Parallel Computation of the Minimal Elements of a Poset

Charles E. Leiserson
Massachusetts Institute of Technology
Cambridge, MA 02139, USA
cel@mit.edu

Liyun Li
University of Western Ontario
London ON, Canada N6A 5B7
lli287@csd.uwo.ca

Marc Moreno Maza
University of Western Ontario
London ON, Canada N6A 5B7
moreno@csd.uwo.ca

Yuzhen Xie
University of Western Ontario
London ON, Canada N6A 5B7
yxie@csd.uwo.ca

ABSTRACT

Computing the minimal elements of a partially ordered finite set (poset) is a fundamental problem in combinatorics with numerous applications such as polynomial expression optimization, transversal hypergraph generation and redundant component removal, to name a few. We propose a divide-and-conquer algorithm which is not only cache-oblivious but also can be parallelized free of data races. We have implemented it in Cilk++ targeting multi-cores. For our test problems of sufficiently large input size our code demonstrates a linear speedup on 32 cores.

Categories and Subject Descriptors

G.4 [**Mathematical Software**]: Parallel and vector implementations; G.2.2 [**Graph Theory**]: Hypergraphs

General Terms

Algorithms, Theory

Keywords

Partial ordering, minimal elements, multithreaded parallelism, Cilk, polynomial evaluation, transversal hypergraph

1. INTRODUCTION

Partially ordered sets arise in many topics of mathematical sciences. Typically, they are one of the underlying algebraic structures of a more complex entity. For instance, a finite collection of algebraic sets $V = \{V_1, \ldots, V_e\}$ (subsets of some affine space $\mathbb{K}^n$ where $\mathbb{K}$ is an algebraically closed field) naturally forms a partially ordered set (poset, for short) for the set-theoretical inclusion. Removing from V any V_i which is contained in some V_j for $i \neq j$ is an important practical question which simply translates to computing the maximal elements of the poset $(V, \subseteq)$. This simple problem is in fact challenging since testing the inclusion $V_i \subseteq V_j$ may require costly algebraic computations. Therefore, one may want to avoid unnecessary inclusion tests by using an efficient algorithm for computing the maximal elements of the poset $(V, \subseteq)$. However, this problem has received little attention in the literature [6] since the questions attached to algebraic sets (like decomposing polynomial systems) are of much more complex nature.

Another important application of the calculation of the minimal elements of a finite poset is the computation of the transversal of a hypergraph [2, 12], which itself has numerous applications, like artificial intelligence [8], computational biology [13], data mining [11], mobile communication systems [23], etc. For a given hypergraph $\mathcal{H}$, with vertex set V, the transversal hypergraph $\mathsf{Tr}(\mathcal{H})$ consists of all minimal transversals of $\mathcal{H}$: a transversal $\mathcal{T}$ is a subset of V having nonempty intersection with every hyperedge of $\mathcal{H}$, and is minimal if no proper subset of $\mathcal{T}$ is a transversal. Articles discussing the computation of transversal hypergraphs, as those discussing the removal of the redundant components of an algebraic set generally take for granted the availability of an efficient routine for computing the maximal (or minimal) elements of a finite poset.

Today's parallel hardware architectures (multi-cores, graphics processing units, etc.) and computer memory hierarchies (from processor registers to hard disks via successive cache memories) enforce revisiting many fundamental algorithms which were often designed with *algebraic complexity* as the main complexity measure and with *sequential running time* as the main performance counter. In the case of the computation of the maximal (or minimal) elements of a poset this is, in fact, almost a first visit. Up to our knowledge, there is no published papers dedicated to a general algorithm solving this question. The procedure analyzed in [18] is specialized to posets that are cartesian products of totally ordered sets.

In this article, we propose an algorithm for computing the minimal elements of an arbitrary finite poset. Our motivation is to obtain an efficient implementation in terms of parallelism and data locality. This divide-and-conquer algorithm, presented in Section 2, follows the cache-oblivious philosophy introduced in [9]. Referring to the *multithreaded fork-join parallelism* model of Cilk [10], our algorithm has work $O(n^2)$ and span (or critical path length) $O(n)$, counting the number of comparisons, on an input poset of n elements. An algorithmic solution with a span of $O(\log(n))$ can be achieved in principle. However, we do not know how to derive from it a program that would be free of mutual exclusion mechanisms. Our algorithm does not suffer from data races and can be implemented in Cilk with `sync` as the only mutex. In addition, our experimental results show that our code can reach linear speedup on 32 cores for n large enough.

In several applications, the poset is so large that it is desirable to compute its minimal (or maximal) elements concurrently to the generation of the poset itself, thus avoiding storing the entire poset in memory. We illustrate this strategy with two applications: poly-

PASCO 2010, 21–23 July 2010, Grenoble, France.

nomial expression optimization in Section 4 and transversal hypergraph generation in Section 5. In each case, we generate the poset in a divide-and-conquer manner and at the same time we compute its minimal elements. Since, for these two applications, the number of minimal elements is in general much smaller than the poset cardinality, this strategy turns out to be very effective and allows computations that could not be conducted otherwise.

This article is dedicated to Claude Berge (1926 - 2002) who introduced the third author to the combinatorics of sets.

2. THE ALGORITHM

We start by reviewing the notion of a partially ordered set. Let $\mathcal{X}$ be a set and $\preceq$ be a partial order on $\mathcal{X}$, that is, a binary relation on $\mathcal{X}$ which is reflexive, antisymmetric, and transitive. The pair $(\mathcal{X}, \preceq)$ is called a *partially ordered set*, or poset for short. If A is a subset of $\mathcal{X}$, then $(A, \preceq)$ is the poset induced by $(\mathcal{X}, \preceq)$ on A. When clear from context, we will often write A instead of $(A, \preceq)$. Here are a few examples of posets:

1. $(\mathbb{Z}, |)$ where $|$ is the divisibility relation in the ring $\mathbb{Z}$ of integer numbers,
2. $(2^S, \subseteq)$ where $\subseteq$ is the inclusion relation in the ranked lattice of all subsets of a given finite set S,
3. $(\mathcal{C}, \subseteq)$ where $\subseteq$ is the inclusion relation for the set $\mathcal{C}$ of all algebraic curves in the affine space of dimension 2 over the field of complex numbers.

An element $x \in \mathcal{X}$ is *minimal* for $(\mathcal{X}, \preceq)$ if for all $y \in \mathcal{X}$ we have: $y \preceq x \Rightarrow y = x$. The set of the elements $x \in \mathcal{X}$ which are minimal for $(\mathcal{X}, \preceq)$ is denoted by $\mathsf{Min}(\mathcal{X}, \preceq)$, or simply $\mathsf{Min}(\mathcal{X})$. From now on we assume that $\mathcal{X}$ is finite.

Algorithms 1 and 2 compute $\mathsf{Min}(\mathcal{X})$ respectively in a sequential and parallel fashion. Before describing these algorithms in more details let us first specify the programming model and data-structures. We adopt the multi-threaded programming model of Cilk [10]. In our pseudo-code, the keywords **spawn** and **sync** have the same semantics as the **cilk_spawn** and **cilk_sync** in the `Cilk++` programming language [15]. We assume that the subsets of $\mathcal{X}$ are implemented by a data-structure which supports the following operations for any subsets A, B of $\mathcal{X}$:

Split: if $|A| \geq 2$ then $\mathsf{Split}(A)$ returns a partition A^-, A^+ of A such that $|A^-|$ and $|A^+|$ differ at most by 1.

Union: $\mathsf{Union}(A, B)$ accepts two disjoint sets A, B and returns C where $C = A \cup B$;

In addition, we assume that each subset A of $\mathcal{X}$, with $k = |A|$, is encoded in a *C/C++ fashion* by an array A of size $\ell \geq k$. An element in A can be marked at trivial cost.

In Algorithm 1, this data-structure supports a straight-forward sequential implementation of the computation of $\mathsf{Min}(A)$, which follows from this trivial observation: an element $a_i \in A$ is minimal for $\preceq$ if for all $j \neq i$ the relation $a_j \preceq a_i$ does not hold. However, due to the dependencies between iterations of the inner loop, Algorithm 1 can not be parallelized without using locks. In addition, and unless the the input data fits in cache, Algorithm 1 is not cache-efficient. We shall return to this point in Section 3.

