How much can we speedup Gaussian Elimination with Pivoting? *

M. Leoncini

Dipartimento di Informatica
Università di Pisa, Italy

Abstract

Consider the problem of determining the pivot sequence used by the Gaussian Elimination algorithm with Partial Pivoting (GEPP). Let N stand for the order of the input matrix and let ϵ be any positive constant. Assuming $\mathbf{P} \neq \mathbf{NC}$, we prove that if GEPP were decidable in parallel time $N^{1/2-\epsilon}$ then all the problems in $\mathbf{P}$ would be characterized by polynomial speedup. This strengthens the P-completeness result that holds of GEPP. We conjecture that our result is valid even with the exponent 1 replaced for 1/2, and provide supporting arguments based on our result. This latter improvement would demonstrate the optimality of the naive parallel algorithm for GEPP (modulo $\mathbf{P} \neq \mathbf{NC}$).

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 $\mathbf{P}$. Given a computational problem $\Pi \in \mathbf{P}$, the most ambitious aim is to put it in the complexity class $\mathbf{NC}$, 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 $\mathbf{NC}$ (see [2, 7] for surveys and [8] for a general critique).

Recently, there has been much interest in identifying problems that, though probably not in $\mathbf{NC}$, admit *at least polynomial speedup*. Vitter and Simons [11] identified a number of such problems (see also [8]). In addition, finding parallel algorithms that achieve *only polynomial speedup* can be interesting even for problems in $\mathbf{NC}$. 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.

*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.

SPAA 94 - 6/94 Cape May, N.J, USA
© 1994 ACM 0-89791-671-9/94/0006..$3.50

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, [11] consider the class PC of problems that can be sped up by more than a constant factor. On the other hand, [8] focus on the problems that admit polynomial speedup, and classify these further with respect to their inefficiency*. They introduce the class $\mathbf{EP}$, of problems solvable with constant inefficiency, and the class $\mathbf{SP}$, of problems solvable with polynomial inefficiency.

We clearly do not know whether these new classes of problems actually differ from $\mathbf{NC}$. However, [1] 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 $\mathbf{P}$, 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 $\mathbf{P}$ 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 $\mathbf{P}$ in the stricter sense outlined above is the *Square Circuit Value Problems*, with $T(n) = n^{1/2}$ [1]. 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 address the problem of computing the sequence of pivots chosen by the Gaussian Elimination algorithm with Partial Pivoting (hereafter referred to as GEPP). Pivoting provides an example of the need of control in reliable numerical computations. Such control must be implemented using conditional statements which are in general hard to parallelize. In fact, it is well-known that GEPP is P-complete [10] and thus hardly in $\mathbf{NC}$. Here we prove that GEPP is at most $n^{1/2}$-complete for $\mathbf{P}$. This is a simple consequence of the main Lemma. Before stating the result, we recall that the decision (or language recognition) version of GEPP is a set, namely the set of matrices for which the al-

*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.

gorithm returns a 'yes' answer to a question concerning the elimination order (see Section 4).

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 $k(n) = t(n)s(n)$ 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[†] which is polynomial (with respect to the running time of A) when the number of registers used by A is $O(((t(n))^e)$. 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 discuss the RAM computation model and present certain results that allow to simplify the model to some extent. In Section 3 we show that any computation of a restricted RAM can be simulated by Gaussian Elimination with Partial Pivoting. Using this simulation result we prove the Main Lemma in Section 4. Finally, in Section 4 we report some concluding remarks.

2 Preliminary facts

The computation model we adopt is the RAM introduced by Cook and Reckhow [3], using the logarithmic cost criterion for time. For what concerns space, we adopt the uniform cost model, i.e. we charge unit space for each register used during the computation, independently of the number of bits stored. We view RAMs as language acceptors and assume suitable conventions for the input/output (the output being simply one bit). Table 1 shows the instruction set of the RAM together with the execution times. The function $l(\cdot)$ is defined as follows (see [3]):

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

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.

