Parallel Fourier–Motzkin Elimination

Christoph W. Keßler*

Fachbereich 4 – Informatik
Universität Trier
D-54286 Trier, Germany
e-mail: kessler@psi.uni-trier.de

Abstract. Fourier–Motzkin elimination is a computationally expensive but powerful method to solve a system of linear inequalities for real and integer solution spaces. Because it yields an explicit representation of the solution set, in contrast to other methods such as Simplex, one may, in some cases, take its longer run time into account.
We show in this paper that it is possible to considerably speed up Fourier–Motzkin elimination by massively parallel processing. We present a parallel implementation for a shared memory parallel computer, and sketch several variants for distributed memory parallelization.

1 Introduction

Given a system $Ax \leq b$, $A \in \mathbb{R}^{n,m}$, $b \in \mathbb{R}^n$, of n linear inequalities in m variables, we ask for the existence of (1) a real solution $x \in \mathbb{R}^m$ of $Ax \leq b$, and (2) an integer solution $x \in \mathbb{Z}^m$. Furthermore we are interested in an explicit representation of the set of solutions.

Problem (1), a special case of linear programming, is polynomial in time. Geometrically, it corresponds to determining whether the intersection polytope of n halfspaces of the m–dimensional space is nonempty. It is usually solved using the well–known Simplex algorithm (see e.g. [6] for a survey) which has expected run time $O(nm(n+m))$ but takes exponential time $O(nm2^n)$ in the worst case. — Problem (2), the interior point problem for integer linear programming (cf. [6]) is NP–complete. Geometrically, it asks whether the intersection polytope of n halfspaces of the m–dimensional space contains any integer point.

Already in 1827, Fourier proposed an elimination method [4] that solves both problems. As expected, this algorithm takes non–polynomial run time. Indeed, the complexity can grow dramatically. Consequently, the method did not become widely known, and was re–invented several times, e.g. by Motzkin in 1936. For certain cases, however, it is a quite useful tool, because it is constructive: If a solution exists, it yields a representation of the convex intersection polytope. This representation may, of course, be used to determine the complete set of all feasible integer solutions x by an enumeration procedure, provided that this set is finite. But it can also be used to supply a *symbolic* solution. This feature is used e.g. when applying restructuring loop transformations to a numerical program with the goal of parallelizing it, see [3] for a detailed discussion.

Clearly, its high worst–case computational complexity made Fourier–Motzkin elimination impractical as a general tool to solve the integer case. But even if medium–sized problems would already take too much time on a uniprocessor

* The full version of this paper can be obtained from the author.

system, they could nevertheless be solved on a massively parallel computer. We show that Fourier–Motzkin elimination offers a great potential for the exploitation of massive parallelism. We give an implementation for a shared–memory multiprocessor and sketch variants for a distributed–memory implementation.

2 Fourier-Motzkin Elimination

Since the (sequential) algorithm is not widely known, we give a summary of the excellent description given in [3].

The algorithm is subdivided into seven steps.

Step 1: We are given $A = (a_{ij})_{i,j} \in \mathbb{R}^{n,m}$ and $b \in \mathbb{R}^n$, representing the system $Ax \leq b$. We set up a "working system", consisting of a matrix $T \in \mathbb{R}^{n.m}$ and a vector $q \in \mathbb{R}^n$. We initialize $t_{ij} = a_{ij}$ and $q_i = b_i$ for $1 \leq i \leq n$, $1 \leq j \leq m$, and initialize the current problem sizes r, s by $r = m$ and $s = n$.

Step 2: We sort the s inequalities and determine indices $n_1, n_2 \in \mathbb{N}$, $1 \leq n_1 \leq n_2 \leq s$ such that, after renaming of the indices of the inequalities, $t_{ir} > 0$ for $1 \leq i \leq n_1$, $t_{i'r} < 0$ for $n_1 + 1 \leq i' \leq n_2$, and $t_{i''r} = 0$ for $n_2 + 1 \leq i'' \leq s$.

