00
2.50
2.00
1.50
1.00
0.50
0.00
1-15000
30001-45000
60001-75000
90001-105000
120001-135000
150001-165000
180001-195000
225001-240000
255001-270000
315001-330000
420001-435000
465001-480000
510001-525000
Maximum Set Size
No preprocessing
Preprocessing



Intersection Set Size vs. Processing Time
(6 SPU)
Mean Processing Time (ms)
20.00
18.00
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
0-1999
4000-5999
8000-9999
12000-13999
16000-17999
20000-21999
26000-27999
34000-35999
44000-45999
58000-59999
88000-89999
Intersection Set Size
No preprocessing
Preprocessing


Number of Keys vs. Processing Time
(6 SPU)
Mean Processing Time (ms)
7.00
6.00
5.00
4.00
3.00
2.00
1.00
0.00
Number of Keys per Query
	2	3	4	5	6	7	8	9	10	11
No preprocessing	1.06	1.27	1.58	1.80	1.07	0.83	0.90	0.95	5.66	2.73
Preprocessing	1.09	1.32	1.68	1.94	1.11	0.90	0.93	0.94	6.66	2.80

– Parallel Adaptive begins by choosing 1 as the eliminator; it searches for 1 in the second set. It finds it, so it searches for 1 in the third set. Of course, it doesn't find it. It picks two in the third set as the new eliminator and searches for it in the second set. Again 2 is not found, so it picks 3 and searches for it and again this is not found so it picks 4 and so on. This loop continues all the way up to 1,000,000.

This behaviour is not ideal because the minimum set has only two elements. The ideal behaviour in this case would be two look for 1 in set 2, see that it is there, then look for it in set 3, see that it is not there, then look for 1,000,000 in set 2, see that it's there and look for 1,000,000 in set 3 and see that it is there and the algorithm is done in a mere four search operations. This is how Small Adaptive behaves. The current version of Parallel Adaptive would be required to perform somewhere on the order of 1,000,000 operations to complete in this particular worst case scenario.

On the other hand, this behaviour was deliberately chosen to improve cache hit rate. If there were a lot of random access to different sets then the probability of a cache hit would be lower. In more typical cases (as the Google data suggests), this behaviour reduces cache misses and provides good performance. However, this "Extended Search" issue should be noted and it would be ideal to address this in future revisions of this algorithm.

5.3 Load Balancing

Parallel Adaptive as it is described here makes no guarantee at load balancing across multiple threads. That is, while the minimum set is divided evenly across all threads, the work required to compute the intersection will vary from thread to thread depending on the distribution of the sets. If all sets are uniformly distributed then the load is balanced, more or less. On the other hand, in a non-uniform distribution, one or more threads might be vulnerable to the Extended Search Problem described above. In the case of the pre-processed version of the algorithm, there is no guarantee that the size of each subset will be the same across threads. Thus, some threads may have to do more work than others. As long as the output size of a kernel program is fixed in size (as restricted by RapidMind and some architectures), I do not foresee there being an easy solution to this issue.

6 Future Work

There is still much work to do before any conclusive statements can be made about the success of data parallel set intersection. In the future, a direct comparison needs to be made between this version of Parallel Adaptive and sequential versions of Small Adaptive, SvS and Adaptive on the same platform, among others. Furthermore, there are many more parameters that need to be varied so as to give a more indicative view where this algorithm might still improvement.

6.1 Cache Analysis

Cache hit rate needs to be recorded. There is a strong correlation between the cache hit rate, and the performance of this algorithm. A high cache hit rate avoids expensive main memory accesses and reduces the effect of the contention problem discussed earlier. There is a very linear flow to this algorithm that allows cache usage to be optimised more than it has currently. Once a search for an eliminator is complete, the algorithm will never attempt to read an address below the current index in the current set. Perhaps a cache enhancement would be to mark cache lines that end before the current index in a given set as available to give room for cache blocks that is still relevant without deleting other relevant cache data. The current implementation employs a First-In-First-Out (FIFO) replacement policy which is not ideal in this situation.

In informal testing while developing the cache, it was noticed that large block sizes (512B) improve performance of complex queries (over 2 keywords, and thousands of elements per set) over small block sizes (128B) while they prove detrimental to performance on simple queries. This can be explained by the skipping of a large number of elements when searching for an eliminator in a given set. With smaller block sizes, the entire search space is not contained on the local store so the algorithm will make multiple DMA transfer requests to main memory which can bog down performance. On simple queries, the cache is loading more memory than it needs to search so that slows down performance.

