On the parallel complexity of Gaussian Elimination with Pivoting

M. Leoncini*

TR-94-028

August 1994

Abstract

Consider the Gaussian Elimination algorithm with the well-known Partial Pivoting strategy for improving numerical stability (GEPP). Vavasis proved that the problem of determining the pivot sequence used by GEPP is log space-complete for **P**, and thus inherently sequential. Assuming $\mathbf{P} \neq \mathbf{NC}$, we prove here that either the latter problem cannot be solved in parallel time $O(N^{1/2-\epsilon})$ or all the problems in **P** admit polynomial speedup. Here N is the order of the input matrix and ϵ is any positive constant. This strengthens the P-completeness result mentioned above. We conjecture that the result proved in this paper holds for the stronger bound $O(N^{1-\epsilon})$ as well, and provide supporting evidence to the conjecture. Note that this is equivalent to assert the asymptotic optimality of the naive parallel algorithm for GEPP (modulo $\mathbf{P} \neq \mathbf{NC}$).

*Dipartimento di Informatica, Università di Pisa, Pisa (Italy). Part of this work was done while the author was visiting the "International Computer Science Institute", Berkeley, CA. Support to the author's research has been given by the ESPRIT Basic Research Action, Project 9072 "GEPPCOM", and by the M.U.R.S.T. 40% and 60% funds.

1 Introduction

A fundamental research goal in the area of fast synchronous parallel algorithms is to obtain superpolynomial speedups in the time sufficient to solve given problems in **P**. Given a computational problem $\Pi \in \mathbf{P}$, the most ambitious aim is to put it in the complexity class **NC**, that is to find a parallel algorithm for Π whose running time is a polylogarithmic function of the input size on, e. g., a PRAM with polynomially many processors. There is now a rich literature on the complexity class **NC** (see [5, 14] for surveys and [15] for a general critique).

Recently, there has been much interest in identifying problems that, though probably not in **NC**, admit *at least polynomial speedup*. Vitter and Simons [18] identified a number of such problems (see also [15]). In addition, finding parallel algorithms that achieve *only polynomial speedup* can be interesting even for problems in **NC**. The reason is that polynomial speedup can usually be obtained with a limited number of processors (say, a linear or quadratic function of the input size), while the figures required to obtain superpolynomial speedups are in many cases not practical.

Together with the *algorithmic* interest, there is an obvious interest in finding complexity results. Assuming $\mathbf{P} \neq \mathbf{NC}$, one could try to classify problems in $\mathbf{P} - \mathbf{NC}$ with respect to the achievable speedup. For instance, [18] consider the class PC of problems that can be sped up by more than a constant factor. On the other hand, [15] focus on the problems that admit polynomial speedup, and classify these further with respect to their inefficiency[1]. They introduce the class **EP**, of problems solvable with constant inefficiency, and the class **SP**, of problems solvable with polynomial inefficiency.

We clearly do not know whether these new classes of problems actually differ from **NC**. However, [3] proves that there are P-complete problems that appear to have a bound on the amount of achievable speedup. Such problems are said *strictly $T(n)$-complete* for **P**, for some complexity function $T(n)$. More precisely, to say that a problem Π is strictly $T(n)$-complete amounts to saying that: (1) there is a parallel algorithm solving Π in time $T(n)$, and (2) either there is not a parallel algorithm for Π running in time $O(T(n)^{1-\epsilon})$, for any positive ϵ, or all the problems in **P** admit polynomial speedup. If only (2) can be proved, then Π is *at most $T(n)$-complete*. For all practical purposes (i.e. unless $\mathbf{P} \neq \mathbf{NC}$), proving that a certain problem is strictly $T(n)$-complete implies that its $T(n)$-time parallel algorithm is optimal.

The first problem complete for **P** in the stricter sense outlined above is the *Square Circuit Value Problems*, with $T(n) = n^{1/2}$ [3]. The technique used to prove this result is a generic reduction from an arbitrary RAM computation. However, one difficulty in finding other complete problems through the reduction argument is that a polynomial blowup in the size of the instances may not be acceptable (while clearly this is not the case in the proofs of P-completeness).

In this paper we consider the well-known Gaussian Elimination method for the computation of the LU decomposition of a square matrix. Actual implementations of this method adopt a simple strategy for row interchange, known as Partial Pivoting. The so modified algorithm is known as *Gaussian Elimination with Partial Pivoting* (hereafter referred to

[1] The inefficiency of an algorithm is the ratio pT_p/T, where T is the sequential running time, p is the number of processors and T_p is the parallel time with p processors.

as GEPP), and it is of fundamental importance in computational linear algebra and in the broad field of scientific computing [13]. In fact, when the LU decomposition is known, many problems can be solved with little additional cost. These include linear system solution, determinant and (with some special care) rank computation. It is well-know that, for matrices of order n, the parallel arithmetic complexity of the latter problems is $O(\log^2 n)$ (see [8, 1, 2, 12]). However, the algorithms that achieve this bound are not regarded as practical ones by the numerical analysis community. The reason lies in part in the large number of processors required, but mostly because they are considered numerically unstable.

Achieving numerical accuracy in finite precision computations seems to require a lot more of control than that provided by **NC** algorithms. Such control must be implemented using conditional statements which are in general hard to parallelize. Vavasis [17] proved that answering simple questions about the behavior of GEPP, such as whether a certain row i will be used to eliminate a given column j, is a log space-complete problem for the class **P**, and thus hardly in **NC**. Vavasis' reduction is from a version of the classical monotone circuit value problem, and holds for either exact rational arithmetic or decimal arithmetic rounded to a fixed number of decimal places. The latter model is clearly more realistic for practical numerical computations.

Here we prove that the decision version of GEPP addressed by Vavasis is at most $n^{1/2}$-complete for **P**. Our result holds for both the fixed and floating-point models of arithmetic. The result is a simple consequence of the following main Lemma, where we use the notation $M \in GEPP$ to state that the matrix M is one for which the question on the elimination order mentioned above is affirmative (GEPP as a decision problem is the set of all such matrices).

Main Lemma. *Let $t(n)$ and $s(n)$ be constructible functions, and let A be any RAM decision algorithm running in time $t(n)$ and using $s(n)$ memory registers. Then we can effectively build a square matrix M of order $k(n) = \tilde{O}(t(n)s(n))$ such that $M \in GEPP$ if and only if A accepts the input. The construction is NC computable.*

In the construction of the matrix M in the Main Lemma, we observe a blowup in the input size[2] which is polynomial (with respect to the running time of A) when the number of registers used by A is $O(((t(n))^{\epsilon})$. This is the reason why we are currently unable to prove the optimality (modulo $\mathbf{P} \neq \mathbf{NC}$) of the naive parallel algorithm for GEPP.

The rest of this paper is organized as follows. In Section 2 we give some preliminary definitions and discuss the computation models, sequential and parallel, adopted in this paper. In Section 3 we introduce a restricted model suitable for the simulation of Section 4 and study its relationships with the RAM. In Section 4 we prove that the computations of the restricted model can be encoded as instances of the Gaussian Elimination process with Partial Pivoting. Using this fact we prove our Main Lemma in Section 5. Finally, in Section 6 we report some concluding remarks.

2 Computation models

In this paper we predicate a lower bound on the parallel time required to solve a certain problem modulo the impossibility of obtaining polynomial speedup for the whole class of

[2] In case of GEPP we may take the order of the input matrix as the measure of size.

polynomial time solvable decision problems. This calls for a great deal of accuracy in the choice of the sequential computation model of reference, because the very notion of polynomial speedup is model sensitive. Let us therefore precisely nail down what we mean by speedup and polynomial speedup. Our definition is centered on the computations models involved, but it is not exclusively concerned with the gain in speed due to parallelism. Both models can very well be sequential models.

Definition 1 *Let P be a (decision) problem, and let M and M' be two computation models. Let A and A' be the fastest known algorithms for P running on M and M', respectively. Finally, let $t(n)$ and $t'(n)$ denote the running times of A and A', respectively. Assume that $t(n) = \Omega(t'(n))$. Then we call the ratio $S(n) = t(n)/t'(n)$ the* speedup *observed for P on M' over M. The inverse ratio $1/S(n)$ is the* slowdown *observed for P on M with respect to M'. We observe* polynomial speedup *when $S(n)$ is a polynomial function in $t(n)$, i.e. when $S(n) = \Omega(t(n)^{\epsilon})$, for some positive ϵ.*

When M' in Definition 1 is a parallel computation model, it is also to be intended that the amount of hardware resources available to M' (such as number of processors or circuit gates) is a polynomially bounded function of the input size n.

Clearly, the notion of polynomial speedup is not sensitive to polylogarithmic factors. In fact if there exists a positive ϵ such that $t(n)/t'(n) = \Omega(t(n)^{\epsilon})$, then for sufficiently large n,

$$\frac{t(n)}{t'(n)\left(\log t'(n)\right)^{O(1)}} \geq \alpha \left(t(n)\right)^{\delta},$$

for any positive δ less than ϵ and some positive δ. From this fact we obtain a large degree of freedom in the choice of the parallel computation model. In fact, it is well-known (see, e.g., [14]) that the running times on the various PRAMs are related by polylogarithmic factors. The same is true for PRAMs and uniform boolean circuits. In this paper we will adopt the (say, CREW) PRAM as our parallel computation model.