Before being able to apply our reduction of Section 3, we must simplify the model in various ways. We begin with a lemma on memory compaction which is an easy adaptation of a result in [1].

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.*

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

Lemma 1 means that, if we do not care about logarithmic slowdowns, we may assume that the programs make use of initial segments of the RAM storage. For our purposes we can safely make this assumption. This is because a function polynomially smaller than $t(n)$ is also polynomially smaller than $t(n)\log^k t(n)$, for any constant k. In the following, we use the customary notation $\tilde{O}(t(n))$ to denote $O(t(n))(\log t(n))^{O(1)}$.

We now consider a RAM model without indirect addressing capabilities. The question of the power of such a model is related to the issue of uniformity. A uniform RAM (say, a RAM whose control program is independent of n) without indirect addressing is essentially a counter machine equipped with full addition. Dymond gave evidence that counter machines augmented with the capability of adding at most a constant at each step, which he called Augmented Counter Machines (ACMs), are polynomially weaker than RAMs [5]. More precisely, he proved that a $T(n)$ time bounded ACM with k registers can be simulated by a RAM running in time $O((T(n))^{(k+1)/(k+2)})$. Also, extending his simulation to ACMs with full addition seems possible [6]. On the other hand, if we assume a nonuniform RAM model without indirect addressing, the loss in the running time is at most polylogarithmic.

Lemma 2 *A RAM with running time $t(n)$ and space demand $s(n)$ can be simulated by a nonuniform RAM without indirect addressing capabilities with only polylogarithmic slowdown. The length of the program of the simulating RAM is $O(s(n))$.*

Proof By Lemma 1 we may assume that the original RAM only accesses the first S 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. The macro statement leaves the result in a fixed register, not otherwise used by the program. It can be easily shown that the length of each macro statement is $\Theta(s(n))$, and clearly the number of such statements is independent of n. The running time of each macro statement is $O(\log^2 s(n))$ ($O(\log s(n))$ in the unit cost RAM, see also [9]) plus the cost of the simulated instruction. Clearly this implies a $O(\log^2 s(n))$ slowdown. ∎

Following [1], we say that a RAM is restricted if the computations it executes are oblivious of the actual input (i.e. if the kth instruction executed depends only on k and the input size n). The following fact is known (see [1]).

Lemma 3 *Any RAM with running time $t(n)$ can be simulated by a restricted RAM with running time $O(t(n)L)$, where L is the length of the program.*

Lemma 3 holds for the general instruction set, and thus also for a program which does not make use of indirection.

Combining Lemmas 1, 2, and 3 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 nonuniform restricted RAM M' such that the following hold: (i) M and M' accept the same language, (ii) the length of the program of M' is $O(s(n))$, (iii) M' has running time $\tilde{O}(t(n)s(n))$.*

The RAM 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** $R_j \leftarrow R_j \pm R_i$.

3 Simulating RAM computations by means of Gaussian elimination

Let A be a RAM decision algorithm of time complexity $t(n)$, and let I be an input for A. We assume that A is oblivious and does not make use of indirection. We describe how to build a matrix $M(A, I)$ such that the execution of Gaussian Elimination with Partial Pivoting on $M(A, I)$ simulates the execution of A on I. The model of arithmetic we adopt is a reasonable one. We assume that the GEPP is performed using a fixed point number system, with minimum absolute value m and maximum absolute value M. The value M must satisfy the inequality $M > t(n)2^{c\sqrt{t(n)}}$, where $2^{c\sqrt{t(n)}}$ is the magnitude of the largest number that can be generated by a $t(n)$ time bounded RAM [3]. We also suppose that $m = M^{-1}$‡, and that the operations are performed with rounding (rather than truncation). It is not difficult to see that our result holds, with minor modifications, also if we assume a floating point number system.

We view the matrix $M(A, I)$ as a two dimensional "program", and the GEPP algorithm as the interpreter for it. Using this viewpoint, we show how any given statement of our restricted RAM model can be translated into a corresponding "statement" in $M(A, I)$. The matrix we obtain is essentially block diagonal, having 0s almost everywhere outside the main (block) diagonal. See Figure 1.