This cache behaviour needs to be analysed in depth. Perhaps mean processing time can be reduced further if each thread dynamically adjusted its cache behaviour based on a complexity parameter such the average size of the sets it needs to intersect. The current cache implementation is 4-way set associative. In such a cache, a memory address or array index in this case is divided up into three pieces, a tag, a set and a block. The least significant bits identify the block. Following the block bits are the set bits which provide an index into the cache. Finally, the most significant bits uniquely identify the block within a given set. Each set is composed of four slots. The cache knows whether the given block is in store by looking at the tag of each of the blocks in the given set and compares it against the address being requested. In the current implementation, there are 8 sets, and each block is 128-bytes in size. As data is processed in 128-bits at a time, each array index to main memory specifies 128-bits (or 16bytes). Thus 3 bits are associated with set, and 3 bits are associated with block resulting in a 4KB cache. This doesn't even begin to use the full capabilities of the SPU local store cache.

In the future, it might be interesting to observe the behaviour of the cache if 8 or even 10 bits were reserved for set and block resulting in 16-64KB caches. As described above, the cache could be adaptive based on the average size of the sets. Queries with large sets might allocate more bits for block (e.g. 5 or 7-bits) and fewer bits for set. On smaller queries, the cache might allocate fewer bits for block (e.g. 3-bits) and more bits for set.

6.2 Comparison Analysis

The number of comparisons performed needs to be recorded as well for analysis. This can be done as a by-product of recording cache hit rate. Every time an access to data is requested, the cache can increment a counter. This counter effectively counts the number of comparisons performed. This will be extremely useful when analysing the current implementation of Parallel Adaptive with other sequential intersection algorithms. This will also give a quantifiable way to measure the overhead of the non-pre-processing version of Parallel Adaptive in comparison to the pre-processed version and to sequential versions of the algorithm. Furthermore, Demaine et. al [6] have shown an in-depth proof for determining the minimum number of comparisons necessary to find the set intersection. In their paper, Demaine et. al show that various sequential adaptive algorithms are within a constant factor of optimal given a particular instance of a set intersection problem. This proof will be a valuable instrument in demonstrating whether or not Parallel Adaptive can be considered a near optimal algorithm.

6.3 Scalability Analysis

Current testing was confined to 1 to 6 SPUs. This is hardly sufficient to conclude that this algorithm is scalable. Current data suggests that there may be a plateau in performance improvement beyond four processors. It is not clear if and why this occurs at this point. The algorithm might be hitting one of the potential pitfalls discussed early in this report. In the future, testing can be done on up to 16 SPUs on the Cell Blade processor. This will provide a better idea of the scalability of the algorithm. Of course, the number of threads created is a function of the size of the smallest set in the query. If the minimum set is small and there are fewer threads created than there are processors then no performance improvement will be seen. This may be remedied by reducing the input size to each thread and thus require more threads. The current implementation passes 128 elements of the minimum set to each thread to intersect with other threads. This has the side effect of adding more overhead to the algorithm so it will be challenging to find the best compromise between maximising the number of threads and minimising overhead.

6.4 Pre-processed Parallel Adaptive Optimisation

The pre-processed version of Parallel Adaptive presents a venue for further optimisations that was not considered in the current implementation. Previous work by Barbay et. al has shown that the sorting property allows Small Adaptive to outperform other algorithms in many scenarios. In picking a set for the first eliminator in non-pre-processed Parallel Adaptive, the choice is trivial, the minimum set is chosen. All sets are, by definition, larger than the minimum set. As the input sets are sorted by size initially, the algorithm searches for the eliminator in the second smallest set. In the pre-processed version more information is known about the work necessary. The reality is the subsets processed by each thread are not in any particular order. While the eliminator will be picked from the smallest set; the search may not start in the second smallest set. If Parallel Adaptive pre-sorts the sets before beginning, performance may be notably improved.