Algorithm 2 follows the cache-oblivious philosophy introduced in [9]. More precisely, and similarly to the matrix multiplication algorithm of [9], Algorithm 2 proceeds in a divide-and-conquer fashion such that when a subproblem fits into the cache, then all subsequent computations can be performed with no further cache misses. However, Algorithm 2, and other algorithms in this paper, use a threshold such that, when the size of the input is within this threshold, then a base case subroutine is called. In principle, this threshold can be set to the smallest meaningful value, say 1, and thus Algorithm 2 is cache-oblivious. In a software implementation, this threshold should be large enough so as to reduce parallelization overheads and recursive call overheads. Meanwhile, this threshold should be small enough in order to guarantee that, in the base case, cache misses are limited to cold misses. In the implementation of the matrix multiplication algorithm of [9], available in the `Cilk++` distribution, a threshold is used for the same purpose.

Algorithm 1: SerialMinPoset

```
Input  : a poset A
Output : Min(A)
1  for i from 0 to |A|−2 do
2      if a_i is unmarked then
3          for j from i+1 to |A|−1 do
4              if a_j is unmarked then
5                  if a_j ⪯ a_i then
6                      mark a_i and break inner loop;
7                  if a_i ⪯ a_j then
8                      mark a_j;
9  A ← {unmarked elements in A};
10 return A;
```

Algorithm 2: ParallelMinPoset

```
Input  : a poset A
Output : Min(A)
1  if |A| ≤ MIN_BASE then
2      return SerialMinPoset(A);
3  (A⁻, A⁺) ← Split(A);
4  A⁻ ← spawn ParallelMinPoset(A⁻);
5  A⁺ ← spawn ParallelMinPoset(A⁺);
6  sync;
7  (A⁻, A⁺) ← ParallelMinMerge(A⁻, A⁺);
8  return Union(A⁻, A⁺);
```

In Algorithm 2, when $|A| \leq \mathsf{MIN\_BASE}$, where $\mathsf{MIN\_BASE}$ is the threshold, Algorithm 1 is called. Otherwise, we partition A into a balanced pair of subsets A^-, A^+. By balanced pair, we mean that the cardinalities $|A^-|$ and $|A^+|$ differ at most by 1. The two recursive calls on A^- and A^+ in Lines 4 and 5 of Algorithm 2 will compare the elements in A^- and A^+ separately. Thus, they can be executed in parallel and free of data races. In Lines 4 and 5 we overwrite each input subset with the corresponding output one so that at Line 6 we have $A^- = \mathsf{Min}(A^-)$ and $A^+ = \mathsf{Min}(A^+)$. Line 6 is a synchronization point which ensures that the computations in Lines 4 and 5 must complete before starting the procedure in Line 7. Next we make cross-comparisons between A^- and A^+, by means of the operation $\mathsf{ParallelMinMerge}$ of Algorithm 3.

We also apply a divide-and-conquer-with-threshold strategy for the operation $\mathsf{ParallelMinMerge}$ in Algorithm 3, which takes as input a pair B, C of subsets of $\mathcal{X}$, such that $\mathsf{Min}(B) = B$ and $\mathsf{Min}(C) = C$ hold. Note that this pair is not necessarily balanced. This leads to the following four cases in Algorithm 3, depending on the values of $|B|$ and $|C|$ w.r.t. the threshold $\mathsf{MIN\_MERGE\_BASE}$.

Case 1: both $|B|$ and $|C|$ are no more than MIN_MERGE_BASE. We simply call the operation SerialMinMerge of Algorithm 4 which cross-compares the elements of B and C in order to remove the larger ones in each of B and C. The minimal elements from B and C are stored separately in an ordered pair (the same order as the input) to remember the provenance of each result. In Cases 2, 3 and 4, this output specification helps clarifying the successive cross-comparisons when the input posets are divided into subsets.

Algorithm 3: ParallelMinMerge

```
   Input  : B, C such that Min(B) = B and
            Min(C) = C hold
   Output : (E, F) such that E ∪ F = Min(B ∪ C),
            E ⊆ B and F ⊆ C hold
 1 if |B| ≤ MIN_MERGE_BASE and
 2    |C| ≤ MIN_MERGE_BASE then
 3   return SerialMinMerge(B, C);
 4 else if |B| > MIN_MERGE_BASE and
 5        |C| > MIN_MERGE_BASE then
 6   (B⁻, B⁺) ← Split(B);
 7   (C⁻, C⁺) ← Split(C);
 8   (B⁻, C⁻) ← spawn
 9              ParallelMinMerge(B⁻, C⁻);
10   (B⁺, C⁺) ← spawn
11              ParallelMinMerge(B⁺, C⁺);
12   sync;
13   (B⁻, C⁺) ← spawn
14              ParallelMinMerge(B⁻, C⁺);
15   (B⁺, C⁻) ← spawn
16              ParallelMinMerge(B⁺, C⁻);
17   sync;
18   return (Union(B⁻, B⁺), Union(C⁻, C⁺));
19 else if |B| > MIN_MERGE_BASE and
20        |C| ≤ MIN_MERGE_BASE then
21   (B⁻, B⁺) ← Split(B);
22   (B⁻, C) ← ParallelMinMerge(B⁻, C);
23   (B⁺, C) ← ParallelMinMerge(B⁺, C);
24   return (Union(B⁻, B⁺), C);
25 else
     // |B| ≤ MIN_MERGE_BASE and
     // |C| > MIN_MERGE_BASE
26   (C⁻, C⁺) ← Split(C);
27   (B, C⁻) ← ParallelMinMerge(B, C⁻);
28   (B, C⁺) ← ParallelMinMerge(B, C⁺);
29   return (B, Union(C⁻, C⁺));
```

Case 2: both $|B|$ and $|C|$ are greater than MIN_MERGE_BASE. We split B and C into balanced pairs of subsets B^-, B^+ and C^-, C^+ respectively. Then, we recursively merge these 4 subsets, as described in Lines 8–14 in Algorithm 3. Merging B^-, C^- and merging B^+, C^+ can be executed in parallel without data races. These two computations complete half of the cross-comparisons between B and C. Then, we perform the other half of the cross-comparisons between B and C by merging B^-, C^+ and merging B^+, C^- in parallel. At the end, we return the union of the subsets from B and the union of the subsets from C.

Algorithm 4: SerialMinMerge

```
   Input  : B, C such that Min(B) = B and
            Min(C) = C hold
   Output : (E, F) such that E ∪ F = Min(B ∪ C),
            E ⊆ B and F ⊆ C hold
 1 if |B| = 0 or |C| = 0 then
 2   return (B, C);
 3 else
 4   for i from 0 to |B|−1 do
 5     for j from 0 to |C|−1 do
 6       if c_j is unmarked then
 7         if c_j ⪯ b_i then
 8           mark b_i and break inner loop;
 9         if b_i ⪯ c_j then
10           mark c_j;
11   B ← {unmarked elements in B};
12   C ← {unmarked elements in C};
13   return (B, C);
```

Case 3, 4: either $|B|$ or $|C|$ is greater than MIN_MERGE_BASE, but not both. Here, we split the larger set into two subsets and make the appropriate cross-comparisons via two recursive calls, see Lines 15–25 in Algorithm 3.

3. COMPLEXITY ANALYSIS AND EXPERIMENTATION

We shall establish a worst case complexity for the work, the span and the cache complexity of Algorithm 2. More precisely, we assume that the input poset of this algorithm has $n \geq 1$ elements, which are pairwise incomparable for $\preceq$, that is, neither $x \preceq y$ nor $y \preceq x$ holds for all $x \neq y$. Our running time is estimated by counting the number of comparisons, that is, the number of times that the operation $\preceq$ is invoked. The costs of all other operations are neglected. The principle of Algorithm 2 is similar to that of a parallel merge-sort algorithm with a parallel merge subroutine, which might suggest that the analysis is standard. The use of thresholds requires, however, a bit of care.