Let N be the number of statements executed by the program A. Then the matrix $M(A, I)$ is block $(N+2)\times(N+2)$, with block indexes ranging from 0 to $N+1$. The order of the first diagonal block equals the number of RAM registers that hold the input I to the program. We assume, as customary, that the input is contained in consecutive registers $R_1, \ldots, R_i$, with the number i stored in register R_0. The last diagonal block has order 1. Blocks 1 through N have orders 2 or 10. The order of the matrix is thus at most $10N+i+2$. For $k = 1, \ldots, N$, the kth (diagonal) block represents the kth statement of the program A. Block 0 serves for initialization purposes. The last diagonal element is not involved in the elimination process.

In the lower triangular part of $M(A, I)$ there is a 1 somewhere in block (h, k) iff, for some register R_j, the hth and kth intructions of A 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 $M(A, I)$, there are nonzero entries only in the block $(0, N+1)$, and in any block $(0, j)$, for $1 \leq j \leq N$, such that the jth instruction executed by the program is a conditional statement. Such nonzero entries (represented by x in Figure 1) contain the input values stored in the first consecutive RAM registers.

‡In real finite arithmetic usually $m^{-1} > M$. Our result is not affected by the assumption that m has no multiplicative inverse. We may use any pair of representable numbers m' and M' such that $m' = M'^{-1}$, e.g. $m' = 2m$ provided that the representation base is an even number, with now M' strictly greater than $t(n)2^{c\sqrt{t(n)}}$

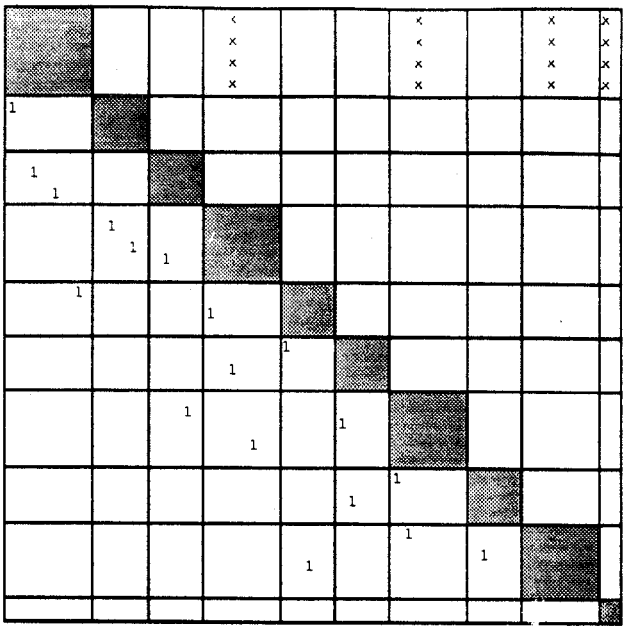

Figure 1: Structure of the matrix $M(A, I)$.

$$
\begin{array}{c}
R_0 \\ R_1 \\ R_2 \\ \vdots \\ R_2 \\ \vdots \\ R_0 \\ \vdots \\ R_1 \\ \vdots
\end{array}
\begin{pmatrix}
-1 & 0 & 0 & \ldots & 2 \\
0 & -1 & 0 & \ldots & \alpha \\
0 & 0 & -1 & \ldots & \beta \\
\vdots & \vdots & \vdots & \ldots & \vdots \\
0 & 0 & 1 & \ldots & \vdots \\
\vdots & \vdots & \vdots & \ldots & \vdots \\
1 & 0 & 0 & \ldots & \vdots \\
\vdots & \vdots & \vdots & \ldots & \vdots \\
0 & 1 & 0 & \ldots & \vdots \\
\vdots & \vdots & \vdots & \ldots & \vdots
\end{pmatrix}
$$

