Submission to EUROPAR'96, Workshop #07 Parallel Numerical Algorithms

Parallel Fourier–Motzkin Elimination

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

July 18, 1996

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 the first parallel implementation of this method, one variant for shared memory parallel computers, and one for distributed memory systems.

Key words: integer linear programming, linear optimization, system of linear inequalities, Fourier-Motzkin elimination, massive parallelism

1 Introduction

We consider the common problem of solving a system of n linear inequalities in m variables. Formally, we would like to find a solution of the system

$$Ax \leq b \quad \text{with } A \in \mathbb{R}^{n,m},\ b \in \mathbb{R}^n \tag{1}$$

and distinguish between two different goals:

1. Does there exist a real solution $x \in \mathbb{R}^m$ of $Ax \leq b$?
2. Does there exist an integer solution $x \in \mathbb{Z}^m$ of $Ax \leq b$?

Furthermore we are interested in an explicit representation of the set of solutions.

The first problem, well–known as a special case of linear programming, has been shown to be polynomial in time [Kha79]. Geometrically, it corresponds to the problem to determine 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. [Sch86] for a survey) which has expected run time $O(nm(n+m))$ but takes exponential time $O(nm2^n)$ in the worst case [KM72].

The second problem, the interior point problem for integer linear programming, is well–known to be NP–complete. Geometrically, it asks whether the intersection polytope of n halfspaces of the m–dimensional space contains any integer point of $\mathbb{Z}^m$.

Already in 1827, the French mathematician Fourier proposed an elimination method [Fou27] 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 [Mot36]. For certain cases, however, it is a quite useful tool, because it is constructive, i.e. it yields, if the answer is "yes", 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 [DE73, Wil76, Wil83]. 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 [Ban93] for a detailed discussion.

The special case $x \in \{0,1\}^m$ corresponds to an n–fold Knapsack problem and is not considered here; there exist other, more suitable methods for this binary case (cf. [Sch86]).

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 system, they could nevertheless be solved on a massively parallel computer. In order to prove this claim, we show in this paper that Fourier–Motzkin elimination offers a great potential for the exploitation of massive parallelism. We give an implementation for a shared–memory multiprocessor and for a distributed–memory system.

The rest of this paper is organized as follows: Section 2 revisits the sequential algorithm. A parallel implementation for a shared memory parallel computer is given in Section 3. Section 4 proposes a variant for a distributed–memory parallel system. Section 5 concludes.

2 Fourier-Motzkin Elimination

Since the (sequential) algorithm is not widely known, we provide here a summary of the excellent description given in [Ban93]. The notation that we introduce in this section will later be used in the parallel implementations.

2.1 The algorithm

The Fourier–Motzkin elimination algorithm is subdivided into seven steps.

Step 1: We are given a matrix $A = (a_{ij})_{i,j} \in \mathbb{R}^{n,m}$ and a vector $b \in \mathbb{R}^n$, representing the system

$$\begin{array}{rcl} a_{11}x_1 + a_{12}x_2 + \dots & +a_{1m}x_m & \leq b_1 \\ \vdots & \vdots & \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \dots & +a_{nm}x_m & \leq b_n \end{array} \tag{2}$$

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

$$\begin{array}{rcll} t_{ij} & = & a_{ij} & \text{for all } i,j,\ 1 \leq i \leq n, 1 \leq j \leq m \\ q_i & = & b_i & \text{for all } i,\ 1 \leq i \leq n. \end{array}$$

Furthermore, we initialize the current problem sizes r and 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, finally

$$\begin{array}{rcll} t_{ir} & > & 0 & \text{for } 1 \leq i \leq n_1, \\ t_{i'r} & < & 0 & \text{for } n_1+1 \leq i' \leq n_2, \text{ and} \\ t_{i''r} & = & 0 & \text{for } n_2+1 \leq i'' \leq s. \end{array}$$

.