Step 3: We normalize the first n_2 inequalities by $t_{ij} = t_{ij}/t_{ir}$ and $q_i = q_i/t_{ir}$ for $1 \leq i \leq n_2$, $1 \leq j \leq r-1$. Now the system looks as follows:

$$t_{i1}x_1 + t_{i2}x_2 + \ldots + t_{i,r-1}x_{r-1} + x_r \leq q_i, \quad 1 \leq i \leq n_1 \tag{1}$$

$$t_{i'1}x_1 + t_{i'2}x_2 + \ldots + t_{i,r-1}x_{r-1} + x_r \geq q_{i'}, \quad n_1 + 1 \leq i' \leq n_2 \tag{2}$$

$$t_{i''1}x_1 + t_{i''2}x_2 + \ldots + t_{i,r-1}x_{r-1} \leq q_{i''}, \quad n_2 + 1 \leq i'' \leq n_2. \tag{3}$$

Step 4: From subsystem (1) we obtain $x_r \leq q_i - \sum_{j=1}^{r-1} t_{ij}x_j$ for $1 \leq i \leq n_1$, thus $B_r^U(x_1, ..., x_{r-1}) = \min_{1 \leq i \leq n_1}(q_i - \sum_{j=1}^{r-1} t_{ij}x_j)$ is an upper bound for x_r. If $n_1 = 0$, we set $B_r^U(x_1, ..., x_{r-1}) = +\infty$.

In the same way, (2) yields $x_r \geq q_{i'} - \sum_{j=1}^{r-1} t_{i'j}x_j$ for $n_1 + 1 \leq i' \leq n_2$, thus $B_r^L(x_1, ..., x_{r-1}) = \max_{n_1+1 \leq i' \leq n_2}(q_{i'} - \sum_{j=1}^{r-1} t_{i'j}x_j)$ is a lower bound for x_r. If $n_2 = n_1$, we set $B_r^L(x_1, ..., x_{r-1}) = -\infty$. Thus, the range $B_r^L(x_1, ..., x_{r-1}) \leq x_r \leq B_r^U(x_1, ..., x_{r-1})$ of feasible values for variable x_r is given in terms of feasible values for variables $x_1, ..., x_{r-1}$. We record these bounds for later use.

Step 5: If $r = 1$, we are done, since the bounds B_1^L, B_1^U are constants (maybe $\pm\infty$). In this case we can return the answer to the original problem:

If and only if $B_1^L \leq B_1^U$ and $q_{i''} \geq 0$ for all i'', $n_2 + 1 \leq i'' \leq s$, then the original system has a *real* solution $x \in \mathbb{R}^m$. This feature installs correctness of the algorithm, provided that exact arithmetic has been used. A proof by induction is straightforward.

Otherwise, if $r > 1$, we have to continue:

Step 6: We eliminate x_r. As minimizations and maximizations cannot be directly expressed in a linear system, we do this by setting each component of the lower bound for x_r less than or equal to each component of the upper bound for x_r. This produces $n_1(n_2 - n_1)$ new inequalities in $r - 1$ variables:

$$q_{i'} - \sum_{j=1}^{r-1} t_{i'j}x_j \leq x_r \leq q_i - \sum_{j=1}^{r-1} t_{ij}x_j \quad \text{for all } i, i', \text{ with } 1 \leq i \leq n_1,\ n_1 + 1 \leq i' \leq n_2.$$

To these we add the $s - n_2$ old inequalities from (3). This yields a new system with $s' = s - n_2 + n_1(n_2 - n_1)$ inequalities in $r - 1$ variables. The new system has

a real solution iff system (1,2,3) has a real solution. By induction, we obtain that the new system has a real solution iff the original system has a real solution.

If $s' = 0$, we are done; then the variables $x_1, ..., x_{r-1}$ can be chosen arbitrarily; the system has infinitely many solutions. Otherwise, we continue:

Step 7: In the new system, we renumber the coefficients as $t_{i,j}$ and q_i with $1 \leq i \leq s'$ and $1 \leq j \leq r-1$. We set $s = s'$, $r = r-1$ and iterate from step 2.

The algorithm determines whether $Ax = b$ has a real solution $x \in \mathbb{R}^m$, and, if yes, supplies, as a byproduct, an explicit representation of the solution set.