We introduce some notations. For Algorithms 1 and 2 the size of the input is $|A|$ whereas for Algorithms 3 and 4 the size of the input is $|B| + |C|$. We denote by $W_1(n)$, $W_2(n)$, $W_3(n)$ and $W_4(n)$ the work of Algorithms 1, 2, 3 and 4, respectively, on an input of size n. Similarly, we denote by $S_1(n)$, $S_2(n)$, $S_3(n)$ and $S_4(n)$ the span of Algorithms 1, 2, 3 and 4, respectively, on an input of size n. Finally, we denote by N_2 and N_3 the thresholds MIN_BASE and MIN_MERGE_BASE, respectively.

Since Algorithm 4 is sequential and under our worst case assumption, we clearly have $W_4(n) = S_4(n) = \Theta(n^2)$. Similarly, we have $W_1(n) = S_1(n) = \Theta(n^2)$.

Observe that, under worst case assumption, the cardinalities of the input sets B, C differ at most by 1 in both Algorithms 3 and 4. Hence, the work of Algorithm 3 satisfies:

$$W_3(n) = \begin{cases} W_4(n) & \text{if } n \leq N_3 \\ 4W_3(n/2) & \text{otherwise.} \end{cases}$$

This implies: $W_3(n) \leq 4^{\log_2(n/N_3)} N_3^2$ for all n. Thus we have $W_3(n) = O(n^2)$. On the other hand, our assumption implies that every element of B needs to be compared with every element of C.

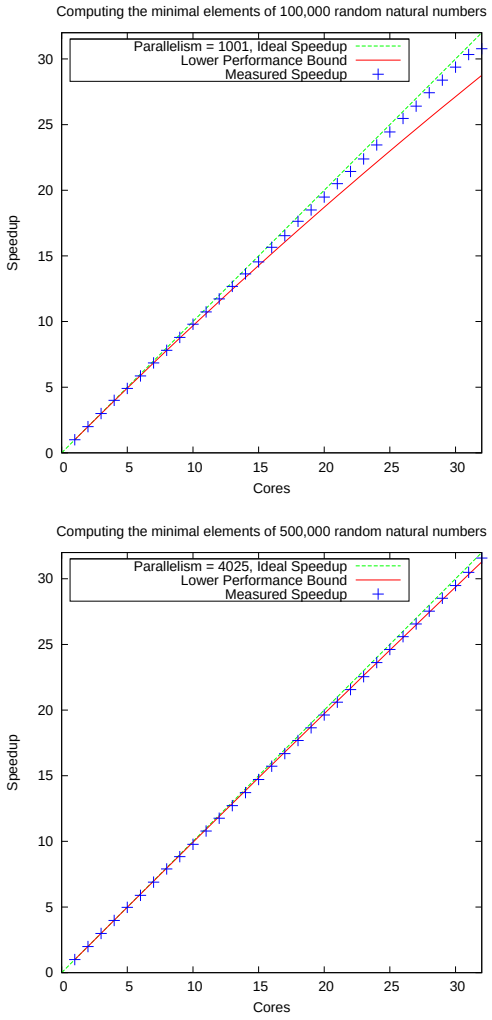

Figure 1: Scalability analysis for ParallelMinPoset **by Cilkview**

Therefore $W_3(n) = \Theta(n^2)$ holds. Now, the span satisfies:

$$S_3(n) = \begin{cases} S_4(n) & \text{if } n \leq N_3 \\ 2S_3(n/2) & \text{otherwise.} \end{cases}$$

This implies: $S_3(n) \leq 2^{\log_2(n/N_3)} N_3^2$ for all n. Thus we have $S_3(n) = O(nN_3)$. Moreover, $S_3(n) = \Theta(n)$ holds for $N_3 = 1$.

Next, the work of Algorithm 2 satisfies:

$$W_2(n) = \begin{cases} W_1(n) & \text{if } n \leq N_2 \\ 2W_2(n/2) + W_3(n) & \text{otherwise.} \end{cases}$$

This implies: $W_2(n) \leq 2^{\log_2(n/N_2)} N_2^2 + \Theta(n^2)$ for all n. Thus we have $W_2(n) = O(nN_2) + \Theta(n^2)$.

Finally, the span of Algorithm 2 satisfies:

$$S_2(n) = \begin{cases} S_1(n) & \text{if } n \leq N_2 \\ S_2(n/2) + S_3(n) & \text{otherwise.} \end{cases}$$

Thus we have $S_2(n) = O(N_2^2 + nN_3)$. Moreover, for $N_3 = N_2 = 1$, we have $S_2(n) = \Theta(n)$.

We proceed now with cache complexity analysis, using the ideal cache model of [9]. We consider a cache of Z words where each cache line has L words. For simplicity, we assume that the elements of a given poset are packed in an array, occupying consecutive slots, each of size 1 word. We focus on Algorithms 2 and 3, denoting by $Q_2(n)$ and $Q_3(n)$ the number of cache misses that they incur respectively on an input data of size n. We assume that the thresholds in Algorithms 2 and 3 are set to 1. Indeed, Algorithms 1 and 4 are not cache-efficient. Both may incur $\Theta(n^2/L)$ cache misses, for n large enough, whereas $Q_2(n) \in O(n/L + n^2/(ZL))$ and $Q_3(n) \in O(n^2/(ZL))$ hold, as we shall prove now. Observe first that there exist positive constants α_2 and α_3 such that we have:

$$Q_2(n) = \begin{cases} \Theta(n/L+1) & \text{if } n \leq \alpha_2 Z \\ 2Q_2(n/2) + Q_3(n) + \Theta(1) & \text{otherwise.} \end{cases}$$

and:

$$Q_3(n) = \begin{cases} \Theta(n/L+1) & \text{if } n \leq \alpha_3 Z \\ 4Q_3(n/2) + \Theta(1) & \text{otherwise.} \end{cases}$$

This implies: $Q_3(n) \leq 4^{\log_2(n/(\alpha_3 Z))}\Theta(Z/L)$ for all n, since $Z \in \Omega(L^2)$ holds. Thus we have $Q_3(n) \in O(n^2/(ZL))$. We deduce:

$$Q_2(n) \leq 2^k \Theta(Z/L) + \sum_{i=0}^{i=k-1} 2^i Q_3(n/2^i) + \Theta(2^k)$$

where $k = \log_2(n/(\alpha_2 Z))$. This leads to: $Q_2(n) \leq O(n/L + n^2/(ZL))$. Therefore, we have proved the following result.

PROPOSITION 1. *Assume that $\mathcal{X}$ has $n \geq 1$ elements, such that neither $x \preceq y$ nor $y \preceq x$ holds for all $x, y \in \mathcal{X}$. Set the thresholds in Algorithms 2 and 3 to 1. Then, the work, span and cache complexity of* Min$(\mathcal{X})$*, as computed by Algorithm 2, are $\Theta(n^2)$, $\Theta(n)$ and $O(n/L + n^2/(ZL))$, respectively.*

We leave for a forthcoming paper other types of analysis such as average case algebraic complexity. We turn now our attention to experimentation.

We have implemented the operation ParallelMinPoset of Algorithm 2 as a template function in Cilk++. It is designed to work for any poset providing that a method Compare(a_i, a_j) for determining any two elements a_i and a_j whether $a_j \preceq a_i$, or $a_i \preceq a_j$, or a_i and a_j are incomparable. Our code offers two data structures for encoding the subsets of the poset $\mathcal{X}$: one is based on arrays and the other uses the *bag structure* introduced by the first Author in [20].

For the benchmarks reported in this section, $\mathcal{X}$ is a finite set of natural numbers compared for the divisibility relation. For example, the set of the minimal elements of $\mathcal{X} = \{6, 2, 7, 3, 5, 8\}$ is $\{2, 7, 3, 5\}$. Clearly, we implement natural numbers using the type `int` of C/C++. Since checking of integer divisibility is cheap, we expect that these benchmarks could illustrate the intrinsic parallel efficiency of our algorithm.

We have benchmarked our program on sets of random natural numbers, with sizes ranging from $50,000$ to $500,000$ on a 32-core machine at SHARCNET [1].This machine has 8 Quad Core AMD Opteron 8354 @ 2.2 GHz connected by 8 sockets. Each core has 64 KB L1 data cache and 512 KB L2 cache. Every four cores share 2 MB of L3 cache. The total memory is 128.0 GB. We have compared the timings with MIN_BASE and MIN_MERGE_BASE being $8, 16, 32, 64$ and 128 for different sizes of input. As a result, we choose 64 for both MIN_BASE and MIN_MERGE_BASE to reach the best timing for all the test cases.