Step 3: We normalize the first n_2 inequalities as follows:

$$\begin{array}{rcll} t_{ij} & = & t_{ij}/t_{ir} & \text{for } 1 \leq i \leq n_2, 1 \leq j \leq r-1 \\ q_i & = & q_i/t_{ir} & \text{for } 1 \leq i \leq n_2. \end{array}$$

Now the system looks as follows:

$$t_{i1}x_1 + t_{i2}x_2 + \dots + t_{i,r-1}x_{r-1} + x_r \leq q_i, \quad 1 \leq i \leq n_1 \tag{3}$$
$$t_{i'1}x_1 + t_{i'2}x_2 + \dots + t_{i,r-1}x_{r-1} + x_r \geq q_{i'}, \quad n_1+1 \leq i' \leq n_2 \tag{4}$$
$$t_{i''1}x_1 + t_{i''2}x_2 + \dots + t_{i,r-1}x_{r-1} \leq q_{i''}, \quad n_2+1 \leq i'' \leq n_2. \tag{5}$$

Step 4: From subsystem (3) we obtain

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

thus

$$B_r^U(x_1, ..., x_{r-1}) = \min_{1 \leq i \leq n_1} \left(q_i - \sum_{j=1}^{r-1} t_{ij}x_j \right)$$

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, we obtain from subsystem (4) that

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

thus

$$B_r^L(x_1, ..., x_{r-1}) = \max_{n_1+1 \le i' \le n_2} \left(q_{i'} - \sum_{j=1}^{r-1} t_{i'j} x_j \right)$$

is a lower bound for x_r. If $n_2 = n_1$, we set $B_r^L(x_1, ..., x_{r-1}) = -\infty$.

Now we have the range

$$B_r^L(x_1, ..., x_{r-1}) \le x_r \le B_r^U(x_1, ..., x_{r-1})$$

of feasible values for variable x_r, 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 $B_1^L \le B_1^U$ and $q_{i''} \ge 0$ for all i'', $n_2 + 1 \le i'' \le s$, then the system (2) 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 from the current system. 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:

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

To these inequalities we add the $s - n_2$ old inequalities from (5). This yields a new system with $s' = s - n_2 + n_1(n_2 - n_1)$ inequalities in $r - 1$ variables. It is easy to see that this new system has a real solution if and only if system (3,4,5) has a real solution. By induction, we obtain that the new system has a real solution iff the original system (2) 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

$$\begin{cases} \sum_{j=1}^{r-1} (t_{ij} - t_{i'j})\, x_j & \le q_i - q_{i'} \quad \text{for all } i, i', \text{ with } 1 \le i \le n_1, \; n_1 + 1 \le i' \le n_2, \\ \sum_{j=1}^{r-1} t_{i''j} x_j & \le q_{i''} \quad \text{for all } i, i', \text{ with } n_2 + 1 \le i'' \le s, \end{cases}$$

we renumber the coefficients as $t_{i,j}$ and q_i with $1 \le i \le s'$ and $1 \le j \le r - 1$. We set $s = s'$, $r = r - 1$ and iterate, starting at step 2.

2.2 Properties of the algorithm

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

According to the construction of the algorithm, any real solution $x \in \mathbb{R}^m$ fulfills

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

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:

$$
\begin{array}{rcl}
\lceil B_m^L(x_1,...,x_{m-1})\rceil & \leq x_m \leq & \lfloor B_m^U(x_1,...,x_{m-1})\rfloor \\
\lceil B_{m-1}^L(x_1,...,x_{m-2})\rceil & \leq x_{m-1} \leq & \lfloor B_{m-1}^U(x_1,...,x_{m-2})\rfloor \\
\vdots & \vdots & \vdots \\
\lceil B_1^L\rceil & \leq x_1 \leq & \lfloor B_1^U\rfloor
\end{array}
\tag{6}
$$

There are several ways to solve this question:

- If one knows (from the information provided in the course of the Fourier–Motzkin elimination process) that the (integer) solution set is finite, i.e. there are no upper bounds $B_r^U = +\infty$ for some r, $1 \leq r \leq m$, and no lower bounds $B_r^L = -\infty$ for some r, then 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.

- 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 [Ban93] for an example), then the bounds for x due to (6) directly supply this representation.

2.3 Run time estimation

The run time of Fourier Motzkin elimination may be disastrous in the worst case:

Let $T(n,m)$ denote the run time of the algorithm for a system of n inequalities in m variables. We have $T(n,1) = O(1)$ due to the special case $r = 1$ in step 5, and $T(0,m) = O(1)$ due to the special case $s' = 0$ in step 6. For the general case, we obtain (merely from steps 3 and 6)

$$T(s,r) = O(sr) + \max_{1\leq n_1\leq n_2\leq s} T(n_1(n_2-n_1)+s-n_2, r-1)$$

The first argument of T in the recursion is maximized if $n_1 = n_2 = n/2$. This yields

$$
\begin{aligned}
T(n,m) &\leq O(nm) + T(n^2/4, m-1) \\
&= O\left(\sum_{r=0}^{m-1} (m-r) \frac{n^{2^r}}{4(2^r-1)}\right)
\end{aligned}
$$

thus the algorithm may be quite expensive if m is not small.