Figure 2: Initialization block (input in two registers)

$$
\begin{array}{c}
\vdots \\ R_j \\ R_i \\ \vdots \\ R_j \\ \vdots \\ R_i \\ \vdots
\end{array}
\begin{pmatrix}
\ldots & \vdots & \vdots & \vdots & \vdots \\
\ldots & -1 & 0 & \ldots & 0(\beta) \\
\ldots & 0 & -1 & \ldots & 0(\alpha) \\
\ldots & \vdots & \vdots & \vdots & \vdots \\
\ldots & 1 & 0 & \ldots & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
\ldots & 1 & 0 & \ldots & 0 \\
\ldots & \vdots & \vdots & \vdots & \vdots
\end{pmatrix}
$$

Figure 3: Assignment $R_i \leftarrow R_j$

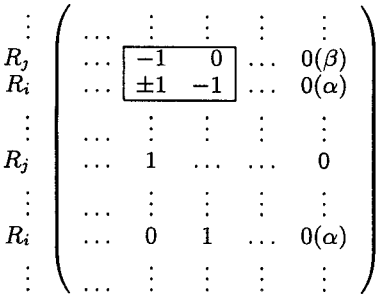

Figure 4: Assignment $R_i \leftarrow R_i \pm R_j$

The figures 2 to 5 show (enclosed in boxes) the diagonal blocks corresponding to the initialization phase (block 0) and to the three statements of our restricted RAM. We use the notation $0(X)$ to indicate that one entry contains 0 initially and X by the time the simulation of the instruction begins. These entries are called *memory contents*, because the value X is related to the value stored by a certain RAM register (denoted by the label on the left of the matrix) at some point during the computation. The memory contents entries correspond (i.e. have the same column index) to the entries denoted by x in Figure 1.

We now prove informally that the execution of the GEPP algorithm on the matrix $M(A, I)$ does indeed simulate the execution of A on I. To do this we need introduce some basic terminology. We let S denote the current instruction, and assume that S is the kth instruction executed by A. Therefore, the simulation of S is performed by the elimination process on the rows corresponding to the kth diagonal block. When we refer to the matrix $M(A, I)$ at the step h of the GEPP algorithm, we intend the state of $M(A, I)$ immediately before the execution of the hth pivot operation.

At any given step h, we classify the rows of the matrix $M(A, I)$ according to their role in the elimination process.

1. A row that has been used as the pivot row in a step $s < h$ is *dead* at time h.
2. A non-dead row that has already been modified by a previous pivot operation is *living* at step h. However, initially, we assume that all the rows in block 0 are living. The *active part* of a living row is the set of entries with column indexes not less than than h.
3. A non-dead and non-living row is *unborn* at time h.

The idea behind the simulation is that certain rows of the matrix correspond to the RAM registers used by A. In general, there is more than one row corresponding to a given register. However, at any given step of the elimination process, and for any register used by A, there is exactly one living row corresponding to that register (all the other rows being dead or unborn). Assume that the register R_j is used by the hth instruction and subsequently by the kth instruction of A. Then, within the simulation of the two instructions the living row corresponding to R_j will be the one in the kth block.

The overall simulation process consists of a sequence of *pivot operations*. Let $a_{ij}^{(h)}$ denote the entries of $M(A, I)$ before the hth pivot step is performed. Then the pivot row for the hth step is the ith iff the index i satisfies:

$$i = \min\{j : |a_{jh}^{(h)}| \leq |a_{lh}^{(h)}|, l = h, h+1, \ldots\}.$$

Note that we adopt the usual strategy of choosing the topmost row, among those with entries of maximum absolute value in column h [§]. Once the pivot row has been selected, the GEPP algorithm axchanges it with row h. This is usually done by simply exchanging the row indexes, kept in a separate array. The last phase of the pivot operation consists of updating the submatrix made of the i, jth 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. As a consequence of this operation, some unborn rows may change their status to living.

To illustrate the simulation of an instruction S, we assume that the registers used by S are R_i, R_j, and (in case of the if statement) R_k, and that the subsequent instructions that involve these registers are the qth, rth, and sth, respectively. With these assumptions, for each kind of instruction, we show the entries of the kth diagonal block and the positions of the 1s in the blocks (q, k), (r, k), and (possibly) (s, k) in the lower triangular part of the matrix. Moreover, we always show the memory contents entries in the last column of the matrix.