Figure 1 shows the results measured by the Cilkview [14] scalability analyzer for computing the minimal elements of $100,000$ and $500,000$ random natural numbers. The reference sequential algorithm for the speedup is Algorithm 2 running on 1 core; the running time of this latter code differs only by 0.5% or 1% from the C elision of Algorithm 2. On 1 core, the timing for computing the

[1] `http://www.sharcnet.ca`

minimal elements of $100,000$ and $500,000$ random natural numbers is respectively 260 and 6454 seconds, which is slightly better (0.5%) than Algorithm 1. The number of minimal elements for the two sets of random natural numbers is respectively $99,919$ and $498,589$. These results demonstrate the abundant parallelism created by our divide-and-conquer algorithm and the very low parallel overhead of our program in Cilk++. We have also used Cilkview to check that our program is indeed free of data races.

4. POLYNOMIAL EXPRESSION OPTIMIZATION

We present an application where the poset can be so large that it is desirable to compute its minimal elements concurrently to the generation of the poset itself, thus avoiding storing the entire poset in memory. As we shall see this approach is very successful for this application

We briefly describe this application which arises in the optimization of polynomial expressions. Let $f \in \mathbb{K}[X]$ be a multivariate polynomial with coefficients in a field $\mathbb{K}$ and with variables in $X = \{x_1, \ldots, x_n\}$. We assume that f is given as the sum of its terms, say $f = \sum_{m \in \mathsf{monoms}(f)} c_m\, m$, where $\mathsf{monoms}(f)$ denotes the set of the monomials of f and c_m is the coefficient of m in f.

A key procedure in this application computes a *partial syntactic factorization* of f, that is, three polynomials $g, h, r \in \mathbb{K}[X]$, such that f writes $gh + r$ and the following two properties hold: (1) every term in the product gh is the product of a term of g and a term of h, (2) the polynomials r and gh have no common monomials. It is easy to see that if both g and h are not constant and one has at least two terms, then evaluating f represented as $gh + r$ requires less additions and multiplications in $\mathbb{K}$ than evaluating f represented as $\sum_{m \in \mathsf{monoms}(f)} c_m\, m$, that is, as the sum of its terms. Consider for instance the polynomial $f = ax + ay + az + by + bz \in \mathbb{Q}[x, y, z, a, b]$. One possible partial syntactic factorization is $(g, h, r) = (a+b, y+z, ax)$ since we have $f = (a+b)(y+z)+ax$ and since the above two properties are clearly satisfied. Evaluating f after specializing x, y, z, a, b to numerical values will amount to 9 additions and multiplications in $\mathbb{Q}$ with $f = ax+ay+az+by+bz$ while 5 are sufficient with $f = (a + b)(y + z) + ax$.

One economic and popular approach to reduce the size of polynomial expression and facilitate their evaluation is to use Horner's rule. This high-school trick well-known for univariate polynomials is extended to multivariate polynomials via different schemes [4, 21, 22, 5]. However, it is difficult to compare these extensions and obtain an optimal scheme from any of them. Indeed, they all rely on selecting an appropriate ordering of the variables. Unfortunately, there are $n!$ possible orderings for n variables, which limits this approach to polynomials with moderate number of variables.

In [19], given a finite set $\mathcal{M}$ of monomials in $x_1, \ldots, x_n$, the authors propose an algorithm for computing a partial syntactic factorization (g, h, r) of f such that $\mathsf{monoms}(g) \subseteq \mathcal{M}$ holds. The complexity of this algorithm is polynomial in $|\mathcal{M}|, n, d, t$ where d and t are the total degree and number of terms of f, respectively. One possible choice for $\mathcal{M}$ would consist in taking all monomials dividing a term in f. The resulting *base monomial set* $\mathcal{M}$ would often be too large since the targeted n and d in practice are respectively in the ranges $4 \cdots 16$ and $2 \cdots 10$, which would lead $|\mathcal{M}|$ to be in the order of thousands or even millions. In [19], the set $\mathcal{M}$ is computed in the following way:

1. compute $\mathcal{G}$ the set of all non-constant $\gcd(m_1, m_2)$ where m_1, m_2 are any two monomials of f, with $m_1 \neq m_2$,
2. compute the minimal elements of $\mathcal{G}$ for the divisibility relation of monomials.

In practice, this strategy produces a more efficient evaluation representation comparing to the Horner's rule based polynomial expression optimization methods. However, there is an implementation challenge. Indeed, in practice, the number of terms of f is often in the thousands, which implies that $|\mathcal{G}|$ could be in the millions.

This has led to the design of a procedure presented through Algorithms 5, 6, 7 and 8, where $\mathcal{G}$ and $\mathsf{Min}(\mathcal{G})$ are computed concurrently in a way that the whole set $\mathcal{G}$ does not need to be stored. The proposed procedure is adapted from Algorithms 1, 2, 3 and 4. The top-level routine is Algorithm 5 which takes as input a set A of monomials. In practice one would first call this routine with $A = \mathsf{monoms}(f)$. Algorithm 5 integrates the computation of $\mathcal{G}$ and $\mathcal{M}$ (as defined above) into a "single pass" divide-and-conquer process. In Algorithms 5, 6, 7 and 8, we assume that monomials support the operations listed below, where m_1, m_2 are monomials:

- $\mathsf{Compare}(m_1, m_2)$ returns 1 if m_1 divides m_2 (that is, if m_2 is a multiple of m_1) and returns -1 if m_2 divides m_1. Otherwise, m_1 and m_2 are incomparable. This function implements the partial order used on the monomials.
- $\mathsf{gcd}(m_1, m_2)$ computes the gcd of m_1 and m_2.

In addition, we have a data-structure for monomial sets which support the following operations, where A, B are monomial sets.

- $\mathsf{innerPairsGcds}(A)$ computes $\mathsf{gcd}(a_1, a_2)$ for all $a_1, a_2 \in A$ where $a_1 \neq a_2$ and returns the non-constant values only.
- $\mathsf{crossPairsGcds}(A, B)$ computes $\mathsf{gcd}(a, b)$ for all $a \in A$ and for all $b \in B$ and returns the non-constant values only.
- $\mathsf{SerialInnerBaseMonomials}(A)$ first calls $\mathsf{innerPairsGcds}(A)$, and then passes the result to SerialMinPoset of Algorithm 1.
- $\mathsf{SerialCrossBaseMonomials}(A, B)$ applies SerialMinPoset to the result of $\mathsf{crossPairsGcds}(A, B)$.

With the above basic operations, we can now describe our divide-and-conquer method for computing the base monomial set of A, that is, $\mathsf{Min}(\mathcal{G}_A)$, where $\mathcal{G}_A$ consists of all non-constant $\gcd(a_1, a_2)$ for $a_1, a_2 \in A$ and $a_1 \neq a_2$. The top-level function is ParallelBaseMonomial of Algorithm 5. If $|A|$ is within a threshold, namely MIN_BASE, the operation $\mathsf{SerialInnerBaseMonomials}(A)$ is called. Otherwise, we partition A as $A^- \cup A^+$ and observe that

$$\mathsf{Min}(\mathcal{G}_A) = \mathsf{Min}(\mathsf{Min}(\mathcal{G}_{A^-}) \cup \mathsf{Min}(\mathcal{G}_{A^+}) \cup \mathsf{Min}(\mathcal{G}_{A^-,A^+}))$$

holds where $\mathcal{G}_{A^-,A^+}$ consists of all non-constant $\gcd(x, y)$ for $(x, y) \in A^- \times A^+$. Following the above formula, we create two computational branches: (1) one for $\mathsf{Min}(\mathsf{Min}(\mathcal{G}_{A^-}) \cup \mathsf{Min}(\mathcal{G}_{A^+}))$ which is computed by the operation SelfBaseMonomials of Algorithm 6; (2) one for $\mathsf{Min}(\mathcal{G}_{A^-,A^+})$ which is computed by the operation CrossBaseMonomials of Algorithm 7. Algorithms 6 and 7 proceed in a divide-and-conquer manner:

- Algorithm 6 makes two recursive calls in parallel, then merges their results with Algorithm 3.
- Algorithm 7 uses a threshold. In the base case, computations are performed serially. Otherwise, both input monomial sets are split evenly then processed via two concurrent calls to Algorithm 8, whose results are merged with Algorithm 3.
- Algorithm 8 simply performs two concurrent calls to Algorithm 7 whose results are merged with Algorithm 3.