The choice of the sequential computation model requires a more careful handling. While the class **P** can be defined with respect to one of many reasonable sequential computation models, whether or not a problem exhibits polynomial speedup when solved on the PRAM will depend on the particular sequential model of reference. If M and M' are two sequential models, we might observe polynomial speedup on the PRAM over M but not over M'. Or we might observe polynomial speedup in both cases, but we very different polynomials. It follows that the computation model must not be too weak, for otherwise it would be possible that the speedup would be determined by such a weakness rather than by the power of parallelism. For instance, if M were the Turing Machine, then to observe polynomial speedup it would be sufficient in many cases just to pick a PRAM with a single processor.

The sequential model we will use is the classic RAM introduced by Cook and Reckhow [6]. This is a natural model of computation, and one widely adopted in the study of concrete algorithms. In this latter setting, one often assumes that the cost of performing an operation is a fixed constant, independent of the length of the operand(s) involved. This is the well-known *unit cost* criterion. However, for complexity-theoretic investigations, the *logarithmic cost* criterion (which is the one adopted in [6]) is usually regarded as more precise than the unit cost criterion.

The instruction set of the RAM is showed in Table 1. The table also shows the execution times charged to each instruction under the logarithmic cost criterion. The function $l(\cdot)$ is defined as follows (see [6]):

$$l(i) = \begin{cases} \lceil \log_2 |i| \rceil & \text{if } |i| \geq 2 \\ 1 & \text{otherwise.} \end{cases}$$

Instruction	Execution time
$R_i \leftarrow \alpha$	1
$R_i \leftarrow R_j$	$l(R_j)$
$R_i \leftarrow R_j \pm R_k$	$l(R_j) + l(R_k)$
$R_i \leftarrow R_{R_j}$	$l(R_j) + l(R_{R_j})$
$R_{R_i} \leftarrow R_j$	$l(R_{R_i}) + l(R_j)$
goto L	1
if $R_i \leq 0$ then *inst.*	$l(R_i)$(+ cost of *inst.* if $R_i \leq 0$)

Table 1: RAM instructions and execution times.

As for the space, which will play an important role in our reduction, the RAM introduced in [6] adopted a logarithmic cost criterion as well. According to such criterion, a cost is charged only to those registers that are accessed at some point during the computation. The amount charged to a register R_i is the maximum value of $l(x)$, over all integers x stored in R_i, and the overall space cost is the sum of all the costs charged to the used registers.

3 A restricted RAM model

There are two aspects that make it difficult for the computations generated by the RAM model discussed above to be encoded as instances of the Gaussian Elimination process. These are the (possible) lack of locality in the instruction flow and the use of indirect storage accesses.

Indirect addressing capabilities appear to be one of the features that should not be given up in a concrete (as well as reasonable) computation model. Here, however, we are not interested in concrete algorithm design. Our sole concern is to understand how exactly we loose if we eliminate the indirect addressing instructions from the set of Table 1. This is an interesting question per se, but unfortunately one that has not yet received a satisfying answer. Dymond [9] studied thoroughly this problem. He introduced the Augmented Counter Machine (ACM) model and studied its relationships with the RAM (among the others). An ACM, with k registers, can be viewed as a RAM without indirect addressing capabilities, but further restricted to add and subtract small amounts only. As a consequence of this last restriction, the values in the ACM registers change by at most a constant each step. Dymond proved that these ACMs could be simulated with *polynomial speedup* by log cost RAMs. The polynomial depends on the number of registers. For k registers and time $t(n)$ on the ACM, the RAM can accomplish the simulation in time $(t(n))^{(k+1)/(k+2)}$. Therefore, no fast simulation of the RAM by an ACM is possible. There must always be polynomial slowdown in view of the above result and the RAM time hierarchy proved by Cook and Reckhow.

Extending the simulation above to ACMs with full addition and subtraction seems possible [10]. If we accept this, we also accept that there must be polynomial slowdown in the simulation of unrestricted RAMs by RAMs without indirection capabilities.

In the rest of this section we introduce a model, that we call *restricted RAM*, or *rRAM* for short, that will be suitable for the simulation of Section 4, and study the slowdown incurred by such model with respect to the RAM. The result we obtain is an easy one. However, it appears to be difficult to obtain stronger bounds [4, 10].

Definition 2 *A uniform family $M_1, M_2, \ldots$ of ACMs with full addition and subtraction is a restricted RAM if: (1) M_n accepts inputs of length n only, (2) the computations generated by each M_n are oblivious of the actual inputs.*

To say that the computations generated by a machine are oblivious amounts to saying that the sequence of instructions executed by the n^{th} control program is fixed (i.e. it doesn't depend on the actual input). The uniformity condition that we place on the family is simply that the n^{th} machine (i.e. the program and the number k of registers used) can be NC computable. The time charging criterion for our rRAM will be the customary logarithmic measure. As for the space, this will be the number $k = k(n)$ of registers used.

In order to determine the slowdown incurred by RAM simulations on the rRAM model, we begin with a lemma on memory compaction which is an easy adaptation of a result in [3].

Lemma 1 *A RAM with space demand $s(n)$ can be restricted to access only cells whose addresses are $O(s(n))$ on input of length n, with only a loss of a factor $O(\log s(n))$ in the running time.*

Note that Cook and Reckhow proved that the the maximum number that can be generated (and thus the maximum address that can be referenced) by a $t(n)$ time bounded RAM is as large as $2^{O(\sqrt{t(n)})}$. As will be evident in the following lemma, the result of Lemma 1 implies a substantial improvement in the number of registers required for a rRAM to simulate RAM computations. Also, this improvement will turn in a substantial saving in the simulation time. In the next lemma we forget, for the moment, the requirement on the obliviousness of the rRAM computations.

Lemma 2 *A RAM with running time $t(n)$ and space demand $s(n)$ can be simulated with only polylogarithmic slowdown by an rRAM M': $M'_1, M'_2, \ldots$ such that the length of the program of M'_n (and therefore also the number of registers it uses) is $O(s(n))$.*

Proof By Lemma 1 we may assume that the original RAM only accesses the first $S = O(s(n))$ registers. We replace each indirect addressing instruction (i.e. indirect load or store) with a macro statement performing a binary search in the set of the first S registers. Figures 1 and 2 show the macros for the instruction $R_i \leftarrow R_{R_j}$ for $S \leq 2$ and $S \leq 4$, respectively. In the cited figures, the instruction $R_0 \leftarrow R_j - k$, where k is a constant, is a shorthand for the sequence $R_t \leftarrow k$, $R_0 \leftarrow R_j - R_t$, where R_t is a register not otherwise used by the program. We also assume that R_0 is never used by the program (but otherwise we can use any register not used by the original RAM program).

```
R_0 ← R_j − 1
      if R_0 ≤ 0 then goto L1
      R_i ← R_2
      goto EXIT
L1    R_i ← R_1
      goto EXIT
EXIT
```

Figure 1: Macro for $R_i \leftarrow R_{R_j}$ and $S \leq 2$

```
R_0 ← R_j − 2
      if R_0 ≤ 0 then goto L2
      R_0 ← R_j − 3
      if R_0 ≤ 0 then goto L3
      R_i ← R_4
      goto EXIT
L3    R_i ← R_3
      goto EXIT
L2    R_0 ← R_j − 1
      if R_0 ≤ 0 then goto L1
      R_i ← R_2
      goto EXIT
L1    R_i ← R_1
      goto EXIT
EXIT
```

Figure 2: Macro for $R_i \leftarrow R_{R_j}$ and $S \leq 4$

Note that the macro for the case $S \leq 2^k$ is actually a couple of macro for the case $S \leq 2^{k-1}$ stuck together, plus a couple of leading instructions. It is then easy to see that, for $S \geq 2$, the length of the macro statement is $2^{\lceil \log S(n) \rceil + 2} - 2$, which is clearly $O(s(n))$. It is also easy to see that the generation of the macro statement can be performed in space $O(log(s(n)))$, provided that $s(n)$ (or an upper bound $S = O(s(n))$ to $s(n)$) can be computed within this time bound. Since the length of the (original) program, and thus the number of indirect addressing instructions, is independent of n, the length of the rRAM program is $O(s(n))$.

As for the running time of the macro statement this is given by the cost of the simulated instruction (essentially, by the contents of the register R_{R_j}) plus the cost of the binary search. The latter is certainly $O(\log^2 s(n))$ because at most $O(\log s(n))$ instructions are executed, each one having cost $O(\log s(n))$. Clearly this implies an $O(\log^2 s(n))$ slowdown in the running time of the rRAM program with respect to the original RAM program. ∎

The last step consists in forcing the computations to be oblivious. Condon [3] proves the following result.

Lemma 3 *Any RAM with running time $t(n)$ can be simulated by an oblivious RAM with running time $O(t(n))$.*

The simulation does not make use of the indirect addressing instructions, and thus holds also in case of our rRAM. Also, the generation of the simulating (oblivious) program is NC computable. The constant hidden in the asymptotic notation in Lemma 3 contains the length of the program being simulated. Unfortunately, in our case this is not independent of n. In fact, by Lemma 2, it is $\tilde{O}(s(n))$.

Combining Lemmas 1, 2, and 3, and the last observation we get the following result.