The invariant conditions that hold at the end of the simulation of the kth instruction, for $k \leq N$, are the following: (1) for any index $j \geq 0$ there is at most one living row corresponding to the RAM register R_j (this is the row with mimimum index among those in the blocks $k+1, k+2, \ldots, N$); (2) the memory contents entries of such row contain the value aX, where X is the value stored in R_j after the execution of the kth instruction by the program A, and a is a value that depends on A, I, and k, but not on the register R_j.

- $R_i \leftarrow R_j$ (see Figure 3). The assignment is simulated by two pivot operations. The first operation copies the memory contents of the living row corresponding to R_j to the first two unborn rows corresponding to R_i and R_j (if any). After this operation has been performed there are (possibly) two living rows corresponding to R_i, but the one in block k has old memory contents. This row is forced to become dead with the second pivot operation (that leaves all the other rows unchanged).
- $R_i \leftarrow R_i \pm R_j$ (see Figure 4). Also this simulation is carried out with two pivot operations. As the result of the first step, the value β in the memory contents of the living row corresponding to R_j are: (1) copied to first unborn row corresponding to R_j (if any), and (2) added to (subtracted from) the value α in the memory contents of the living row corresponding to R_i. The second pivot operation copies the value $\alpha + \beta$ in the memory contents of R_i to the first unborn row corresponding to the same register (if any). Note that if $\alpha = aX$ and $\beta = aY$, then $\alpha + \beta = a(X + Y)$, and so also the invariant condition (2) is satisfied.
- **if** $R_k \leq 0$ **then** $R_j \leftarrow R_j + R_i$ (see Figures 5 to 8). The simulation of the conditional statement is performed by means of ten pivot steps. Among the rows

[§]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.

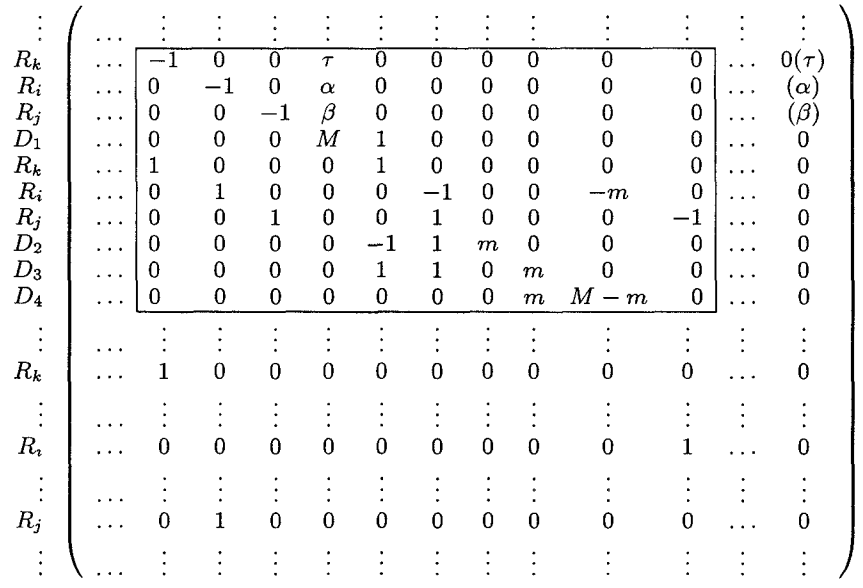

Figure 5: Initial configuration for: **if** $R_k \leq 0$ **then** $R_j \leftarrow R_j + R_i$

involved, there are four that do not correspond to any RAM register. In the figures, these rows are labelled $D_1, \ldots, D_4$. Here the sequence of pivot rows chosen by the Gaussian elimination algorithm depends on the memory contents. If the test condition is not satisfied the elimination sequence is

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

and otherwise is

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

In the first case ($R_k > 0$), after the simulation has been completed, the memory contents of the living rows corresponding to R_k, R_i, and R_j (Figure 7) coincide with the memory contents of the rows that were living before the simulation (Figure 5).