6.5 B-tree representation of sets

One way to reduce the number of memory accesses per thread is to use a B-tree representation of each of the sets. From there, each search can be done by traversing the B-tree and locating the current eliminator. Search across a B-tree can be done adaptively through the use of "Finger searching". A finger is a reference to an element in the list represented by the B-tree. According to Blelloch *et al.* [6], finger searching matches the O(log n) time bound of a classical search; but in "applications like merging where the locality of reference in the sequence of search targets finger searching yields a significantly tighter time bound." In finger search, the goal is to reduce the complexity of a B-tree tree search from O(log n) to O(log d) where d is defined to be the difference in rank between some element at position i and the search element at position j. In set intersection, eliminators are continuously located and marked, and subsequent searches are performed starting at the successive element to last eliminator position in the case of a successful match or the ending position of a last search in the case of an unsuccessful match. This behaviour makes set intersection a good match for finger searching. Furthermore, Blelloch *et al.*demonstrate a "Hand" structure that enables this type of searching with only O(log n) additional space. Using the Hand structure, the worst-case time complexity of an increment operation is only O(1). The Hand structure works by performing an "eager in-order walk" of the tree as the tree is traversed in search for a given key. The "eagerness" of this algorithm acts as a sort of pre-fetching algorithm for nodes of the B-tree.

Future research into parallel set intersection algorithms should consider B-tree set representations. The "Hand" structure can be implemented to work in conjunction with the current cache implementation. Ideally, a B-tree's node should have a size equal to that of the block size in the cache. As the hand structure is manipulated during search, nodes are pre-fetched, potentially further improving cache hit rate.

7 Conclusion

Preliminary results discussed here show that there is potential for this approach to the set intersection problem. Performance runs have demonstrated good scalability from one to six processors. At this point it is difficult to draw any definitive conclusions without more in-depth scalability testing and direct comparisons against sequential set intersection algorithms. Current data suggests that performance might be hitting a plateau at around four or five processors. Furthermore, this algorithm must be tested against sequential intersection algorithms currently developed. There is no clear indication yet that this algorithm outperforms existing algorithms. Furthermore, a Parallel Small Adaptive algorithm must be implemented in comparison to this algorithm. It is not clear that the current implementation would increase cache hit rate and reduce contention well beyond Parallel Small Adaptive. Regardless, preliminary work has been promising. This algorithm shows merit in terms of scalability. Further research should be conducted in order to produce definitive conclusions.

References

[1] R. A. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 3109 of *LNCS*, pages 400–408. Springer, 2004.

[2] R. A. Baeza-Yates and A. Salinger. Experimental analysis of a fast intersection algorithm for sorted sequences. In *Proceedings of 12th International Conference on String Processing and Information Retrieval (SPIRE)*, pages 13–24, 2005.

[3] J. Barbay. Optimality of randomized algorithms for the intersection problem. In A. Albrecht, editor, *Proceedings of the Symposium on Stochastic Algorithms, Foundations and Applications (SAGA)*, volume 2827 / 2003, pages 26–38. Lecture Notes in Computer Science (LNCS), Springer-Verlag Heidelberg, November 2003.

[4] J. Barbay and C. Kenyon. Adaptive intersection and t-threshold problems. In *Proceedings of the thirteenth ACM-SIAM Symposium On Discrete Algorithms (SODA)*, pages 390–399. ACM-SIAM, ACM, January 2002.

[5] J. Barbay, A. López-Ortiz, and T. Lu. Faster adaptive set intersections for text searching. In M. S. Carme Ãlvarez, editor, *Experimental Algorithms: 5th International Workshop, WEA 2006, Cala Galdana, Menorca, Spain, May 24-27, 2006. Proceedings*, volume 4007 of *Lecture Notes in Computer Science (LNCS)*, pages 146–157. Springer Berlin / Heidelberg, 2006.

[6] G. E. Blelloch, B. M. Maggs, and S. L. M. Woo. Space-efficient finger search on degree-balanced search trees. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 374–383. Society for Industrial and Applied Mathematics, 2003.

[7] W. F. de la Vega, A. M. Frieze, and M. Santha. Average-case analysis of the merging algorithm of hwang and lin. *Algorithmica*, 22(4):483–489, 1998.

[8] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the* 11th *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 743–752, 2000.

[9] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Experiments on adaptive set intersections for text retrieval systems. In *Proceedings of the 3rd Workshop on Algorithm Engineering and Experiments, Lecture Notes in Computer Science*, pages 5–6, Washington DC, January 2001.

[10] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 215–226, New York, NY, USA, 2004. ACM Press.

[11] F. K. Hwang and S. Lin. Optimal merging of 2 elements with n elements. *Acta Informatica*, pages 145–158, 1971.

[12] F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM Journal of Computing*, 1(1):31–39, 1972.

[13] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, Aug. 2005.