Theorem 4 *Let M be any RAM that runs in time $t(n)$ using $s(n)$ space. Then there is a restricted RAM M': $M'_1, M'_2, \ldots$, such that the following hold: (i) M and M' accept the same language, (ii) the length of the program of M'_n is $O(s(n))$, (iii) M'_n has running time $\tilde{O}(t(n)s(n))$, (iv) M'_n can be generated in parallel time $O(\log s(n))$.*

The rRAM M' in Theorem 4 makes only use of the following instructions:

1. $R_i \leftarrow \alpha$;
2. $R_i \leftarrow R_j$;
3. $R_i \leftarrow R_i \pm R_j$;
4. **if** $R_k \leq 0$ **then** $R_j \leftarrow R_j \pm R_i$.

4 Simulating rRAM computations by means of Gaussian Elimination

Let $P = \{P_n\}_{n \in \mathbf{N}}$ be a decision problem in **P**. Here P_n are the instances of P of size n. Let A be a restricted RAM algorithm that solves P. A is actually a family of programs, $A = \{A_n\}_{n \in \mathbf{N}}$, such that A_n solves the instances in P_n. Assume that the running time of A_n is $t(n) \geq n$. Given a positive integer n and an input $I = I(n)$ for A_n, we describe how to build a square matrix $M(A_n, I)$ of order $O(t(n))$ such that the execution of Gaussian Elimination with Partial Pivoting on $M(A_n, I)$ simulates the execution of A_n on input I.

Arithmetic model

No definition of the behavior of GEPP is possible without a precise description of the arithmetic model. The reduction described in this section is from a generic rRAM computation to an instance of (the decision version of) "GEPP on the fixed point arithmetic model with truncation". A similar reduction, however, holds in case of a floating point system as well.

The set of numbers represented in a fixed point number system are

$$F_{\mu,l} = \{-l\mu, -(l-1)\mu, \ldots, -\mu, 0, \mu, \ldots, (l-1)\mu\},$$

for some positive μ and some positive integer l. μ is the *machine precision*. When the representation base is the usual binary one, with f bits before and d after the point (plus one sign bit), then $\mu = 2^{-d}$ and $l = 2^{f+d}$. It is also customary to assume $f = d$.

Let m be the minimum positive number in $F_{\mu,l}$ that has a multiplicative inverse in $F_{\mu,l}$. For instance, in the binary system outlined in the above paragraph m would be 2μ. We will require that

$$m^2 L^3 t(n) < \mu, \tag{1}$$

where $L = L(t(n))$ is the largest positive integer that can be generated by a $t(n)$ time bounded RAM. Clearly, in view of (1) and the fact that μ is the machine precision, the number $m^2 L^3 t(n)$ is a "machine zero". Equation (1) has implications on the word length. As already pointed out in Section 3, Cook and Reckhow [6] proved that the largest integer that can be generated by a $t(n)$ time bounded RAM has magnitude $2^{O(\sqrt{t(n)})}$. This implies that a word length polynomial in the input size is sufficient. We observe, however, that this bound can be greatly improved by using the following result, due to Wiedermann.

Lemma 5 *[19] A RAM R working within time $t(n)$ can be simulated, with only constant slowdown, by a RAM R' that uses integers of length $O(\log t(n))$ and addresses of value $O(t^2(n))$.*

Even if R' in the above Lemma can generate addresses of value $O(t^2(n))$, which is not good for our purposes, we can still apply the compression result of Lemma 1 to R'. By virtue of this result, even a word length logarithmic in the input size would be sufficient.

We will use the notation $\odot$ to denote the machine operation corresponding to the exact operation $\cdot$, where $\cdot \in \{+, -, \times, /\}$. It holds $x \cdot y = \mathrm{trn}(x \cdot y)$. Among the others, the following properties hold in the machine arithmetic outlined here.

1. $\mathrm{trn}(xm^2) = 0$, for any x generated by the rRAM computation.
2. $\mathrm{trn}((1 - xm)^{-1}) = \mathrm{trn}(1 + xm + (xm)^2 + \ldots) = 1 + \tau m$, for any x generated by the rRAM computation.
3. $\mathrm{trn}(\mu x) = 0$, if $x < 1$.

Construction of the matrix $M(A_n, I)$

We shall view the matrix $M(A_n, I)$ as a two dimensional program, and GEPP as the interpreter for it. Using this viewpoint, we will show how any given statement of the restricted RAM can be translated into a corresponding "statement" in $M(A_n, I)$. The matrix we will obtain is essentially block diagonal, having 0s almost everywhere outside the main (block) diagonal. See Figure 3.

Let $N = N(n)$ be the number of statements executed by the program A_n. Then, there will be $N + 1$ blocks along the main diagonal, numbered 0 to N. Blocks 1 through N will correspond to the instructions executed by A_n. The order of these blocks is either 2 or 9, the latter being the case of a block corresponding to an **if** statement. The order of block 0 is 1 plus the number of inputs (not to be confused with the length n of the input). The input conventions are that the number i of inputs is stored in register R_0, with the actual input stored in R_1 through R_i. The order of the matrix is thus at most $9N + i + 1$. Since $N \leq t(n)$

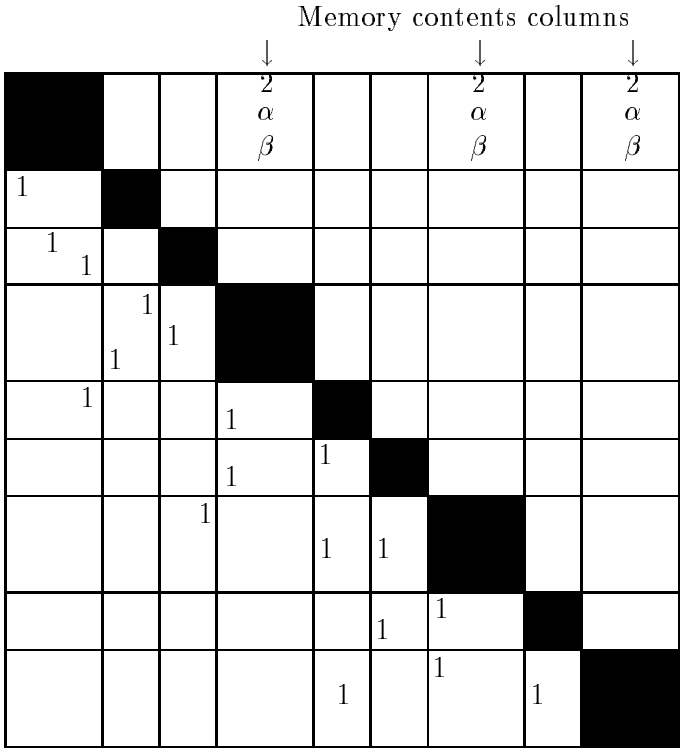

Figure 3: Structure of the matrix $M(A_n, I)$.

(each instruction costs at least 1 under the logarithmic cost criterion), and $i \leq n \leq t(n)$, the order of $M(A_n, I)$ is certainly $O(t(n))$.

In the lower triangular part of $M(A_n, I)$ there is a 1 somewhere in block (h, k) *if and only if*, for some register R_j, the hth and kth instructions of A_n use R_j (with no other intruction in the middle using this register). Such 1s implement the logical pipes between two consecutive intructions that make use of the same RAM register. In the upper triangular part of $M(A_n, I)$, there are nonzero entries only in those blocks $(0, j)$ such that the jth instruction executed by A_n is a conditional statement, $1 \leq j \leq N$. Such nonzero entries contain, from top to bottom, the number of input registers and the actual input (2, α, and β in the example of Figure 3). The columns corresponding to these entries will be called *memory contents* columns. In fact, during the execution of the Gaussian Elimination algorithm, certain entries in these columns will represent the rRAM storage. Observe that there is one memory contents column only if the corresponding diagonal block is a conditional statement block (these are the largest diagonal blocks in Figure 3). This depends on the fact that only the conditional statements need to peek at the storage.