Due to the construction of the algorithm, any real solution $x \in \mathbb{R}^m$ fulfills $B_r^L(x_1, ..., x_{r-1}) \leq x_r \leq B_r^U(x_1, ..., x_{r-1})$ for $1 \leq r \leq m$. However, if an *integer* solution $x \in Z^m$ is required, the answer "yes" by Fourier–Motzkin elimination does not suffice to guarantee an integer solution. This means that we have to test explicitly whether the following system is fulfilled:

$$\begin{array}{rcl} \lceil B_m^L(x_1, ..., x_{m-1}) \rceil & \leq x_m \leq & \lfloor B_m^U(x_1, ..., x_{m-1}) \rfloor \\ \lceil B_{m-1}^L(x_1, ..., x_{m-2}) \rceil & \leq x_{m-1} \leq & \lfloor B_{m-1}^U(x_1, ..., x_{m-2}) \rfloor \\ \vdots & \vdots & \vdots \\ \lceil B_1^L \rceil & \leq x_1 \leq & \lfloor B_1^U \rfloor \end{array} \tag{4}$$

If the (integer) solution set is finite, i.e. there are no infinite upper or lower bounds B_r^U, B_r^L for some r, $1 \leq r \leq m$, the following loop nest produces the complete solution set:

forall $x_1 \in \{\lceil B_1^L \rceil, ..., \lfloor B_1^U \rfloor\}$
  forall $x_2 \in \{\lceil B_2^L(x_1) \rceil, ..., \lfloor B_2^U(x_1) \rfloor\}$
    $\ddots$
      forall $x_m \in \{\lceil B_{m-1}^L(x_1, ..., x_{m-1}) \rceil, ..., \lfloor B_{m-1}^U(x_1, ..., x_{m-1}) \rfloor\}$
        print x

This makes, of course, only sense if the solution set does not become too large; thus a-priori knowledge on the maximum size of the solution set is required here. Clearly, if only the existence of an integer solution x is in question, it suffices to abort all these **forall** loops after the first feasible x has been found.

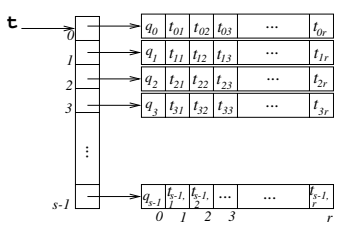
Moreover, if one is interested in a symbolic representation of the solution set, e.g. when determining the new loop limits for a restructured loop nest (see [3] for an example), the bounds for x due to (4) directly supply this representation.

The run time of Fourier Motzkin elimination may be disastrous in the worst case because the number of inequalities may square in each iteration (if always $n_1 = n_2 = s/2$). Nevertheless, on the average it should be considerably lower: The probability that the first argument of T is maximal in each recursion step is rather small. Moreover, the sparsity structure of A has a considerable influence on the run time, because $n_2 \ll s$ if the matrix contains many zero elements. At least for the inequalities (3) that do not participate in a specific elimination step, the sparsity pattern is preserved by the algorithm. For the other inequalities, the number of non–zero coefficients may, in the worst case, double in each iteration.

3 Parallelization for Shared Memory

We found the following shared data structure useful for speeding up the sorting steps (2 and 7): Pointers to the inequalities of each iteration are stored in a

dynamically allocated array `t` with s entries. Thus, interchanging of inequalities can be done in constant time by just interchanging the pointers to them.



The coefficients t_{ij} of each inequality i in r variables are stored in a dynamically allocated array `t[`i`]` with $r+1$ entries. For simplicity and space economy, we store the right hand side values q_i as the zeroth entry `t[`i`][0]` of each inequality array. The pointers `t` to the overall system of all iterations r are, in turn, stored in an array that later allows accessing the lower and upper bound expressions for each x_r.

If the original matrix A is sparse, it suffices to store the nonzero elements t_{ij} for each inequality, together with the column index j. We implemented only the dense variant because (a) sparsity becomes worse in the course of the algorithm, and (b) exploiting sparsity only pays off if m exceeds a certain value, which, on the other hand, may lead to very long run times.

We assume a multiprocessor with p processors. Each processor has constant time access to a large shared memory. Concurrent write operations are resolved by using an atomic *fetch&add* construct that takes constant time, independent of the number of processors participating in this operation. A research prototype of a machine with this ideal behaviour, the SB-PRAM [1, 2], is currently being built by W.J. Paul's group at the University of Saarbrücken. As programming language, we use Fork95, an extension of ANSI C for general–purpose PRAM programming. See [5] and `http://www-wjp.cs.uni-sb.de/fork95/` for further details.

Step 2 of the algorithm can be done in parallel. The `mpadd` instruction, an atomic *fetch&add* primitive, performs in 1 CPU cycle on the SB-PRAM, regardless of the number of participating processors.