Nevertheless, the average run time should be considerably lower, because of two reasons:

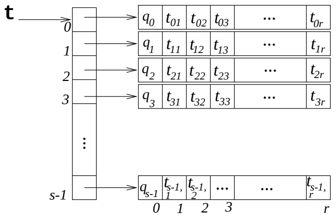
1. The probability that the first argument of T is maximal in each recursion step is rather small.

2. 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 (5) 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–negative coefficients may double in the worst case (*fill–in*) in each iteration.

3 Parallelization for Shared Memory

3.1 Data structure

We found the following shared data structure useful for speeding up the sorting steps (step 2 and step 7) of the algorithm: Pointers to the inequalities of each iteration are stored in a dynamically allocated array `t` with s entries. Thus, interchanging of two 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. However, 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.

3.2 Model of parallel computation: PRAM

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 [AKP91, ADK+93], 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. We refer to [KS95] and `http: //www-wjp.cs.uni-sb.de/fork95/` for further details.

3.3 Exploiting Parallelism

Step 2 of the algorithm can be done in parallel. The `mpadd` instruction is an atomic *fetch&add* primitive performing 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

```
n2=0; nn = s-1;
gforall (i, 0, s, p ) {
   pr int mypos;
   if (t_old[i][r] != 0)  mypos = mpadd( &n2, 1 );
   else /* == 0 */        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 ) {
   pr int mypos;
   if (t_old[i][r] > 0)   mypos = mpadd( &n1, 1 );
   else /* < 0 */         mypos = mpadd( &nn,-1 );
   t[mypos] = t_old[i];
}
free( t_old );
```

producing a sorted system `t` from an unsorted system `t_old`, is performed by $p \leq n$ 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 be completely executed in parallel provided that a data dependency cycle is resolved by a temporary shared array `factor[]`:

```
{ determine pi,pj with pi*pj=p, pi<=min(n1,n2-n1) maximal }
```

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

gforall (i, n1, n2, pi)
   factor[i] = -(1.0 / t[i][r]);
   /*then the inequality sign needs not be reversed */
gforall (i, n1, n2, pi)
   gforall (j, 0, r+1, pj)
      t[i][j] *= factor[i];
```

Thus, step 3 runs in time $O(n_2(r+1)/p)$ on $p \leq n_2(r+1)$ processors.

Step 4 records the inequalities from (3) and (4) that install upper resp. lower bounds on x_r, for later use. Thus, storage for these inequalities cannot be freed.

Step 5 handles the special case $r = 1$. Explicit computing of B_1^U and B_1^L is done in parallel 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. The kernel of the parallel implementation is:

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

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.

Appending the old $s - n_2$ inequalities from (5), we only need to copy the pointers to them:

	$n = 12,\ m = 4$	
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

	$n = 16,\ m = 4$	
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

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.