Algorithm 5: ParallelBaseMonomials

Input : a monomial set A
Output: $\mathsf{Min}(\mathcal{G}_A)$ where $\mathcal{G}_A$ consists of all non-constant $\gcd(a_1, a_2)$ for $a_1, a_2 \in A$ and $a_1 \neq a_2$

1 **if** $|A| \leq$ MIN_BASE **then**
2 **return** SerialInnerBaseMonomials(A);
3 **else**
4 $(A^-, A^+) \leftarrow \mathsf{Split}(A)$;
5 $B \leftarrow$ **spawn** SelfBaseMonomials(A^-, A^+);
6 $C \leftarrow$ **spawn** CrossBaseMonomials(A^-, A^+);
7 **sync**;
8 $(D_1, D_2) \leftarrow \mathsf{ParallelMinMerge}(B, C)$;
9 **return** $\mathsf{Union}(D_1, D_2)$;

Algorithm 6: SelfBaseMonomials

Input : two disjoint monomial sets B, C
Output: $\mathsf{Min}(\mathcal{G}_B \cup \mathcal{G}_C)$ where $\mathcal{G}_B$ (resp. $\mathcal{G}_C$) consists of all non-constant $\gcd(x, y)$ for $x, y \in B$ (resp. C) with $x \neq y$

1 $E \leftarrow$ **spawn** ParallelBaseMonomials(B);
2 $F \leftarrow$ **spawn** ParallelBaseMonomials(C);
3 **sync**;
4 $(D_1, D_2) \leftarrow \mathsf{ParallelMinMerge}(E, F)$;
5 **return** $\mathsf{Union}(D_1, D_2)$;

Algorithm 7: CrossBaseMonomials

Input : two disjoint monomial sets B, C
Output: $\mathsf{Min}(\mathcal{G}_{B,C})$ where $\mathcal{G}_{B,C}$ consists of all non-constant $\gcd(b, c)$ for $(b, c) \in B \times C$

1 **if** $|B| \leq$ MIN_MERGE_BASE **then**
2 **return** SerialCrossBaseMonomials(B, C);
3 **else**
4 $(B^-, B^+) \leftarrow \mathsf{Split}(B)$;
5 $(C^-, C^+) \leftarrow \mathsf{Split}(C)$;
6 $E \leftarrow$ **spawn**
7 HalfCrossBaseMonomials(B^-, C^-, B^+, C^+);
8 $F \leftarrow$ **spawn**
9 HalfCrossBaseMonomials(B^-, C^+, B^+, C^-);
10 **sync**;
11 $(D_1, D_2) \leftarrow \mathsf{ParallelMinMerge}(E, F)$;
12 **return** $\mathsf{Union}(D_1, D_2)$;

Algorithm 8: HalfCrossBaseMonomials

Input : four monomial sets A, B, C, D pairwise disjoint
Output: $\mathsf{Min}(\mathcal{G}_{A,B} \cup \mathcal{G}_{C,D})$ where $\mathcal{G}_{A,B}$ (resp. $\mathcal{G}_{C,D}$) consists of all non-constant $\gcd(x, y)$ for $(x, y) \in A \times B$ (resp. $C \times D$)

1 $E \leftarrow$ **spawn** CrossBaseMonomials(A, B);
2 $F \leftarrow$ **spawn** CrossBaseMonomials(C, D);
3 **sync**;
4 $(G_1, G_2) \leftarrow \mathsf{ParallelMinMerge}(E, F)$;
5 **return** $\mathsf{Union}(G_1, G_2)$;

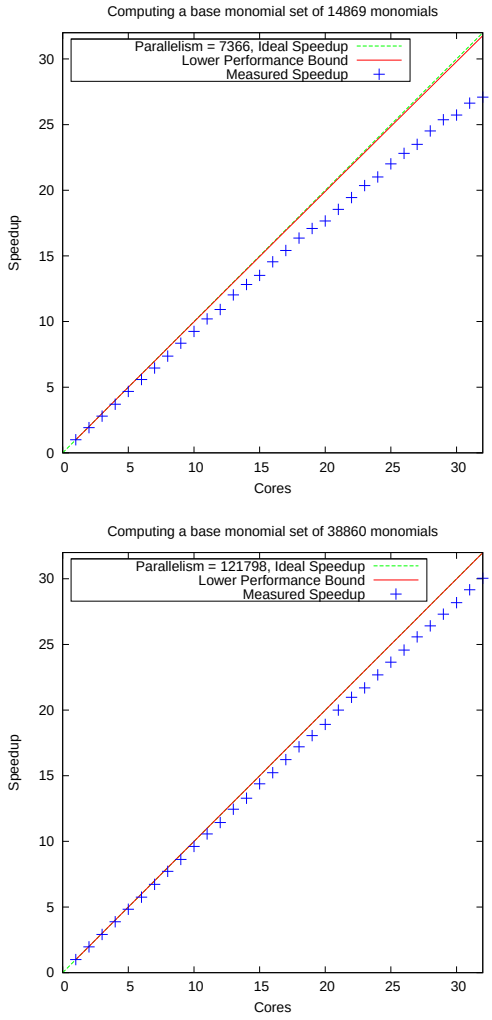

Figure 2: Scalability analysis for ParallelBaseMonomials **by Cilkview**

We have implemented these algorithms in Cilk++. A monomial is represented by an exponent vector. Each entry of an exponent vector is an unsigned `int`. A set A of input monomials is represented by an array of $|A|\,n$ unsigned `int`s where n is the number of variables. Accessing the elements in A is simply by indexing.

When the divide-and-conquer process reaches the base cases, at Lines 1–2 in Algorithm 5 and lines 1–2 in Algorithm 7, we compute either $\mathsf{innerPairsGcds}(A)$ or $\mathsf{crossPairsGcds}(B, C)$, followed by the computation of the minimal elements of these gcds. Here, we allocate dynamically the space to hold the gcds. Each execution of $\mathsf{innerPairsGcds}(A)$ allocates memory for $|A|(|A| - 1)/2$ gcds. Each execution of $\mathsf{crossPairsGcds}(B, C)$ allocates space for $|B||C|$ gcds. The size of these allocated memory spaces in the base cases is rather small, which, ideally, should fit in cache. Right after computing the gcds, we compute the minimal elements of these gcds in place. In other words, we remove those gcds which are not minimal for the divisibility relation. In the Union operations, for example the $\mathsf{Union}(D_1, D_2)$ in Line 9 in Algorithm 5, we reallocate the larger one between D_1 and D_2 to accommodate $|D_1| + |D_2|$ monomials and free the space of the smaller one. This dynamical-memory management strategy combined with the divide-and-conquer technique permits us to handle large sets of monomials otherwise impossible. This is confirmed by the benchmarks of our implementation.

Figure 2 gives the scalability analysis results by Cilkview on computing the base monomial sets of two large monomial sets. One has 14869 monomials with 28 variables, shown in the top. The number of minimal elements here is 14. Both of MIN_BASE and MIN_MERGE_BASE are 64. Its timing on 1 core is about 3.5 times less than the serial loop method, which is the function SerialInnerBaseMonomials. Using 32 cores we gain a speedup factor of 27 with respect to the timing on 1 core. Another monomial set has 38860 monomials with 30 variables. There are 15 minimal elements. The serial loop method for this case aborted due to memory allocation failure. However, our parallel execution reaches 30 speedup on 32 cores. We also notice that the ideal parallelism and the lower performance bound estimated by Cilkview for both benchmarks are very high but our measured speedup curve is lower than the lower performance bound. We attribute this performance degradation to the cost of our dynamic memory allocation.