The figures 4 to 7 show, enclosed in boxes, the diagonal blocks corresponding to the initialization phase (first diagonal block) and to the different statements of our restricted RAM. We use the notation $0(X)$ to indicate that one entry in a memory contents column contains 0 initially and X by the time the simulation of the corresponding instruction begins. For the instructions whose behavior does not depend on the values stored in the rRAM registers (i.e. all the instructions, except the conditional statement) we only show

one memory contents column. In case of the conditional statement, we show two such columns.

$$
\begin{array}{c}
R_0 \\ R_1 \\ R_2 \\ \vdots \\ R_2 \\ \vdots \\ R_0 \\ \vdots \\ R_1 \\ \vdots
\end{array}
\begin{pmatrix}
\boxed{\begin{array}{ccc} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{array}} & \cdots & 2 \\
\vdots & \cdots & \vdots \\
0 & \cdots & 0 \\
\vdots & \cdots & \vdots \\
1 & \cdots & 0 \\
\vdots & \cdots & \vdots \\
0 & \cdots & 0 \\
\vdots & \cdots & \vdots
\end{pmatrix}
$$

Figure 4: Initialization block (input in two registers)

$$
\begin{array}{c}
\vdots \\ R_i \\ \beta \\ \vdots \\ R_i \\ \vdots
\end{array}
\begin{pmatrix}
\cdots & \vdots & \vdots & \vdots & \vdots \\
\cdots & \boxed{\begin{array}{cc} -1 & 0 \\ 0 & -1 \end{array}} & \cdots & 0(\tilde{\alpha}) \\
\cdots & & \cdots & \tilde{\beta} \\
\cdots & \vdots & \vdots & \vdots & \vdots \\
\cdots & 0 & 1 & \cdots & 0 \\
\cdots & \vdots & \vdots & \vdots & \vdots
\end{pmatrix}
\qquad
\begin{array}{c}
\vdots \\ R_i \\ R_j \\ \vdots \\ R_j \\ \vdots \\ R_i \\ \vdots
\end{array}
\begin{pmatrix}
\cdots & \vdots & \vdots & \vdots & \vdots \\
\cdots & \boxed{\begin{array}{cc} -1 & 0 \\ 0 & -1 \end{array}} & \cdots & 0(\tilde{\alpha}) \\
\cdots & & \cdots & 0(\tilde{\beta}) \\
\cdots & \vdots & \vdots & \vdots & \vdots \\
\cdots & 0 & 1 & \cdots & 0 \\
\cdots & \vdots & \vdots & \vdots & \vdots \\
\cdots & 0 & 1 & \cdots & 0 \\
\cdots & \vdots & \vdots & \vdots & \vdots
\end{pmatrix}
$$

Figure 5: Assignments $R_i \leftarrow \beta$ and $R_i \leftarrow R_j$

$$
\begin{array}{c}
\vdots \\ R_j \\ R_i \\ \vdots \\ R_j \\ \vdots \\ R_i \\ \vdots
\end{array}
\begin{pmatrix}
\cdots & \vdots & \vdots & \vdots & \vdots \\
\cdots & \boxed{\begin{array}{cc} -1 & 0 \\ \pm 1 & -1 \end{array}} & \cdots & 0(\tilde{\beta}) \\
\cdots & & \cdots & 0(\tilde{\alpha}) \\
\cdots & \vdots & \vdots & \vdots & \vdots \\
\cdots & 1 & \cdots & \cdots & 0 \\
\cdots & \vdots & \vdots & \vdots & \vdots \\
\cdots & 0 & 1 & \cdots & 0(\alpha) \\
\cdots & \vdots & \vdots & \vdots & \vdots
\end{pmatrix}
$$

Figure 6: Assignment $R_i \leftarrow R_i \pm R_j$

Behavior of GEPP applied to $M(A_n, I)$

Proving that the execution of GEPP, on input the matrix $M(A_n, I)$ constructed according the above rules, does indeed simulate the execution of A_n on I amounts essentially to performing error analysis. However, we must be able to view into the elimination process

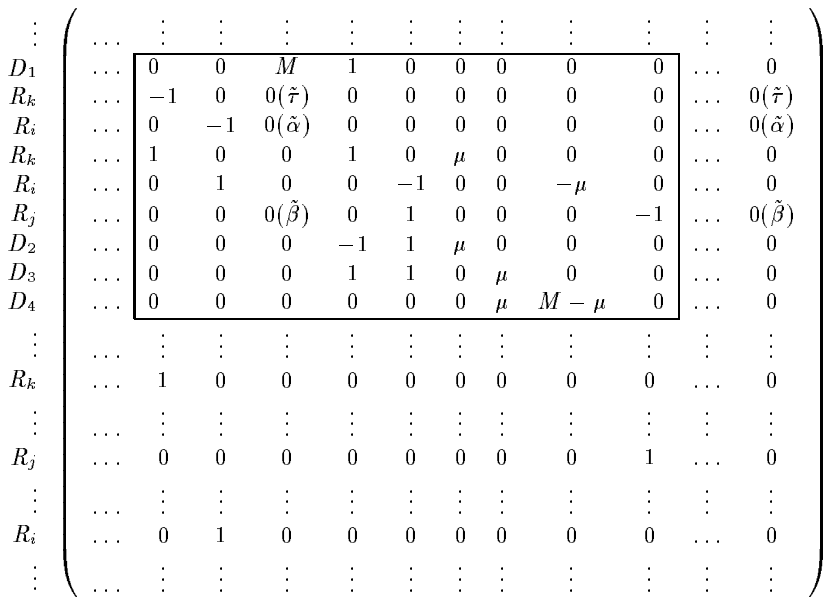

Figure 7: Initial configuration for: **if** $R_k \leq 0$ **then** $R_j \leftarrow R_j + R_i$

and spell out the similarities with the rRAM computation. To this end, we first introduce some basic terminology and recall the fundamental ideas behind Gaussian Elimination with Partial Pivoting (for more details on the latter, we refer the reader to the classic reference [13]).

- For $h = 1, \ldots, N+1$, we let $M^{(h)}$ stand for the the matrix resulting from the execution of the first $h-1$ *pivot steps*. Thus $M^{(1)} = M(A_n, I)$, while $M^{(N+1)}$ is the final (triangular) matrix. Moreover, we use $a_{ij}^{(h)}$ for the i,j^{th} entry of $M^{(h)}$.

- The Gaussian Elimination algorithm with Partial Pivoting consists of a sequence of N pivot steps. Each consists of three phases.

  1. *Pivot row selection.* The pivot row for the h^{th} step is the one whose index i satisfies
  $$i = \min\{j \geq h : |a_{jh}^{(h)}| \leq |a_{lh}^{(h)}|, l = h, h+1, \ldots\}.$$
  Note that we adopt the usual strategy of choosing the lowest indexed row among those with entries of maximum absolute value in column h [3].

  2. *Row exchange.* Once the pivot row has been selected, the algorithm exchanges it with row h. This is usually done by simply exchanging the row indexes, kept in a separate array.

[3] This is by no means a loss of generality, as long as we will restrict to deterministic strategies for pivot selection. On the other hand, our simulation will not work in case of a randomized strategy for breaking ties.

3. *Submatrix update.* The last phase consists of updating the submatrix made of the i, j^{th}th entries, for $i > h$ and $j \geq h$. As is well-known, this is done by means of linear combinations with the pivot row, in such a way that the entries in position (i, h), for $i > h$, are set to zero.

- For $k = 1, \ldots, N$, we let S_k denote the k^{th} instruction executed by A_n. The simulation of S_k will be accomplished by the execution of a set of pivot steps. The pivot elements for the simulation of S_k will always be taken from the diagonal block number k. We will regard this set of pivot steps as the k^{th} *stage* of the simulation. This definition makes sense, since, as we will see, stage k will be completed before stage $k+1$ begins, $k = 1, \ldots, N-1$. We will also regard the elimination process performed on the first diagonal block as the 0^{th} stage of the simulation, even if this does not correspond to any program statement.

- Let $\{h_k\}_{k=0,\ldots,N}$ be the set of indexes corresponding to the first rows (and columns) of the diagonal blocks. For instance, $h_0 = 1$, $h_1 = i + 2$, while h_2 through h_N depend on the program A_n. For $k = 0, \ldots, N$, we classify the rows of the matrix $M^{(h_k)}$ according to their role in the elimination process.

  - A row that has been used as the pivot row in a step $s \leq h_k$ is *dead* at stage k.
  - A non-dead row that has already been modified by a previous pivot operation is *living* at stage k. Initially, we assume that all the rows that form block 0 are living.
  - A non-dead and non-living row is *unborn* at stage k.

The idea behind the simulation is that certain rows of the matrix correspond to the rRAM registers. In general, there is more than one row corresponding to a given register. This is required by the fact that, as soon as is selected in a pivot step, a row will not be used any more (i.e. it becomes dead). Therefore, another row corresponding to the same register must somehow come into play.

For any $k \in \{0, 1, \ldots, N\}$, there are two invariant conditions that hold before the execution of stage k of the simulation.

1. For any register R_j of the simulated rRAM there is *at most one* living row corresponding to it in the matrix $M^{(h_k)}$, all the others being unborn or dead.

2. Let R_i be any register of the rRAM, and let l be the index of the living row corresponding to R_i in $M^{(h_k)}$. Let the *mvalue* of R_i at stage k, or simply $mv(R_i)$ if the stage is understood, denote the (common) value in the entries at the intersection between row l and the memory contents columns with index $\geq h_k$. The invariant condition states that

$$mv(R_i) = (1 + C_{n,k}m)x_i,$$

   where x_i is the value in R_i before the execution of the k^{th} instruction of the program, and $|C_{n,k}| \leq kL$. Note that $C_{n,k}$ is a quantity that depends on the program A_n, the stage number k, but not the index register i.

Before stage 0 begins, the two invariant conditions are satisfied by the initial matrix $M^{(0)}$, with $C_{n,0} = 0$.

Below we describe the behavior of GEPP with respect to the different blocks that correspond to the instructions in the set of the rRAM. We assume that the registers used are R_i, R_j, and R_k, and that the values stored in these registers before the execution of the k^{th} instruction are α, β, and τ, respectively. We assume that the corresponding mvalues are $\tilde{\alpha} = \alpha(1 + C_{n,k}m)$, $\tilde{\beta} = \beta(1 + C_{n,k}m)$, and $\tilde{\tau} = \tau(1 + C_{n,k}m)$, and prove that the invariant conditions continue to hold after the execution of stage k.