```
gforall (i, n2, s, p) {
   pr int mypos = mpadd( &s_new, 1 );
   t_new[mypos] = t[i];
}
```

resulting in run time $O((s - n_2)/p)$ on $p \leq s - n_2$ processors.

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

3.4 Results

Tables 1 and 2 show 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 are currently porting the Fork95 program to a Cray EL98 with 8 processors. 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 would enable processing all inequalities owned by a processor as one large vector update operation. The measured execution times for the Cray EL98 will be included into the final version of this paper.

	$n = 200,\ m = 10$	
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 2: 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.

4 Parallelization for Distributed Memory

We sketch 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 the same amount of inequalities of each of the three categories (3), (4) and (5), 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. Step 2 and Step 6 do not require any communication at all, but Step 3 now requires a broadcast for each inequality.

Let us consider these scenarios in more detail.

4.1 Data Distribution, First Variant

We start with an arbitrary, equal distribution of the n inequalities over the p processors.

In each iteration of the algorithm, we maintain the invariant that each processor holds approximately the same amount of inequalities of each of the three categories (3), (4) and (5), i.e., that

$$n_1^{(k)} \approx n_1/p, \quad n_2^{(k)} \approx n_2/p, \quad \text{for each processor } k \tag{7}$$

where $n_1^{(k)}$ denotes the local part of processor k of the n_1 inequalities of category (3), and $n_2^{(k)}$ denotes the local part of processor k of the n_2 inequalities of categories (3) and (4). (7) also implies equal load balance for step 3.

Step 2 sorts the inequalities in order to enforce this invariant. This requires expensive communication. First, n_1 and n_2 have to be computed as

$$n_1 = \sum_{k=0}^{p-1} n_1^{(k)} \qquad \text{and} \qquad n_2 = \sum_{k=0}^{p-1} n_2^{(k)}. \tag{8}$$

Each processor inspects the sign of the coefficient of x_r and computes its local contribution to n_1 resp. n_2. The global values of n_1 and n_2 are then computed by two global summations and broadcast to each processor. Now each processor determines the amount of inequalities of each category that it will hold, and the amount of each category that it would like to send to resp. receive from other processors in order to maintain the invariant. After some global administrative work that matches senders with receivers, parallel interprocessor communication of $O(s)$ messages of length r each is required; this may be imbalanced. Altogether, the communication overhead due to step 2 is quite high, and can only partially be overlapped with the computational work following in step 3.

Creation of new inequalities in step 6 requires parallel broadcast of the smaller set of inequalities to all processors:

```
if (n1 < n2) {
   broadcast my n1/p inequalities from global range i=1,...,n1
   receive n1 - n1/p inequalities from the other processors
   /* now I hold all n1 inequalities in the range i=1,...,n1 */
   forall these i=1,...,n1
     forall my (n2-n1)/p inequalities i' from global range i'=n1+1,...,n2
       build combined inequality (i,i') according to step 6
}
else /* n1 >= n2 */ {
   broadcast my (n2-n1)/p inequalities from global range i=n1+1,...,n2
   receive (n2-n1) - (n2-n1)/p inequalities from the other processors
   /* now I hold all n2-n1 inequalities in the range i=n1+1,...,n2 */
   forall these i'=n1+1,...,n2
     forall my n1/p inequalities i from global range i=1,...,n1
       build combined inequality (i,i') according to step 6
}
```

Altogether, the communication overhead of step 6, consisting of $O(s)$ messages (equally distributed over the processors) of length r each, will be dominated by the computational work $O(rs^2)$ (again equally distributed) when building the new inequalities for the next iteration (if s is sufficiently large); thus communication could be tolerated for a modest number of processors.

4.2 Data Distribution, Second Variant

As in the first variant, we start with a cyclic distribution of the inequalities over the processors. Here, in contrast, we do not enforce the invariant (7) for step 3. Thus the computation performed in step 3 may be imbalanced since the $n_2^{(k)}$ are generally not equal for each processor k.

Step 2 now consists mainly of a local sorting of inequalities (which can be done in time $O(s/p)$). Computing n_1 and n_2 is done as above (8). Step 2 is thus very much cheaper here than for the first variant.

Nevertheless, for larger numbers of processors, we cannot expect that the $n_1^{(k)}$ and $n_2^{(k)}$ are approximately equal on each processor k. But we need such a distribution in order to balance the computational effort for generating the $O(s^2)$ new inequalities. As a consequence, the processors must (in parallel) broadcast all s inequalities, instead of only $\min(n_1, n_2 - n_1)$, in order to achieve invariant (7) immediately before step 6. This yields an equal distribution of the s' new inequalities across the processors for the new system, i.e. also for the next iteration. Step 6 thus implies a global redistribution of inequalities. The asymptotic communication overhead is $O(s/p)$ messages of size r each to be broadcasted/received by each processor; the same argument on amortizing communication against computation as given in the previous paragraph holds here again.

4.3 Data Distribution, Third Variant

We apply a cyclic distribution of the m variables over p processors. This makes step 2 and step 6 cheaper because the data dependencies are internalized.

However, before normalization (step 3), the processor holding the coefficients of x_r must broadcast the divisor to all other processors for each inequality. This communication can be partially overlapped with the normalization computation performed at step 3.

The most crucial problem is that processor load gets imbalanced for the last $p-1$ iterations which are probably the most expensive ones. As m is usually substantially smaller than n, and this ratio gets even more extreme in the course of the algorithm, this distribution scenario alone is unsuitable to utilize a massively parallel computer efficiently.

It may, however, be combined with the first or second variant, resulting in a two–dimensional data distribution, i.e. we organize the processors as a two–dimensional grid with extents pi (across the inequalities) and pj (across the variables), such that $pi \cdot pj = p$. Furthermore, the distribution across the variables has to be adapted before load balance would get substantially worse: For instance, just before the iteration eliminating x_{pj-1}, we halve pj, double pi, and redistribute the current system for the new processor grid. This could be integrated into the redistribution phase before step 6 of the second variant. For the first variant, it requires additional communication time $O(sr/p)$. Thereafter, load balance across the variables will remain acceptable for the following $pj/2$ iterations.

Further experiments will show which combination is most suitable here. We are currently implementing these alternatives using PVM on a workstation cluster. The results will be contained in the final version of the paper.

5 Conclusion

Fourier–Motzkin elimination is a 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, one is, in some cases, willing to take its longer run time into account, compared with other methods such as Simplex.

We have shown that it is possible to considerably speed up Fourier–Motzkin elimination by massively parallel processing, as well for shared memory as for distributed memory parallel machines. In this way, some medium–sized problem instances can now be tackled that would take too long on a uniprocessor machine but perform in acceptable time on a massively parallel machine.

This is particularly interesting for the class of problem instances that are typically encountered at restructuring of loop nests for automatic parallelization [Ban93]. There the number of variables, which is most critical for the run time behavior of Fourier–Motzkin elimination, is typically small, less than 6 in nearly all cases. Thus, why not use the parallel variant of Fourier–Motzkin elimination in the optimizing phase of a *parallel* compiler? Why should we still compile in sequential for a parallel machine, where we have the computational power available to get an exact solution? In this context, we regard this contribution as one of the first building blocks for a future *parallel parallelizer*, a project that we are currently planning for the SB-PRAM hardware platform.

References

[ADK+93] F. Abolhassan, R. Drefenstedt, J. Keller, W.J. Paul, and D. Scheerer. On the physical design of PRAMs. *Computer Journal*, 36(8):756–762, December 1993.

[AKP91] F. Abolhassan, J. Keller, and W.J. Paul. On the cost–effectiveness of PRAMs. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 2–9. IEEE, December 1991.

[Ban93] Utpal Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, 1993.

[DE73] G.B. Dantzig and B.C. Eaves. Fourier–Motzkin elimination and its dual. *Journal of Combinatorial Theory*, 14:288–97, 1973.

[Fou27] J.B.J. Fourier. (reported in:) Analyse des travaux de l'Académie Royale des Sciences pendant l'année 1824, Partie mathématique, 1827. Reprinted as: Second extrait, in: *Œuvres de Fourier, Tome II* (G. Darboux, ed.), Gauthier–Villars, Paris, 1890. English translation (partially) in: D.A. Kohler, *Translation of a report by Fourier on his work on linear inequalities*, *Opsearch* **10** (1973) 38–42.

[Kha79] L.G. Khachiyan. Polynomial algorithm for Linear Programming. *Doklady Akad. Nauk USSR*, 244(5):1093–96, 1979. Translated in Soviet Math. Doklady **20**, pp. 191–94.

[KM72] V. Klee and G.J. Minty. How Good is the Simplex Algorithm? In O. Shishna, editor, *Inequalities III.* Academic Press, New York, 1972.

[KS95] 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. See also: Technical Report 95-05, FB IV Informatik der Universität Trier, 1995. `http: //www-wjp. cs.uni-sb.de /fork95/`.

[Mot36] T.S. Motzkin. Beiträge zur Theorie der linearen Ungleichungen. (Inaugural Dissertation Basel), Azriel, Jerusalem, 1936. English translation: Contributions to the theory of linear inequalities, RAND Corporation Translation 22, Santa Monica, CA, 1952. Reprinted in *Theodore S. Motzkin: Selected Papers* (D. Cantor, B. Gordon, B. Rothschild, eds.), Birkhäuser, Boston, 1983, pp. 1–80.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1986.

[Wil76] H.P. Williams. Fourier–Motzkin elimination extension to integer programming. *Journal of Combinatorial Theory (A)*, 21:118–123, 1976.

[Wil83] H.P. Williams. A characterization of all feasible solutions to an integer program. *Discrete Applied Mathematics*, 5:147–155, 1983.