5. TRANSVERSAL HYPERGRAPH GENERATION

Hypergraphs generalize graphs in the following way. A *hypergraph* $\mathcal{H}$ is a pair $(V, \mathcal{E})$ where V is a finite set and $\mathcal{E}$ is a set of subsets of V, called the *edges* (or *hyperedges*) of $\mathcal{H}$. The elements of V are called the *vertices* of $\mathcal{H}$. The number of vertices and edges of $\mathcal{H}$ are denoted here by $n(\mathcal{H})$ and $|\mathcal{H}|$ respectively; they are called the *order* and the size of $\mathcal{H}$. We denote by $\mathsf{Min}(\mathcal{H})$ the hypergraph whose vertex set is V and whose hyperedges are the minimal elements of the poset $(\mathcal{E}, \subseteq)$. The hypergraph $\mathcal{H}$ is said *simple* if none of its hyperedges is contained in another, that is, whenever $\mathsf{Min}(\mathcal{H}) = \mathcal{H}$ holds.

We denote by $\mathsf{Tr}(\mathcal{H})$ the hypergraph whose vertex set is V and whose hyperedges are the minimal elements of the poset $(\mathcal{T}, \subseteq)$ where $\mathcal{T}$ consists of all subsets A of V such that $A \cap E \neq \emptyset$ holds for all $E \in \mathcal{E}$. We call $\mathsf{Tr}(\mathcal{H})$ the *transversal* of $\mathcal{H}$. Let $\mathcal{H}' = (V, \mathcal{E}')$ and $\mathcal{H}'' = (V, \mathcal{E}'')$ be two hypergraphs. We denote by $\mathcal{H}' \cup \mathcal{H}''$ the hypergraph whose vertex set is V and whose hyperedge set is $\mathcal{E} \cup \mathcal{E}'$. Finally, we denote by $\mathcal{H}' \vee \mathcal{H}''$ the hypergraph whose vertex set is V and whose hyperedges are the $E' \cup E''$ for all $(E', E'') \in \mathcal{E}' \times \mathcal{E}''$. The following proposition [2] is the basis of most algorithms for computing the transversal of a hypergraph.

PROPOSITION 2. *For two hypergraphs $\mathcal{H}' = (V, \mathcal{E}')$ and $\mathcal{H}'' = (V, \mathcal{E}'')$ we have*

$$\mathsf{Tr}(\mathcal{H}' \cup \mathcal{H}'') = \mathsf{Min}(\mathsf{Tr}(\mathcal{H}') \vee \mathsf{Tr}(\mathcal{H}'')).$$

All popular algorithms for computing transversal hypergraphs, see [12, 16, 1, 7, 17], make use of the formula in Proposition 2 in an incremental manner. That is, writing $\mathcal{E} = E_1, \ldots, E_m$ and $\mathcal{H}_i = (V, \{E_1, \ldots, E_i\})$ for $i = 1 \cdots m$, these algorithms compute $\mathsf{Tr}(\mathcal{H}_{i+1})$ from $\mathsf{Tr}(\mathcal{H}_i)$ as follows

$$\mathsf{Tr}(\mathcal{H}_{i+1}) = \mathsf{Min}(\mathsf{Tr}(\mathcal{H}_i) \vee (V, \{\{v\} \mid v \in E_{i+1}\}))$$

The differences between these algorithms consist of various techniques to minimize the construction of unnecessary intermediate hyperedges. While we believe that these techniques are all important, we propose to apply Berge's formula *à la lettre*, that is, to divide the input hypergraph $\mathcal{H}$ into hypergraphs $\mathcal{H}'$, $\mathcal{H}''$ of similar sizes and such that $\mathcal{H}' \cup \mathcal{H}'' = \mathcal{H}$. Our intention is to create opportunity for parallel execution. At the same time, we want to control the intermediate expression swell resulting from the computation of

$$\mathsf{Tr}(\mathcal{H}) \vee \mathsf{Tr}(\mathcal{H}').$$

To this end, we compute this expression in a divide-and-conquer manner and apply the Min operator to the intermediate results.

Algorithm 9: ParallelTransversal

Input : A hypergraph $\mathcal{H}$
Output: $\mathsf{Tr}(\mathcal{H})$

```
1  if |H| ≤ TR_BASE then
2      return SerialTransversal(H);
3  (H⁻, H⁺) ← Split(H);
4  H⁻ ← spawn ParallelTransversal(H⁻);
5  H⁺ ← spawn ParallelTransversal(H⁺);
6  sync;
7  return ParallelHypMerge(H⁻, H⁺);
```

Algorithm 10: ParallelHypMerge

Input : $\mathcal{H}$, $\mathcal{K}$ such that $\mathsf{Tr}(\mathcal{H}) = \mathcal{H}$ and $\mathsf{Tr}(\mathcal{K}) = \mathcal{K}$.
Output: $\mathsf{Min}(\mathcal{H} \vee \mathcal{K})$

```
1  if |H| ≤ MERGE_HYP_BASE and
2     |K| ≤ MERGE_HYP_BASE then
3      return SerialHypMerge(H, K);
4  else if |H| > MERGE_HYP_BASE and
5          |K| > MERGE_HYP_BASE then
6      (H⁻, H⁺) ← Split(H);
7      (K⁻, K⁺) ← Split(K);
8      L ← spawn
9          HalfParallelHypMerge(H⁻, K⁻, H⁺, K⁺);
10     M ← spawn
11         HalfParallelHypMerge(H⁻, K⁺, H⁺, K⁻);
12     return Union(ParallelMinMerge(L, M));
13 else if |H| > MERGE_HYP_BASE and
14         |K| ≤ MERGE_HYP_BASE then
15     (H⁻, H⁺) ← Split(H);
16     M⁻ ← ParallelHypMerge(H⁻, K);
17     M⁺ ← ParallelHypMerge(H⁺, K);
18     return Union(ParallelMinMerge(M⁻, M⁺));
19 else
       // |H| ≤ MERGE_HYP_BASE and
       // |K| > MERGE_HYP_BASE
20     (K⁻, K⁺) ← Split(K);
21     M⁻ ← ParallelHypMerge(K⁻, H);
22     M⁺ ← ParallelHypMerge(K⁺, H);
23     return Union(ParallelMinMerge(M⁻, M⁺));
```

Algorithm 11: HalfParallelHypMerge

Input : four hypergraphs $\mathcal{H}, \mathcal{K}, \mathcal{L}, \mathcal{M}$
Output: $\mathsf{Min}(\mathsf{Min}(\mathcal{H} \vee \mathcal{K}) \cup \mathsf{Min}(\mathcal{L} \vee \mathcal{M}))$

```
1  N ← spawn ParallelHypMerge(K, H);
2  P ← spawn ParallelHypMerge(L, M);
3  sync;
4  return Union(ParallelMinMerge(N, P));
```

Algorithm 9 is our main procedure. Similarly to Algorithm 2, it proceeds in a divide-and-conquer manner with a threshold. For the base case, we call SerialTransversal($\mathcal{H}$), which can implement any serial algorithms for computing the transversal of hypergraph $\mathcal{H}$. When the input hypergraph is large enough, then this hypergraph is split into two so as to apply Proposition 2 with the two recursive calls performed concurrently. When these recursive calls return, their results are merged by means of Algorithm 10.

Given two hypergraphs $\mathcal{H}$ and $\mathcal{K}$, with the same vertex set, satisfying $\mathsf{Tr}(\mathcal{H}) = \mathcal{H}$ and $\mathsf{Tr}(\mathcal{K}) = \mathcal{K}$, the operation ParallelHypMerge of Algorithm 10 returns $\mathsf{Min}(\mathcal{H} \vee \mathcal{K})$. This operation is another instance of an application where the poset can be so large that it is desirable to compute its minimal elements concurrently to the generation of the poset itself, thus avoiding storing the entire poset in memory. As for the application described in Section 4, one can indeed efficiently generate the elements of the poset and compute its minimal elements simultaneously.

The principle of Algorithm 10 is very similar to that of Algorithm 3. Thus, we should simply mention two points. First, Algorithm 10 uses a subroutine, namely HalfParallelHypMerge of Algorithm 11, for clarity. Secondly, the base case of Algorithm 10, calls SerialHypMerge($\mathcal{H}, \mathcal{K}$), which can implement any serial algorithms for computing $\mathsf{Min}(\mathcal{H} \vee \mathcal{K})$.