- *Initialization* (refer to Figure 4). Performing the Gaussian Elimination on the first diagonal block will cause the mvalues of the registers R_0 through R_i to be copied where they are first needed, i.e. into the next lowest indexed rows corresponding to those registers. As a consequence of this, the first rows corresponding to $R_0, \ldots, R_j$ become dead, while the others change their status from unborn to living. The two invariant conditions are thus easily met. In particular, $C_{n,1} = C_{n,0} = 0$.

- $R_i \leftarrow \beta$ (refer to Figure 5). The first step has the only purpose of killing the currently living row corresponding to R_i (whose contents will be overridden by β). The second step makes $\tilde{\beta}$ to be the new mvalue of R_i. The two conditions are met with $C_{n,k+1} = C_{n,k}$.

- $R_i \leftarrow R_j$ (see Figure 5). As in the case of constant assignment, the first step has the only purpose of killing the currently living row corresponding to R_i. The second step copies the mvalue of R_j into the next lowest indexed rows corresponding to R_i (to get the assignment done) and to R_j (to save it for another operation involving R_j). Also in this case the two invariant conditions are easily met, again with $C_{n,k+1} = C_{n,k}$.

- $R_i \leftarrow R_i \pm R_j$ (see Figure 6). As the result of the first step, the mvalue $\tilde{\beta}$ of R_j is: (1) copied to the first unborn row corresponding to R_j, and (2) added to (subtracted from) the mvalue $\tilde{\alpha}$ of R_i. The second pivot operation copies the mvalue $\tilde{\alpha} + \tilde{\beta}$ of R_i into the first unborn row corresponding to the same register. The first invariant condition is clearly met. The second is also satisfied. In fact

$$\begin{aligned} \tilde{\alpha} \oplus \tilde{\beta} &= (1 + C_{n,k}m)\alpha \oplus (1 + C_{n,k}m)\beta \\ &= \mathrm{trn}((1 + C_{n,k}m)(\alpha + \beta)) \\ &= (1 + C_{n,k}m)(\alpha + \beta), \end{aligned}$$

  and therefore $C_{n,k+1} = C_{n,k}$. One would be tempted here to replace the second row corresponding to R_i with the first, and remove the occurrence of the first. In this way the simulation would require just one single step. For uniformity of description, however, this is not advisable. In fact, with our solution, the generation of the portion of the matrix corresponding to any given instruction that uses the register R_i only requires to know the first next instruction that uses R_i. This would not be the case with the second solution.

- **if** $R_k \leq 0$ **then** $R_j \leftarrow R_j + R_i$. The simulation of the conditional statement is performed by means of 9 pivot steps. Among the rows involved, there are four that do not

correspond to any rRAM register. In the figures, these rows have labels $D_1, \ldots, D_4$. Initially (see Figure 7), the mvalue of R_k is copied to the first two unborn rows for the same register. The first copy will be used during the simulation of the conditional statement. The second copy is to transmit the value of R_k (which is not to be changed by the execution of the conditional) to the next row where it's needed. This latter will be the only living row corresponding to R_k by the end of stage k. The purpose of the second steps is similar (with R_i instead of R_k). The matrix resulting after the first two steps have been performed is depicted in Figure 8. The largest number in the column being eliminated (a memory contents column) is M. In fact, using the invariant condition on $C_{n,k}$ and the assumption on M, we easily see that, for any index i,

$$\begin{aligned} |mv(R_i)| &= |R_i|(1 + |C_{n,k}|m) \\ &\leq L(1 + kLm) \\ &< 2L \\ &< M. \end{aligned}$$

Applying the pivot step to the matrix of Figure 8 leads to the matrix of Figure 9. The only modified entries are those in column $h_k + 3$. For instance, the entry in position $h_k + 3, h_k + 3$ is determined using the machine arithmetic as follows:

$$\begin{aligned} 1 \ominus 1 \otimes (\tilde{\tau} \otimes (1 \oslash M)) &= 1 \ominus 1 \otimes ((\tau + \tau C_{n,k} m) \otimes m) \\ &= 1 \ominus 1 \otimes \mathrm{trn}(\tau m + \tau C_{n,k} m^2) \\ &= 1 \ominus 1 \otimes (\tau m) \\ &= 1 - \tau m. \end{aligned}$$

There are two possible choice for the pivot element in the matrix of Figure 9, and the choice depends on the value τ contained in register R_k before the execution of the conditional statement.

Assume first that $\tau > 0$ (i.e. that the test condition is not satisfied). In this case the pivot is taken from the row with label D_2 in Figure 9. Applying the pivot step leads to the matrix of Figure 10, and a further step to the matrix of Figure 11. The next 3 pivot steps applied to the matrix of Figure 11 do not result in the execution of linear combinations because the only non zero element in the columns being eliminated is the pivot itself. The last pivot step simply copies the mvalue of R_j (which is unchanged) to the next place where it is needed, and leads to the final matrix $M^{(h_{k+1})}$, depicted in Figure 12. The overall sequence of elimination is thus

$$R_k, R_i, D_1, D_2, D_3, R_k, D_4, R_i, R_j.$$

$$
\begin{array}{c}
\vdots \\
R_k \\
R_i \\
D_1 \\
R_k \\
R_i \\
R_j \\
D_2 \\
D_3 \\
D_4 \\
\vdots \\
R_k \\
\vdots \\
R_j \\
\vdots \\
R_i \\
\vdots
\end{array}
\begin{pmatrix}
\ddots & & & & & & & & & & & \\
 & -1 & 0 & \tilde{\tau} & 0 & 0 & 0 & 0 & 0 & 0 & \dots & \tilde{\tau} \\
 & 0 & -1 & \tilde{\alpha} & 0 & 0 & 0 & 0 & 0 & 0 & \dots & \tilde{\alpha} \\
 & 0 & 0 & \boxed{M} & 1 & 0 & 0 & 0 & 0 & 0 & \dots & 0 \\
 & 0 & 0 & \tilde{\tau} & 1 & 0 & \mu & 0 & 0 & 0 & \dots & \tilde{\tau} \\
 & 0 & 0 & \tilde{\alpha} & 0 & -1 & 0 & 0 & -\mu & 0 & \dots & \tilde{\alpha} \\
 & 0 & 0 & \tilde{\beta} & 0 & 1 & 0 & 0 & 0 & -1 & \dots & \tilde{\beta} \\
 & 0 & 0 & 0 & -1 & 1 & \mu & 0 & 0 & 0 & \dots & 0 \\
 & 0 & 0 & 0 & 1 & 1 & 0 & \mu & 0 & 0 & \dots & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 0 & \mu & M-\mu & 0 & \dots & 0 \\
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
 & 0 & 0 & \tilde{\tau} & 0 & 0 & 0 & 0 & 0 & 0 & \dots & \tilde{\tau} \\
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \dots & 0 \\
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
 & 0 & 0 & \tilde{\alpha} & 0 & 0 & 0 & 0 & 0 & 0 & \dots & \tilde{\alpha} \\
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots
\end{pmatrix}
$$

Figure 8: Simulation of conditionals: step 3

$$
\begin{array}{c}
\vdots \\
R_k \\
R_i \\
D_1 \\
R_k \\
R_i \\
R_j \\
D_2 \\
D_3 \\
D_4 \\
\vdots \\
R_k \\
\vdots \\
R_j \\
\vdots \\
R_i \\
\vdots
\end{array}
\begin{pmatrix}
\ddots & & & & & & & & & & & \\
 & -1 & 0 & \tilde{\tau} & 0 & 0 & 0 & 0 & 0 & 0 & \dots & \tilde{\tau} \\
 & 0 & -1 & \tilde{\alpha} & 0 & 0 & 0 & 0 & 0 & 0 & \dots & \tilde{\alpha} \\
 & 0 & 0 & M & 1 & 0 & 0 & 0 & 0 & 0 & \dots & 0 \\
 & 0 & 0 & 0 & \boxed{1-\tau m} & 0 & \mu & 0 & 0 & 0 & \dots & \tilde{\tau} \\
 & 0 & 0 & 0 & -\alpha m & -1 & 0 & 0 & -\mu & 0 & \dots & \tilde{\alpha} \\
 & 0 & 0 & 0 & -\beta m & 1 & 0 & 0 & 0 & -1 & \dots & \tilde{\beta} \\
 & 0 & 0 & 0 & \boxed{\text{-1}} & 1 & \mu & 0 & 0 & 0 & \dots & 0 \\
 & 0 & 0 & 0 & 1 & 1 & 0 & \mu & 0 & 0 & \dots & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 0 & \mu & M-\mu & 0 & \dots & 0 \\
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
 & 0 & 0 & 0 & -\tau m & 0 & 0 & 0 & 0 & 0 & \dots & \tilde{\tau} \\
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \dots & 0 \\
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
 & 0 & 0 & 0 & -\alpha m & 0 & 0 & 0 & 0 & 0 & \dots & \tilde{\alpha} \\
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots
\end{pmatrix}
$$

Figure 9: Simulation of conditionals: step 4 (two possible pivots)