This feature is very helpful here; the overall sorting step (see on the right) producing a sorted system `t` from an unsorted system `t_old`, is performed by $p \leq s$ processors in time $O(s/p)$. `gforall(i,lb,ub,p)` is a macro that denotes a parallel loop whose (private) loop index variable `i` globally ranges from `lb` to `ub-1`, with iterations being cyclically distributed over the participating `p` processors. If `p` exceeds the number `ub-lb` of iterations, the remaining processors remain idle and could be used for further (interior) levels of parallelism. `pr` is a type qualifier that declares a variable as private to each processor.

Step 3 contains $n_2(r+1)$ divisions; these can completely execute in parallel provided that a data dependency cycle is resolved by a temporary shared array `f[]` (see code on the right). Thus, step 3 runs in time $O(n_2(r+1)/p)$ on $p \leq n_2(r+1)$ processors.

```
pr int mypos, i, j, ii;
n2=0; nn = s-1;
gforall (i, 0, s, p ) {
  if (t_old[i][r] != 0)
       mypos = mpadd(&n2, 1);
  else mypos = mpadd(&nn,-1);
  t[mypos] = t_old[i]; }
gforall (i, 0, s, p )
  t_old[i] = t[i];
n1 = 0; /* nn is now n2-1 */
gforall (i, 0, n2, p ) {
  if (t_old[i][r] > 0)
       mypos = mpadd(&n1, 1);
  else mypos = mpadd(&nn,-1);
  t[mypos] = t_old[i]; }
free( t_old );
{determine pi,pj with pi*pj=p,
 pi<=min(n1,n2-n1) maximal }
gforall (i, 0, n1, pi)
  f[i] = 1.0 / t[i][r];
gforall (i, 0, n1, pi)
  gforall (j, 0, r+1, pj)
    t[i][j] *= f[i];
```

Step 4 records the inequalities from (1) and (2) that install upper resp. lower bounds on x_r, for later use. Thus, storage for these inequalities cannot be freed.

```
gforall (i, n1, n2, pi)
  f[i] = -(1.0 / t[i][r]);
gforall (i, n1, n2, pi)
  gforall (j, 0, r+1, pj)
     t[i][j] *= f[i];
```

Step 5 handles the special case $r = 1$. Explicit computing of B_1^U and B_1^L is done in time $O((n_2 \log p)/p)$ on p processors. If we are interested in an integer solution, we can, compared to conventional parallel minimization / maximization, save the $\log p$ factor using fast integer maximization/minimization which is supplied by the `mpmax` operator, a multiprefix maximization instruction that performs in constant time on the SB-PRAM.

Step 6 constructs a new system of inequalities (see the kernel on the right). If $p \leq n_1(n_2 - n_1)r$, then this kernel executes in time $O(n_1(n_2 - n_1)r/p)$. Note that we may here also compute the position of each new inequality as `mypos = i*n2+ii`, without using the `mpadd` instruction. `alloc()` performs memory allocation of permanent shared heap blocks. Using `mpadd`, it runs in constant time, regardless of the number of participating processors.

```
{ comp. pi,pii,pj with pi max.
  and pi*pii*pj=p  }
gforall (i, 0, n1, pi) {
  gforall (ii, n1, n2, pii) {
    pr ineq myineq;
    farm {
     mypos = mpadd(&s_new,1);
     myineq = (ineq) alloc(
            r*sizeof(double));
     gforall (j, 0, r, pj )
      myineq[j] = t[i][j]
                + t[ii][j];}
    t_new[mypos]=myineq;  } }
```

Appending the old $s - n_2$ inequalities from (3), we only need to copy the pointers to them (see code on the right), resulting in run time $O((s - n_2)/p)$ on $p \leq s - n_2$ processors.

```
gforall (i, n2, s, p)
  t_new[mpadd(&s_new,1)]=t[i];
```

The renumbering as indicated in step 7 is implicitly performed during step 6; thus step 7 takes only constant time.

Results Table 1 shows some measurements for our implementation. Since the SB-PRAM hardware is not yet operational, we use the SB-PRAM simulator running on a SUN workstation. The simulator produces exact timings; one SB-PRAM clock cycle (cc) will take 4 microseconds on the SB-PRAM prototype with 4096 processors currently being built at Saarbrücken University.