We have implemented our algorithms in Cilk++ and benchmarked our code with some well-known problems on the same 32-core machine reported in Section 3. An implementation detail which is worth to mention is data representation. We represent each hyperedge as a bit-vector. For a hypergraph with n vertices, each hyperedge is encoded by n bits. By means of this representation, the operations on the hyperedges such as inclusion test and union can be reduced to bit operations. Thus, a hypergraph with m edges is encoded by an array of $m\,n$ bits. Traversing the hyperedges is simply by moving pointers to the bit-vectors in this array.

Our test problems can be classified into three groups. The first one consists of three types of large examples reported in [16]. We summarize their features and compare the timing results in Table 1. A scalability analysis for the three large problems in data mining on a 32-core is illustrated in Figure 3. The second group considers an enumeration problem (Kuratowski hypergraph), as listed in Table 2 and Figure 4. The third group is Lovasz hypergraph [2], reported in Table 3. The sizes of the three base cases used here (TR_BASE, MERGE_HYP_BASE and MIN_MERGE_BASE) are respectively 32, 16 and 128. Our experimentation shows that the base case threshold is an important influential factor on performance. In this work, they are determined by our test runs. To predict the feasible values based on the type of a poset and the hierarchical memory of a machine would definitely help. We shall develop a tool for this purpose when we deploy our software.

In Table 1, we describe the parameters of each problem following the same notation as in [16]. The first three columns indicate respectively the number of vertices, n, the number of hyperedges, m, and the number of minimal transversals, t. The problems classified as *Threshold*, *Dual Matching* and *Data Mining* are large examples selected from [16]. We have used `thg`, a Linux executable program developed by Kavvadias and Stavropoulos in [16] for their algorithm (namely KS), to measure the time for solving these problems on our machine. We observed that the timing results of `thg` on our machine were very close to those reported in [16]. Thus, we show here the timing results (seconds) presented in [16] in the fourth column (KS) in our Table 1. From the comparisons in [16], the KS algorithm outperforms the algorithm of Fredman and Khachiyan as implemented by Boros et al. in [3] (BEGK) and the algorithm of Bailey et al. given in [1] (BMR) for the *Dual Matching* and *Threshold* graphs. However, for the three large problems from data mining, the KS algorithm is about 30 to 60 percent slower than the best ones between BEGK and BMR.

In the last three columns in Table 1, we report the timing (seconds) of our program for solving these problems using 1 core and 32 cores, and the speedup factor on 32-core w.r.t on 1-core. On 1-core, our method is about 6 to 18 times faster for the selected *Dual Matching* problems and the large problems in data mining. Our program is particularly efficient for the *Threshold* graphs, for which it takes only about 0.01 seconds for each of them, while `thg` took about 11 to 82 seconds. In addition, our method shows significant speedup on multi-cores for the problems of large input size. As shown in Figure 3, for the three data mining problems, our code demonstrates linear speedup on 32 cores w.r.t the timing of the same algorithm on 1 core.

There are three sets of hypergraphs in [16] on which our method does not perform well, namely *Matching*, *Self-Dual Threshold* and *Self-Dual Fano-Plane* graphs. For these examples our code is about 2 to 50 times slower than the KS algorithm presented in [16]. Although the timing of such examples is quite small (from 0.01 to 178 s), they demonstrate the efficient techniques used in [16]. Incooperating such techniques into our algorithm is our future work.

Instance parameters			KS	ParallelTransversal		Speedup Ratio	
n	m	t	(s)	1-core (s)	32-core (s)	KS/1-core	KS/32-core
Threshold problems							
140	4900	71	11	0.01	-	1000	-
160	6400	81	23	0.01	-	2000	-
180	8100	91	44	0.01	-	4000	-
200	10000	101	82	0.02	-	4000	-
Dual Matching problems							
34	131072	17	57	9	0.57	6	100
36	262144	18	197	23	1.77	9	111
38	524288	19	655	56	3.53	12	186
40	1048576	20	2167	131	7.13	17	304
Data Mining problems							
287	48226	97	1648	92	3	18	549
287	92699	99	6672	651	21	10	318
287	108721	99	9331	1146	36	8	259

Table 1: Examples from [16]

The first family of classical hypergraphs that we have tested is related to an enumeration problem, namely the Kuratowski K_n^r hypergraphs. Table 2 gives two representative ones. This type of hypergraphs are defined by two parameters n and r. Given n distinct vertices, such a hypergraph contains all the hyperedges that have exactly r vertices. Our program achieves linear speedup on this class of hypergraphs with sufficiently large size, as reported in Table 2 and Figure 4 for K_{40}^5 and K_{30}^7. We have also used the `thg` program provided by the Authors of [16] to solve these problems. The timing for solving K_{30}^5 by the `thg` program is about 6500 seconds, which is about 70 times slower than our ParallelTransversal on 1-core. For the case of K_{40}^5 and K_{30}^7, it did not finish after running for more than 15 hours.

Instance parameters				KS	ParallelTransversal				
n	r	m	t	(s)	1-core	16-core		32-core	
					(s)	(s)	Speedup	(s)	Speedup
30	5	142506	27405	6500	88	6	14.7	3.5	25.0
40	5	658008	91390	>15 hr	915	58	15.8	30	30.5
30	7	2035800	593775	>15 hr	72465	4648	15.6	2320	31.2

Table 2: Tests for the Kuratowski hypergraphs

Another classical hypergraph is the Lovasz hypergraph, which is defined by a positive integer r. Consider r finite disjoint sets $X_1, \ldots, X_r$ such that X_j has exactly j elements, for $j = 1 \cdots r$.

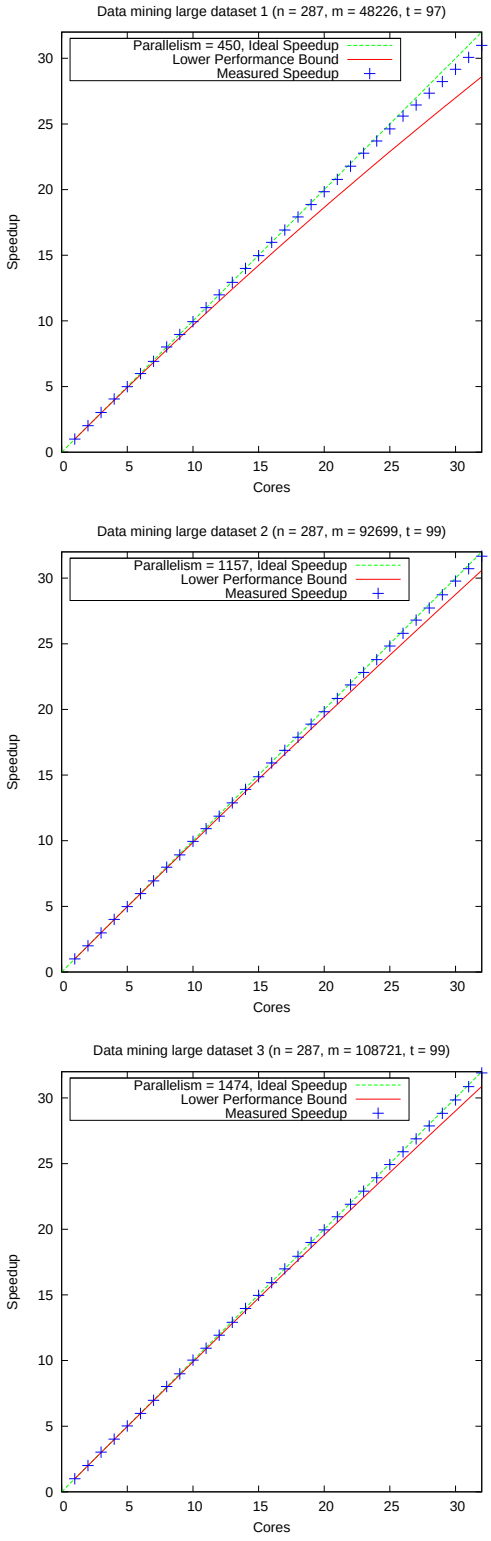

Figure 3: Scalability analysis on ParallelTransversal for data mining problems by Cilkview