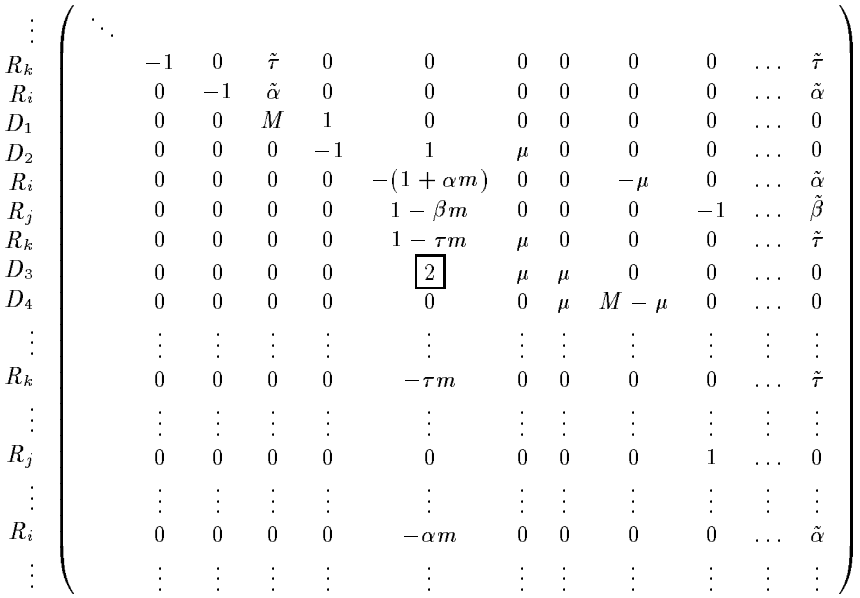

Figure 10: Simulation of conditionals: step 5 ($R_k > 0$)

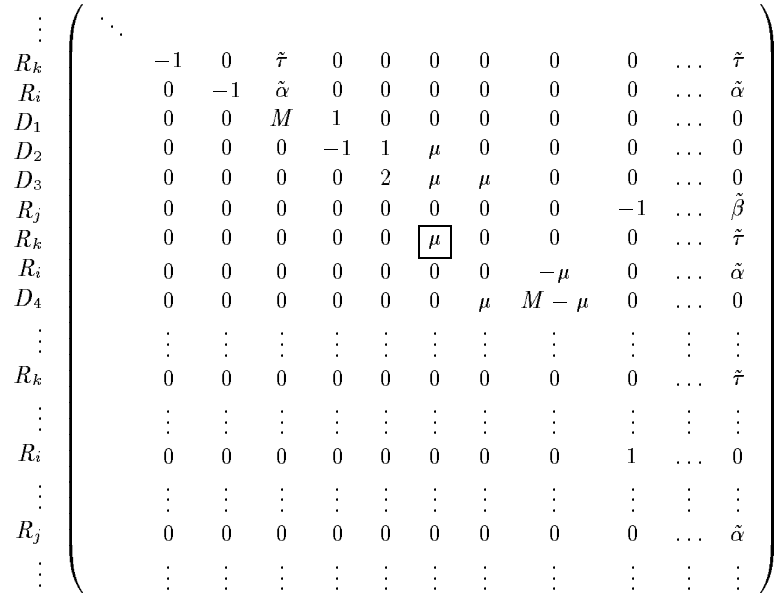

Figure 11: Simulation of conditionals: step 6 ($R_k > 0$)

$$
\begin{array}{c}
\vdots \\
R_k \\
R_i \\
D_1 \\
D_2 \\
D_3 \\
R_k \\
D_4 \\
R_i \\
R_j \\
\vdots \\
R_k \\
\vdots \\
R_i \\
\vdots \\
R_j \\
\vdots
\end{array}
\begin{pmatrix}
\ddots & & & & & & & & & & & \\
 & -1 & 0 & \tilde{\tau} & 0 & 0 & 0 & 0 & 0 & 0 & \ldots & \tilde{\tau} \\
 & 0 & -1 & \tilde{\alpha} & 0 & 0 & 0 & 0 & 0 & 0 & \ldots & \tilde{\alpha} \\
 & 0 & 0 & M & 1 & 0 & 0 & 0 & 0 & 0 & \ldots & 0 \\
 & 0 & 0 & 0 & -1 & 1 & \mu & 0 & 0 & 0 & \ldots & 0 \\
 & 0 & 0 & 0 & 0 & 2 & \mu & \mu & 0 & 0 & \ldots & 0 \\
 & 0 & 0 & 0 & 0 & 0 & \mu & 0 & 0 & 0 & \ldots & \tilde{\tau} \\
 & 0 & 0 & 0 & 0 & 0 & 0 & \mu & M-\mu & 0 & \ldots & 0 \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\mu & 0 & \ldots & \tilde{\alpha} \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & \ldots & \tilde{\beta} \\
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots & \tilde{\tau} \\
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots & \tilde{\beta} \\
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots & \tilde{\alpha} \\
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots
\end{pmatrix}
$$

Figure 12: Simulation of conditionals: final matrix ($R_k > 0$)

Now we backup to the matrix of Figure 9, and assume that the test condition is satisfied, i.e. that $\tau \leq 0$. In this case the pivot is taken from the row with label R_k. Applying the pivot step leads to the matrix of Figure 13. The entries denoted by X are of no interest to us (they do not affect the simulation process). It is easy to see, however, that they satisfy $|X| < 4L$. The important fact here is the way the mvalues of the registers change. Using the machine equality $1 \oslash (1 - \tau m) = 1 + \tau m$, we have, in case, e.g., of register R_i

$$
\begin{aligned}
mv(R_i) &\leftarrow \tilde{\alpha} \ominus \tilde{\tau} \otimes (-\alpha m \otimes (1 + \tau m)) \\
&= (\alpha + \alpha C_{n,k} m) \ominus (\tau + \tau C_{n,k} m) \otimes \mathrm{trn}(-\alpha m - \alpha \tau m^2) \\
&= (\alpha + \alpha C_{n,k} m) \ominus (\tau + \tau C_{n,k} m) \otimes (-\alpha m) \\
&= (\alpha + \alpha C_{n,k} m) \ominus \mathrm{trn}(-\alpha \tau m - \alpha \tau C_{n,k} m^2) \\
&= (\alpha + \alpha C_{n,k} m) \ominus (-\alpha \tau m) \\
&= \mathrm{trn}((\alpha + \alpha C_{n,k} m) + \alpha \tau m) \\
&= \alpha(1 + (C_{n,k} + \tau) m) \\
&= \alpha(1 + C_{n,k+1} m),
\end{aligned}
$$

where $C_{n,k+1} = C_{n,k} + \tau$. Note that $|\tau| \leq L$, hence $|C_{n,k+1}| \leq |C_{n,k}| + L \leq (k+1)L$.

The Figures 14 through 18 illustrate the rest of the elimination process in the case of successful test. The entries in position $h_k + 6, h_k + 5$ and $h_k + 7, h_k + 5$ may have one of two values depending on whether τ is zero or strictly less than zero. However, this does not affect the order of elimination. The pivot step performed on the matrix depicted in Figure 16 is potentially very dangerous, because it involves the row already

containing the updated mvalue of R_j. However, since $|X| < 8L$, the value actually added to this row is a machine 0. The elimination order for the case $\tau \leq 0$ is thus

$$R_k, R_i, D_1, R_k, R_i, D_2, D_3, D_4, R_j.$$

$$\begin{array}{c}
\vdots \\
R_k \\
R_i \\
D_1 \\
R_k \\
R_i \\
R_j \\
D_2 \\
D_3 \\
D_4 \\
\vdots \\
R_k \\
\vdots \\
R_j \\
\vdots \\
R_i \\
\vdots
\end{array}
\begin{pmatrix}
\ddots & & & & & & & & & & & \\
 & -1 & 0 & \tilde{\tau} & 0 & 0 & 0 & 0 & 0 & 0 & \dots & \tilde{\tau} \\
 & 0 & -1 & \tilde{\alpha} & 0 & 0 & 0 & 0 & 0 & 0 & \dots & \tilde{\alpha} \\
 & 0 & 0 & M & 1 & 0 & 0 & 0 & 0 & 0 & \dots & 0 \\
 & 0 & 0 & 0 & 1-\tau m & 0 & \mu & 0 & 0 & 0 & \dots & \tilde{\tau} \\
 & 0 & 0 & 0 & 0 & \boxed{\text{-1}} & 0 & 0 & -\mu & 0 & \dots & \alpha(1+C_{n,k+1}m) \\
 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & \dots & \beta(1+C_{n,k+1}m) \\
 & 0 & 0 & 0 & 0 & 1 & \mu/2\mu & 0 & 0 & 0 & \dots & X \\
 & 0 & 0 & 0 & 0 & 1 & 0/-\mu & \mu & 0 & 0 & \dots & X \\
 & 0 & 0 & 0 & 0 & 0 & 0 & \mu & M-\mu & 0 & \dots & 0 \\
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & \tau(1+C_{n,k+1}m) \\
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \dots & 0 \\
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & \alpha(1+C_{n,k+1}m) \\
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots
\end{pmatrix}$$