We have ported the Fork95 program to a Cray EL98 with 8 processors, using Cray Microtasking. The vector units of this machine are exploited best if interior loops (e.g. the `j` loops) are vectorized, which is generally possible here. Longer vectors are possible if chaining features are exploited; this enables processing all inequalities owned by a processor as one large vector update operation. However, because there is no equivalent to `mpadd()` on the Cray, step 2 is sequentialized; thus the speedup observed is rather modest (1.55 for 4, and 2.6 for 8 processors, applied to a 27×4 random problem).

4 Parallelization for Distributed Memory

We sketch[2] three different scenarios for distributing data across p processors of a distributed memory system. Each possibility has advantages and drawbacks.

(1) The s inequalities are equally distributed among the processors. Step 2 of each iteration installs the invariant that each processor holds approximately

[2] For space limitations we cannot go into more detail here. See the full version.

DENSE	$n = 12,\ m = 4$	
p	time [cc]	speedup
1	15489470	1.00
2	7794002	1.99
4	3945966	3.93
8	1999718	7.75
16	1049920	14.75
32	553168	28.00
64	327088	47.36
128	214408	72.24

DENSE	$n = 16,\ m = 4$	
p	time [cc]	speedup
1	194009292	1.00
2	97116338	2.00
4	48602188	3.99
8	24343832	7.97
16	12648663	15.34
32	6254904	31.02
64	3166208	61.27
128	1695908	114.40
256	960648	201.96
512	592964	327.19
1024	341092	568.79

SPARSE	$n = 200,\ m = 10$	
p	time [cc]	speedup
1	52230692	1.00
2	26178784	2.00
4	13160404	3.97
8	6649968	7.85
16	3872185	13.49
32	1769536	29.52
64	957852	54.53
128	554952	94.12
256	363690	143.61

Table 1. Measurements on the SB-PRAM for feasible dense random systems. All entries are nonzero and chosen such that $n_1 \approx n_2 - n_1$ and $n_2 = s$ in each iteration. Speedup is almost linear. Slight speedup degradations for large numbers of processors arise from many processors being idle in the first, least expensive iterations, and from some sequential overhead. Nevertheless, the combinatorial explosion, especially regarding space requirements, is discouraging for larger dense systems. — The right hand column shows measurements on the SB-PRAM for a sparse random system; 12.5% of the entries a_{ij} are nonzero. Sparsity considerably delays the combinatorial explosion.

the same amount of inequalities of each of the three categories (1), (2) and (3), namely n_1/p, $(n_2 - n_1)/p$, and $(s - n_2)/p$, respectively. Computational load is perfectly balanced. This causes much communication for step 2 but modest communication for step 6.

(2) The s inequalities are equally distributed among the processors, but the local ratios of inequality categories do not necessary correspond to the global ratio of n_1 to n_2 to s. Computational load is perfectly balanced. Less communication is required in step 2 but slightly more in step 6.

(3) The r variables are cyclically distributed among the processors. Computational load is not perfectly balanced for the last $p - 1$ iterations which are probably computationally most expensive (requires thus combining with (1) or (2)). Steps 2 and 6 do not require any communication at all, but Step 3 now requires a broadcast for each inequality.

References

1. F. Abolhassan, R. Drefenstedt, J. Keller, W.J. Paul, and D. Scheerer. On the physical design of PRAMs. *Computer Journal*, 36(8):756–762, Dec. 1993.
2. F. Abolhassan, J. Keller, and W.J. Paul. On the cost–effectiveness of PRAMs. *Proc. 3rd IEEE Symp. on Parallel and Distributed Processing*, 2–9, 1991.
3. U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, 1993.
4. J.B.J. Fourier. (reported in:) Analyse des travaux de l'Académie Royale des Sciences pendant l'année 1824, Partie mathématique, 1827. Engl. transl. (partially) in: D.A. Kohler, *Translation of a report by Fourier on his work on linear inequalities*, *Opsearch* **10** (1973) 38–42.
5. C.W. Keßler and H. Seidl. Integrating Synchronous and Asynchronous Paradigms: The Fork95 Parallel Programming Language. Proc. MPPM-95 Int. Conf. on Massively Parallel Programming Models, Berlin, Germany, 1995.
6. A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1986.

This article was processed using the LaTeX macro package with LLNCS style