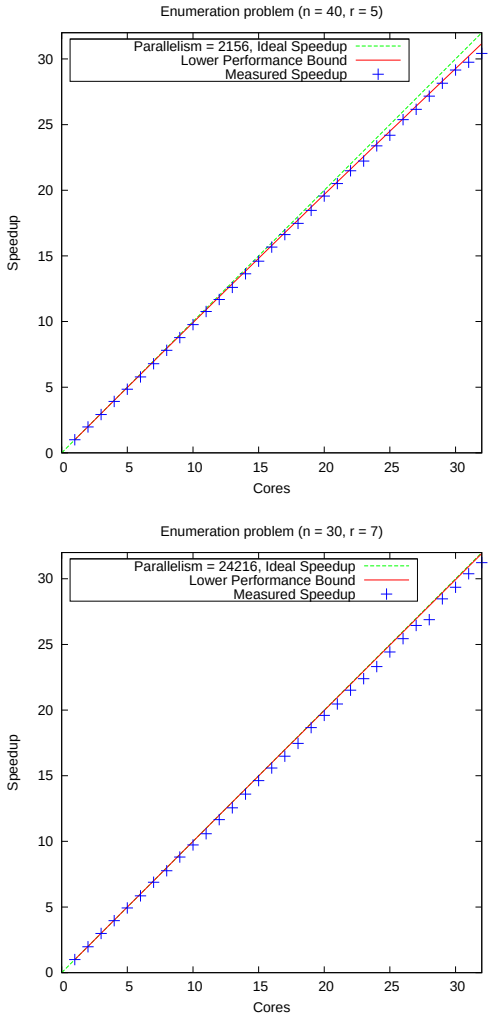

Figure 4: Scalability analysis on ParallelTransversal for K_{40}^5 and K_{30}^7 by Cilkview

The Lovasz hypergraph of rank r, denoted by L_r, has all its hyperedges of the form

$$X_j \cup \{x_{j+1}, \ldots, x_r\},$$

where $x_{j+1}, \ldots, x_r$ belong respectively to $X_{j+1}, \ldots, X_r$, for $j = 1 \ldots r$. We have tested our implementation with the Lovasz hypergraphs up to rank 10. For the problem of rank 9, we obtained 25 speedup on 32-core. For the one of rank 10, due to time limit, we only obtained the timing on 32-core and 16-core, which shows a linear speedup from 16 cores to 32 cores. The `thg` program solves the problem of rank 8 in 8000 seconds. For the problems of rank 9 and 10, the `thg` program did not complete after running for more than 15 hours.

Instance parameters				KS	ParallelTransversal				
n	r	m	t	(s)	1-core	16-core		32-core	
					(s)	(s)	Speedup	(s)	Speedup
36	8	69281	69281	8000	119	13	8.9	10	11.5
45	9	623530	623530	>15 hr	8765	609	14.2	347	25.3
55	10	6235301	6235301	>15 hr	-	60509	-	30596	

Table 3: Tests for the Lovasz hypergraphs

6. CONCLUDING REMARKS

In this paper, we have proposed a parallel algorithm for computing the minimal elements of a finite poset. Its implementation in Cilk++ on multi-cores is capable of processing large posets that a serial implementation could not process. Moreover, for sufficiently large input data set, our code reaches linear speedup on 32 cores.

We have integrated our algorithm into two applications. One is polynomial expression optimization and the other one is the computation of transversal hypergraphs. In both cases, we control intermediate expression swell by generating the poset and computing its minimal elements concurrently. Our Cilk++ code for computing transversal hypergraphs is competitive with the implementation reported by Kavvadias and Stavropoulos in [16]. Moreover, our code outperforms the one of our colleagues on three sets of large input problems, in particular the problems from data mining. However, our code is slower than theirs on other data sets. In fact, our code is a preliminary implementation, which simply applies Berge's formula in a divide-and-conquer manner. We still need to enhance our implementation with the various techniques which have been developed for controlling expression swell in transversal hypergraph computations [12, 16, 1, 7, 17].

We are extending the work presented in this paper in different directions. First, we would like to obtain a deeper complexity analysis of our algorithm for computing the minimal elements of a finite poset. Secondly, we are adapting this algorithm to the computation of GCD-free bases and the removal of redundant components.

Acknowledgements.

Great thanks to our colleagues Dimitris J. Kavvadias and Elias C. Stavropoulos for providing us with their program (implementing the KS algorithm) and their test suite. Sincere thanks to the reviewers for their constructive comments. We are grateful to Matteo Frigo for fruitful discussions on Cilk++. In addition, our benchmarks were made possible by the dedicated resource program of SHARCNET.

7. REFERENCES

[1] J. Bailey, T. Manoukian, and K. Ramamohanarao. A fast algorithm for computing hypergraph transversals and its application in mining emerging patterns. In *ICDM '03: Proceedings of the 3rd IEEE International Conference on Data Mining*, page 485, Washington, DC, USA, 2003. IEEE Computer Society.

[2] C. Berge. *Hypergraphes : combinatoire des ensembles finis*. Gauthier-Villars, 1987.

[3] E. Boros, K. hachiyan, K. Elbassioni, and V. Gurvich. An efficient implementation of a quasi-polynomial algorithm for generating hypergraph transversals. In *Proc. of the 11th European Symposium on Algorithms (ESA)*, volume 2432, pages 556–567. LNCS, Springer, 2003.

[4] J. Carnicer and M. Gasca. Evaluation of multivariate polynomials and their derivatives. *Mathematics of Computation*, 54(189):231–243, 1990.

[5] M. Ceberio and V. Kreinovich. Greedy algorithms for optimizing multivariate Horner schemes. *SIGSAM Bull.*, 38(1):8–15, 2004.

[6] C. Chen, F. Lemaire, M. Moreno Maza, W. Pan, and Y. Xie. Efficient computations of irredundant triangular decompositions with the `regularchains` library. In *Proc. of the International Conference on Computational Science (2)*, volume 4488 of *Lecture Notes in Computer Science*, pages 268–271. Springer, 2007.

[7] G. Dong and J. Li. Mining border descriptions of emerging patterns from dataset pairs. *Knowledge and Information Systems*, 8(2):178–202, 2005.

[8] T. Eiter and G. Gottlob. Hypergraph transversal computation and related problems in logic and ai. In *JELIA '02: Proceedings of the European Conference on Logics in Artificial Intelligence*, pages 549–564, London, UK, 2002. Springer-Verlag.

[9] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297, New York, USA, 1999.

[10] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *ACM SIGPLAN*, 1998.

[11] D. Gunopulos, R. Khardon, H. Mannila, S. Saluja, H. Toivonen, and R. S. Sharma. Discovering all most specific sentences. *ACM Trans. Database Syst.*, 28(2):140–174, 2003.

[12] M. Hagen. Lower bounds for three algorithms for transversal hypergraph generation. *Discrete Appl. Math.*, 157(7):1460–1469, 2009.

[13] U. Haus, S. Klamt, and T. Stephen. Computing knock out strategies in metabolic networks. *ArXiv e-prints*, 2008.

[14] Y. He, C. E. Leiserson, and W. M. Leiserson. The cilkview scalability analyzer. In *SPAA '10: Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pages 145–156, New York, USA, 2010. ACM.

[15] Intel Corp. Cilk++. http://www.cilk.com/.

[16] D. J. Kavvadias and E. C. Stavropoulos. An efficient algorithm for the transversal hypergraph generation. *Journal of Graph Algorithms and Applications*, 9(2):239–264, 2005.

[17] L. Khachiyan, E. Boros, K. M. Elbassioni, and V. Gurvich. A new algorithm for the hypergraph transversal problem. In *COCOON*, pages 767–776, 2005.

[18] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *J. ACM*, 22(4):469–476, 1975.

[19] C. E. Leiserson, L. Li, M. Moreno Maza, and Y. Xie. Efficient evaluation of large polynomials. In *Proc. International Congress of Mathematical Software - ICMS 2010*. Springer, 2010. To appear.

[20] C. E. Leiserson and T. B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *SPAA '10: Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pages 303–314, New York, USA, 2010. ACM.

[21] J. M. Pena. On the multivariate Horner scheme. *SIAM J. Numer. Anal.*, 37(4):1186–1197, 2000.

[22] J. M. Pena and T. Sauer. On the multivariate Horner scheme ii: running error analysis. *Computing*, 65(4):313–322, 2000.

[23] S. Sarkar and K. N. Sivarajan. Hypergraph models for cellular mobile communication systems. *IEEE Transactions on Vehicular Technology*, 47(2):460–471, 1998.