Figure 13: Simulation of conditionals: step 5 ($R_k \leq 0$)

$$\begin{array}{c}
\vdots \\
R_k \\
R_i \\
D_1 \\
R_k \\
R_i \\
R_j \\
D_2 \\
D_3 \\
D_4 \\
\vdots \\
R_k \\
\vdots \\
R_j \\
\vdots \\
R_i \\
\vdots
\end{array}
\begin{pmatrix}
\ddots & & & & & & & & & & & \\
 & -1 & 0 & \tilde{\tau} & 0 & 0 & 0 & 0 & 0 & 0 & \dots & \tilde{\tau} \\
 & 0 & -1 & \tilde{\alpha} & 0 & 0 & 0 & 0 & 0 & 0 & \dots & \tilde{\alpha} \\
 & 0 & 0 & M & 1 & 0 & 0 & 0 & 0 & 0 & \dots & 0 \\
 & 0 & 0 & 0 & 1-\tau m & 0 & \mu & 0 & 0 & 0 & \dots & \tilde{\tau} \\
 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -\mu & 0 & \dots & \alpha(1+C_{n,k+1}m) \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\mu & -1 & \dots & (\alpha+\beta)(1+C_{n,k+1}m) \\
 & 0 & 0 & 0 & 0 & 0 & \boxed{\mu/2\mu} & 0 & -\mu & 0 & \dots & X \\
 & 0 & 0 & 0 & 0 & 0 & 0/-\mu & \mu & -\mu & 0 & \dots & X \\
 & 0 & 0 & 0 & 0 & 0 & 0 & \mu & M-\mu & 0 & \dots & 0 \\
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & \tau(1+C_{n,k+1}m) \\
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \dots & 0 \\
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & \alpha(1+C_{n,k+1}m) \\
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots
\end{pmatrix}$$

Figure 14: Simulation of conditionals: step 6 ($R_k \leq 0$)

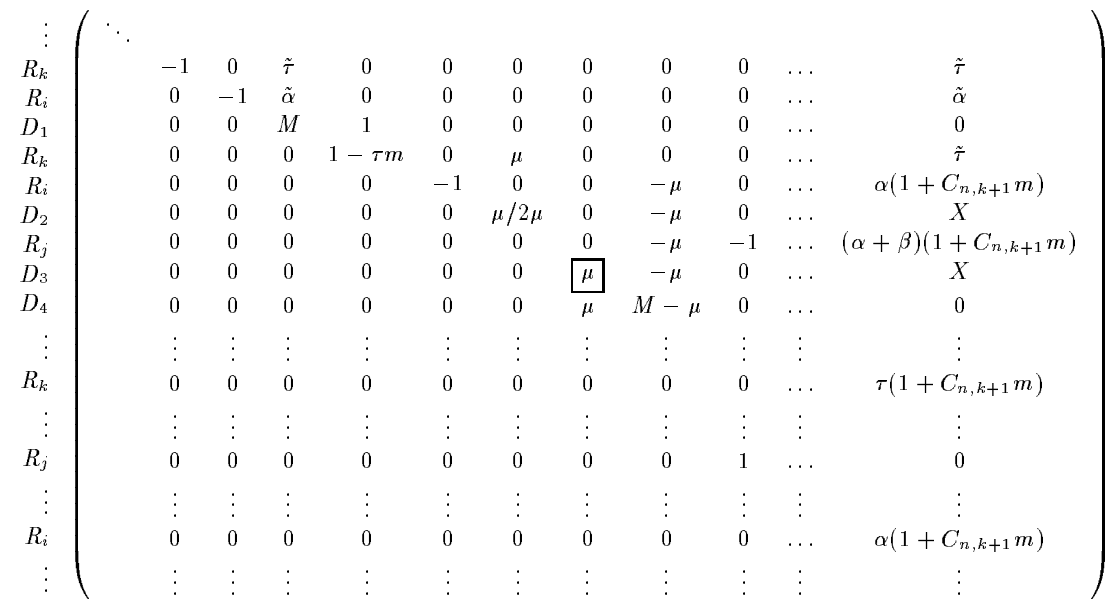

Figure 15: Simulation of conditionals: step 7 ($R_k \leq 0$)

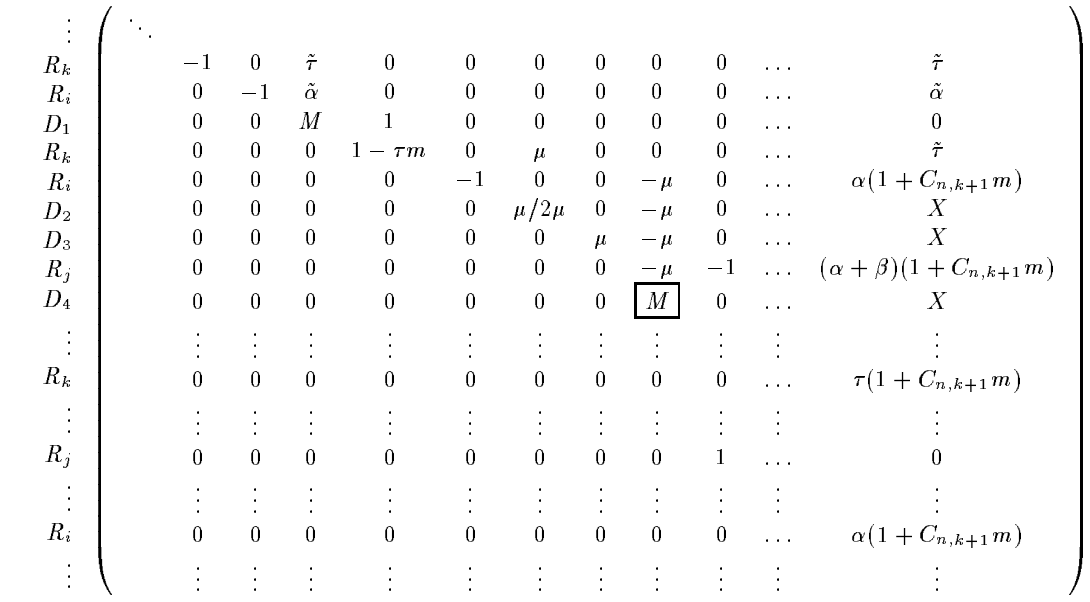

Figure 16: Simulation of conditionals: step 8 ($R_k \leq 0$)

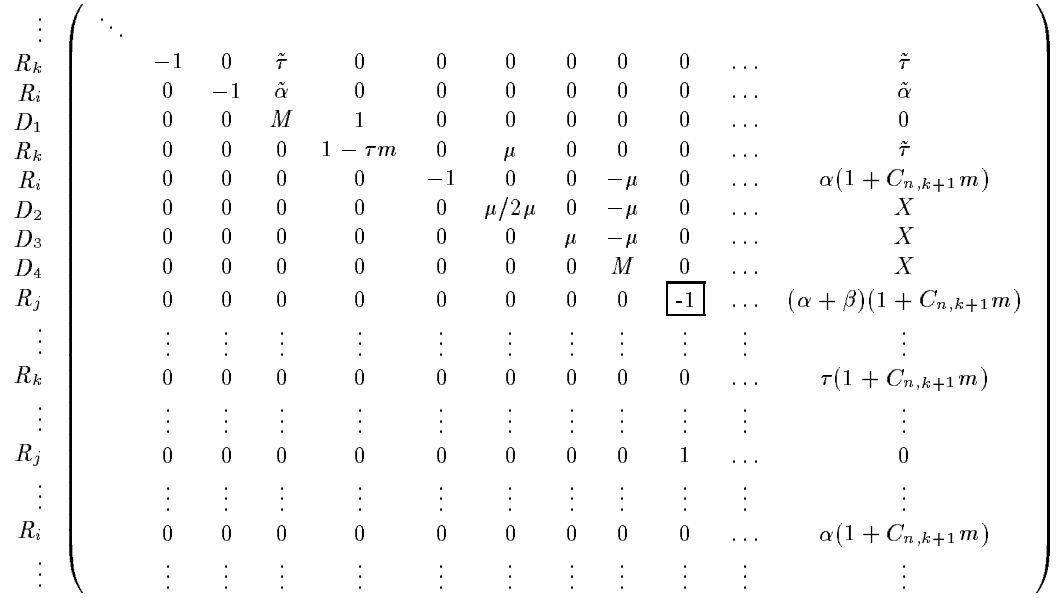

Figure 17: Simulation of conditionals: step 9 ($R_k \leq 0$)

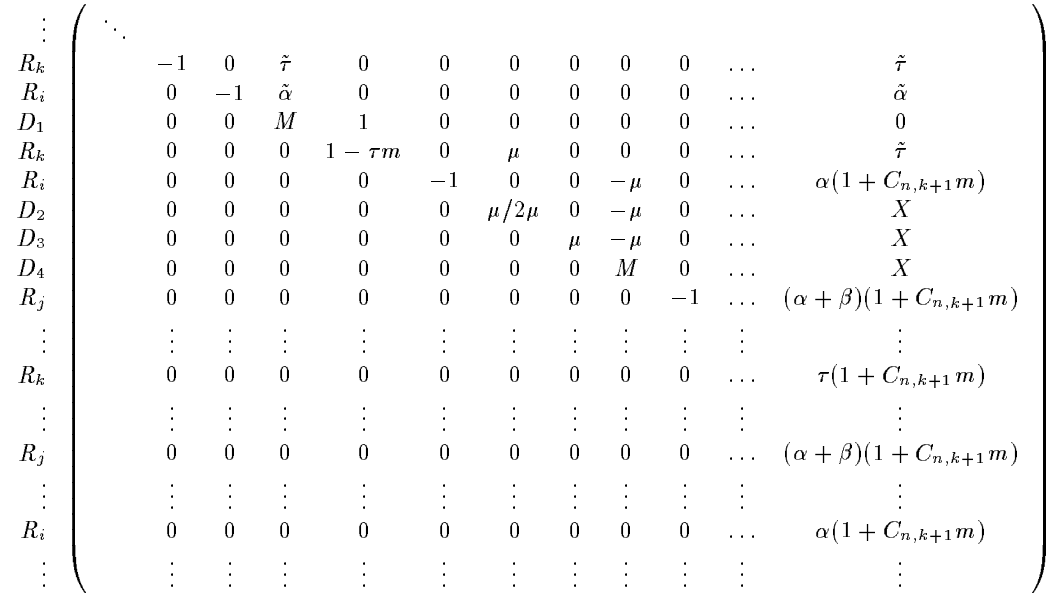

Figure 18: Simulation of conditionals: final matrix ($R_k \leq 0$)

5 A hardness result for Gaussian elimination

Using the simulation of Section 4, we are now ready to prove our main Lemma. According to [17], we formulate Gaussian Elimination as a language recognition problem in the following way:

> Given a matrix A and indexes i and j, will the Gaussian Elimination algorithm with Partial Pivoting use the pivot in (initial) row i to eliminate column j?