In the second case ($R_k \leq 0$), the memory contents of all the living rows have been scaled by the same value $1 - \tau m$ (Figure 8). To prove that the invariant condition (2) is satisfied, assume that, for all indexes i, the value stored in the RAM register R_i before the execution of the instruction S by A is α_i. Assume also that, by the time the simulation of S begins, the value in the memory contents entries of the living row corresponding to R_i is $\tilde{\alpha}_i = \alpha_i(1+Cm)$, where C is an integer. Then, when the simulation of S is completed, the memory contents of the living row corresponding to R_i

$$\begin{aligned} \tilde{\alpha}_i(1-\tau m) &= \alpha_i(1+Cm)(1-\alpha_k(1+Cm)m) \\ &= \alpha_i(1+Cm)(1-\alpha_k m) \\ &= \alpha_i(1+(C-\alpha_k)m). \end{aligned}$$

In the above derivation we have used our rounding mechanism, according to which, if a and b are integers, $(a + bm)m = am$. The invariant condition is thus satisfied.

The fact that, as the simulation proceeds, the memory contents can be scaled by a fixed quantity (when a conditional statement is executed whose condition is satisfied) does not affect the result of successive evaluations of test conditions. To see this, observe that the scaling factor after the execution of Q conditional statements is at least $1 - QLm$, where L is the magnitude of the largest integer generated during the RAM computation. Since Q is at most $t(n)$, it follows that $QL < M$ and thus $1 - QLm$ is always positive.

4 A hardness result for Gaussian elimination

Using the simulation of Section 3, we are now ready to prove our main Lemma. According to [10], 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 5 (Main Lemma) *Let A be any RAM decision algorithm running in time $t(n)$ and using $s(n)$ memory registers (with $t(n)$ and $s(n)$ constructible functions). Let I be an input for A, such that $|I| = n$. Then we can effectively build a matrix $M = M(A, I)$ of order $O(t(n)s(n))$, 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 $O(\log t(n))$.*

Proof We first convert the program A into a restricted RAM program A' that does not make use of indirect addressing. From Theorem 4 we know that the length of A' is $O(s(n))$ and the slowdown in the running time $O(\log^3 s(n))$.

$$
\begin{array}{c}
\vdots \\
R_k \\
R_i \\
R_j \\
D_1 \\
R_k \\
R_i \\
R_j \\
D_2 \\
D_3 \\
D_4 \\
\vdots \\
R_k \\
\vdots \\
R_i \\
\vdots \\
R_j \\
\vdots
\end{array}
\begin{pmatrix}
\cdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\cdots & -1 & 0 & 0 & \tau & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & \tau \\
\cdots & 0 & -1 & 0 & \alpha & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & \alpha \\
\cdots & 0 & 0 & -1 & \beta & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & \beta \\
\cdots & 0 & 0 & 0 & M & 1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
\cdots & 0 & 0 & 0 & 0 & 1-\tau m & 0 & 0 & 0 & 0 & 0 & \cdots & \tau \\
\cdots & 0 & 0 & 0 & 0 & -\alpha m & -1 & 0 & 0 & -m & 0 & \cdots & \alpha \\
\cdots & 0 & 0 & 0 & 0 & -\beta m & 1 & 0 & 0 & 0 & -1 & \cdots & \beta \\
\cdots & 0 & 0 & 0 & 0 & -1 & 1 & m & 0 & 0 & 0 & \cdots & 0 \\
\cdots & 0 & 0 & 0 & 0 & 1 & 1 & 0 & m & 0 & 0 & \cdots & 0 \\
\cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & m & M-m & 0 & \cdots & 0 \\
\cdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\cdots & 0 & 0 & 0 & 0 & -\tau m & 0 & 0 & 0 & 0 & 0 & \cdots & \tau \\
\cdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \cdots & 0 \\
\cdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\cdots & 0 & 0 & 0 & 0 & -\alpha m & 0 & 0 & 0 & 0 & 0 & \cdots & \alpha \\
\cdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots
\end{pmatrix}
$$

Figure 6: Intermediate configuration for: **if** $R_k \leq 0$ **then** $R_j \leftarrow R_j + R_i$

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

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

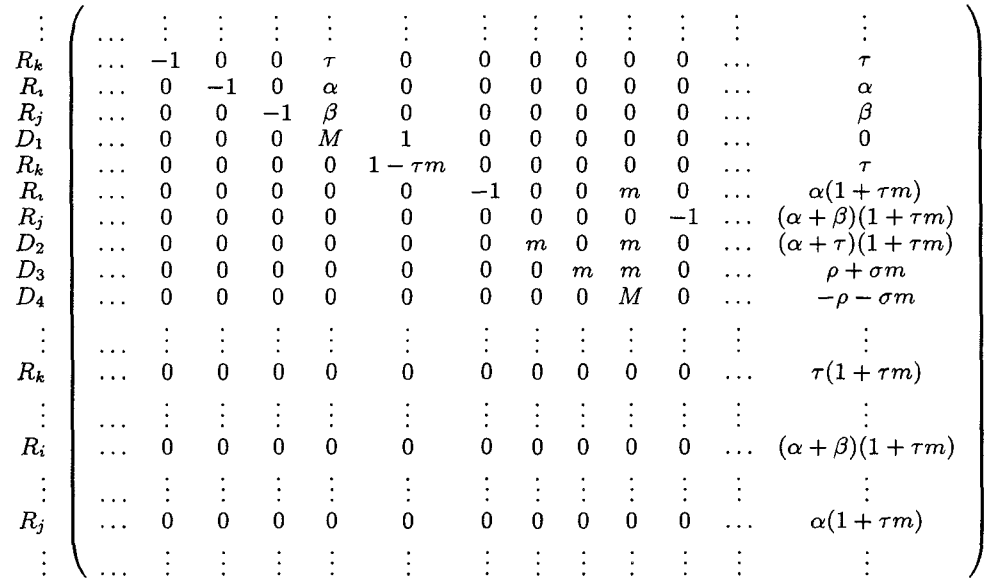

Figure 8: : Final configuration for: **if** $R_k \leq 0$ **then** $R_j \leftarrow R_j + R_i$

The work space required to generate A' is clearly $\Omega(\log s(n))$. However, it is not difficult to see that the macro statements of Lemma 2 that perform the binary search in the set of $s(n)$ RAM registers can be generated in space $O(\log s(n))$. The transformation that makes the program restricted is also logspace computable, and therefore the overall space demand is $O(\log s(n))$.

Since A' is restricted, the sequence of instructions executed is oblivious of the actual input. Assume that the output bit of A (indicating acceptance or rejection) is stored in register R_0 at the end of $A's$ execution. We modify the program by inserting, as the last instruction, the conditional statement "if $R_0 \leq 0$ then $R_1 \leftarrow R_1$". 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''.

From A'' we build the matrix $M(A'', I)$ according to the rules of Section 3. 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''.

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

Since each instruction can be simulated by a costant number of pivot operations, the order of $M(A'', I)$ is at most a constant times the running time of A''. By Theorem4 this is $\tilde{O}(t(n)s(n))$. Using the simulation of Section 3, we conclude that the program A'', and thus A, accepts if and only if to eliminate, e.g., column $N-2$ the GEPP algorithm uses row N. ∎

Theorem 6 *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 5 the question of acceptance by A is reducible, in space $\log(t(n))$ and thus also in parallel time $O(\log t(n))$, to Gaussian Elimination with Partial Pivoting on a matrix of size $O(t^2(n))$. 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. ∎

5 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 obtain polynomial

speedup over A if we are 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 [4]). 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. Condon. A Theory of Strict P-completeness. *Proc. STACS 92*, Lecture Notes in Computer Science, 577:33–44.

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

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

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

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

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

[7] 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.

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

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

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

[11] 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.