Lemma 6 (Main Lemma) *Let $t(n)$ and $s(n)$ be space constructible functions bounded by some polynomials in n, and let A be any RAM decision algorithm running in time $t(n) \geq n$ and using $s(n)$ memory registers. Finally, let I be an input for A, such that $|I| = n$. Then we can effectively build a matrix M of order $\tilde{O}(t(n)s(n))$, with $O(t(n))$ bit entries, such that $M \in GEPP$ if and only if A accepts the input. The construction can be accomplished in space $O(\log s(n))$ and thus is in* **NC**.

Proof We first convert the program A into a restricted RAM program A'_n that accepts inputs of size n and that does not make use of indirect addressing instructions. From Theorem 4, we know that the length of A'_n is $O(s(n))$ and that the slowdown in the running time is polylogarithmic. Also, according to Lemma 2, the work space required for the construction of A'_n is $O(\log s(n))$.

Since A'_n is restricted, the sequence of instructions executed is oblivious of the actual input. Assume that the output bit of A_n (indicating acceptance or rejection) is stored in register R_0 at the end of A'_ns execution. We now modify the program A_n in the following way. Let R_i be any register that has been written to by the program. Then we insert, as the new last instruction, the conditional statement "if $R_0 \leq 0$ then $R_i \leftarrow R_i$". In this way, the question of acceptance by A can be restated as a question on whether the test expression $R_0 \leq 0$ is satisfied. Call the resulting program A''_n. Observe also that the running time of A''_n is within a constant factor from the running time of A'_n.

From A''_n we build the matrix $M = M(A''_n, I)$ according to the rules of Section 4. The construction can be computed in space $O(\log s(n))$. Actually, each diagonal block can be generated in constant space. The only problem is to put the 1s in the lower triangular part of the matrix. To do this, during the generation of the kth diagonal block it is necessary to determine, for any register R_j used by that instruction, the sequence number of the next instruction using R_j and the index of the first row of the block corresponding to it. This information can be easily gathered by a linear search through the input program A''_n.

Clearly, the construction of M cannot be accomplished, as suggested above, in two distinct steps, because the programs A'_n and A''_n require space $O(s(n))$ simply to be stored. However, $M(A)$ can still be generated in $O(\log s(n))$ space, since the log space reduction is transitive.

Since each instruction can be simulated by a constant number of pivot operations, the order of M is at most a constant times the running time of A''_n. By Theorem4 this is $\tilde{O}(t(n)s(n))$. Using the simulation of Section 4, we conclude that the program A''_n, and thus A, accepts I if and only if to eliminate, e.g., column $N-2$ the GEPP algorithm uses row N. ∎

Theorem 7 *Let N denote the order of the input matrices. Then Gaussian Elimination with Partial Pivoting is at most $N^{1/2}$ complete for* **P**.

Proof Let Π be any problem in **P**, and let A be a RAM decision algorithm for Π running in time $t(n)$, on inputs of size n. By Lemma 6 the question of acceptance by A is NC-reducible to Gaussian Elimination with Partial Pivoting on a matrix of size $O(t^2(n))$ and entries of length $O(t(n))$. Therefore, any algorithm solving the Gaussian Elimination problem in time $O(N^{1/2-\epsilon})$, would provide, combined with the reduction algorithm, a decision procedure for Π running in parallel time $O(t^{1-2\epsilon})$, thus giving polynomial speedup. ∎

6 Concluding remarks

The result provided in this paper is not the tightest possible. It could be still possible to devise a parallel algorithm for GEPP running in time, say, $N^{2/3}$ without having dramatic consequences on the whole class **P**. We conjecture that this is not the case and thus that the existing "gap" depends on our current inability to prove the optimality (assuming $\mathbf{P} \neq \mathbf{NC}$) of the naive parallel implementation of Gaussian Elimination with Partial Pivoting.

One reason for believing this is that our Main Lemma says a little more than the at-most $N^{1/2}$ completeness of GEPP. To see this, consider a sequential algorithm A that solves a P-complete problem Π. Suppose that A achieves the best running time $t(n)$ known for Π and, moreover, that A uses substantially less space than $t(n)$, say $s(n) = (t(n))^{1-\epsilon}$, for some positive ϵ. In this case, we would obtain polynomial speedup over A if we were able to exhibit a decision procedure for GEPP running in parallel time $O(N^{\frac{1-\epsilon}{2-\epsilon}})$. For instance, for $\epsilon = 1/2$, $O(N^{2/3})$ parallel time would be sufficient (although, obviously, not sufficient to conclude that Π admits polynomial speedup, because A might not be the fastest sequential algorithm for A).

One computational problem with the above mentioned properties seems to be the decision version of the maximum flow in sparse graphs with n nodes. To the best of author's knowledge, there is currently no parallel algorithm running in time $O(n^{2-\epsilon})$ for this problem. However, we have sequential algorithms running in time $O(n^2)$ and using $O(n)$ space, provided that the number m of edges is $O(n)$ and the capacities on the edges are small size integers (see, e.g. the textbook [7]). Therefore we have $s(n) = (t(n))^{1-\frac{1}{2}}$, i.e. $\epsilon = 1/2$ in the argument of the above paragraph.

Acknowledgements

It is a pleasure to acknowledge the helpful comments and conversations that I had with Bruno Codenotti and Anne Condon on the subject of this paper.

References

[1] A. Borodin, J. von zur Gathen, and J. Hopcroft. Fast parallel matrix and GCD computations. *Inform. and Control*, 52:241–256, 1982.

[2] B. Codenotti and M. Leoncini. *Parallel Complexity of Linear System Solution*. World Scientific Pu. Co., Singapore, 1991.

[3] A. Condon, A Theory of Strict P-completeness. *Proc. STACS 92*, Lecture Notes in Computer Science, 577:33–44.

[4] A. Condon. Personal communication.

[5] S.A. Cook A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2–22, 1985.

[6] S.A. Cook and R.A. Reckhow. Time Bounded Random Access Machines. *Journal of Computer and System Sciences*, 7:354–375, 1973.

[7] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press/McGraw-Hill, 1990.

[8] L. Csanky. Fast parallel matrix inversion algorithms. *SIAM J. Comput.*, 5:618–623, 1976.

[9] P.W. Dymond. Indirect Addressing and the Time Relationships of Some Models of Sequential Computation. *Comp. and Math. with Appl.*, 5:193–209, 1979.

[10] P.W. Dymond. Personal communication through E-mail.

[11] P. van Emde Boas, Machine Models and Simulations. In: J. van Leeuwen (ed.). *Handbook of Theoretical Computer Science, Vol. A*. The MIT Press/Elsevier, 1990, 3–66.

[12] J. Von zur Gathen, Parallel Linear Algebra. In: J. Reif (ed.), *Synthesis of Parallel Algorithm*, Morgan and Kaufmann Publishers, San Mateo, CA, 1993, 573–617.

[13] G. H. Golub and C. F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, 1989.

[14] R.M. Karp and V. Ramachandran. Parallel Algorithms for Shared-Memory Machines. In: J. van Leeuwen (ed.). *Handbook of Theoretical Computer Science, Vol. A*. The MIT Press/Elsevier, 1990, 869–941.

[15] C.P. Kruskal, L. Rudolph, and M. Snir, A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 71:95–132, 1990.

[16] F. Meyer auf der Heide. Lower bounds for solving linear Diophantine equations on Random Access Machines. *J. ACM*, 32:929–937, 1985.

[17] S.A. Vavasis. Gaussian Elimination with Pivoting is P-complete. *SIAM J. Disc. Math.*, 2:413–423, 1989.

[18] J.S. Vitter and R.A. Simons. New classes for parallel complexity: a study of unification and other complete problems. *IEEE Trans. Comput.*, 35:403–418, 1986.

[19] J. Wiedermann. Normalizing and Accelerating RAM Computations and the Problem of Reasonable Space Measures. *Proc. of 17th ICALP, Lecture Notes in Computer Science*, 433:125–138, 1990.