Parallel Solution of Block Tridiagonal Systems on Multi–core Machines

Gustavo Hime[1,2] hime@impa.br
Bruno Schulze[1] schulze@lncc.br
Dan Marchesin[2] marchesi@impa.br

1. Laboratório Nacional de Computação Científica
Av. Getúlio Vargas, 333, Quitandinha, 25651-075, Petrópolis, RJ
2. Instituto Nacional de Matemática Pura e Aplicada
Estrada Dona Castorina 110, 22460-320, Rio de Janeiro, RJ

June 16, 2008

Abstract

Block tridiagonal systems arise naturally when solving non–linear systems of equations that govern physical models for multiphase flow in one dimension. A wide range of phenomena is studied using such models, particularly in oil–related applications. Even though the models are one-dimensional, their simulation can consume computational resources arbitrarily, as grid refinement and reduced time steps provide qualitative insight to the model behaviour.

In order to exploit the new trend of parallel environments based on multi–core processors, we studied several algorithms for the parallel solution of block tridiagonal systems. In this work, we present a simple algorithm which we found to be the most effective to a particular problem size range, and evaluate its performance on a mid-range multi–core machine. We apply ScaLAPACK to solve the same test problems and compare the results.

1 Introduction

The discretization of models for balance systems often leads to linear systems

$$\mathbf{A}\mathbf{x} = \mathbf{y} \tag{1.1}$$

with a particular structure, where the coefficients $\mathbf{A}_{ij}$ are themselves $M \times M$ matrices — thus the term *block* — and are null if $|i - j| > 1$, i.e., the system is *block tridiagonal*, and the total number of variables and unknowns is NM, so $\mathbf{A} \in \mathbb{R}^{NM \times NM}$ and $\mathbf{x}$, $\mathbf{y} \in \mathbb{R}^{NM}$. Although such system can be solved by standard methods, ignoring both its block and tridiagonal peculiarities, the use of specialized solution algorithms can yield much greater accuracy and better performance.

Our motivation is the numerical simulation of physical models for phenomena that evolve in time. The time evolution algorithm solves a non–linear system of equations at each time step and builds the system to be solved in the next time step; this non–linear system is solved using Newton's method, which amounts to solving several linear systems consecutively. These two embedded iterative procedures are not parallelizable, since the data cascading dependency is inherent to both the time evolution and Newton's iterations.

The spatial discretization resolution — to which the size of the linear systems to be solved at the core of the algorithm is directly proportional — is usually small, so the time taken to solve one linear system may be negligible; but the total time is proportional to the number of time steps taken in the simulation, which is arbitrarily large. Highly efficient solvers for relatively small linear systems make longer simulation times feasible in a shorter period of real time. Parallel computing can only be exploited in the solution of these linear systems, which have limited potential for parallelization due to being small.

Parallel solvers for linear systems are available in both commercial and academic packages, one of the most notable being ScaLAPACK. However, performance analyses are usually performed using much larger problems than the ones we have in view. Also, the majority of the available software packages is based on the message passing model, i.e.

is implemented using MPI or a similar middleware, which was designed for distributed memory systems. We wish to investigate the adequacy of the existing software to the nowadays affordable shared memory machines based on multi–core processors.

Many parallelizable algorithms for the solution of special linear systems, i.e., banded, tridiagonal, block structured and such, have been published in the past. We tailored a particularly simple algorithms similar to the one by Mehrmann [6], which we present in section 2. In section 3, we describe the parallel environment used in the following experiments: in section 4, we perform numerical experiments analogous to the ones in [1], using the banded solver from ScaLAPACK; then we perform comparative experiments between the two solvers in section 5, and draw our conclusions from these results in 7.

2 Divide and Conquer Algorithm for Block Tridiagonal Systems

Consider the problem of solving (1.1) with $N = KP + (P-1)$. We can group the block coefficients of $\mathbf{A}$ into P blocks (of blocks) of size K as follows:

$$\begin{bmatrix} \mathbf{A}_1 & & & & & & \\ & \boldsymbol{\Theta}_1 & & & & & \\ \boldsymbol{\Gamma}_1 & \boldsymbol{\Lambda}_1 & \boldsymbol{\Phi}_1 & & & & \\ & \boldsymbol{\Omega}_1 & & & & & \\ & & & \mathbf{A}_2 & & & \\ & & & & \ddots & & \\ & & & & & \mathbf{A}_P & \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \boldsymbol{\xi}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_P \end{bmatrix} = \begin{bmatrix} \mathbf{y}_1 \\ \boldsymbol{\psi}_1 \\ \mathbf{y}_2 \\ \vdots \\ \mathbf{y}_P \end{bmatrix}, \tag{2.1}$$

with

$$\begin{array}{l} \mathbf{A}_i \in \mathbb{R}^{KM \times KM}, \\ \mathbf{x}_i,\ \mathbf{y}_i \in \mathbb{R}^{KM}, \\ \boldsymbol{\Lambda}_i,\ \boldsymbol{\Theta}_i,\ \boldsymbol{\Gamma}_i,\ \boldsymbol{\Phi}_i,\ \boldsymbol{\Omega}_i \in \mathbb{R}^{M \times M}, \text{ and} \\ \boldsymbol{\xi}_i,\ \boldsymbol{\psi}_i \in \mathbb{R}^{M}. \end{array}$$

The entries of $\mathbf{A}$, $\mathbf{x}$ and $\mathbf{y}$ in (1.1) map to the elements of (2.1) through

$$\begin{array}{rll} \boldsymbol{\Lambda}_i & \equiv \mathbf{A}_{i\times(K+1)} & \\ \boldsymbol{\Gamma}_i & \equiv \mathbf{B}_{i\times(K+1)} & \\ \boldsymbol{\Omega}_i & \equiv \mathbf{B}_{i\times(K+1)+1} & \\ \boldsymbol{\Theta}_i & \equiv \mathbf{C}_{i\times(K+1)-1} \ , & i = 1, \ldots, P-1; \\ \boldsymbol{\Phi}_i & \equiv \mathbf{C}_{i\times(K+1)} & \\ \boldsymbol{\xi}_i & \equiv \mathbf{x}_{i\times(K+1)} & \\ \boldsymbol{\psi}_i & \equiv \mathbf{y}_{i\times(K+1)} & \end{array} \tag{2.2}$$

the rest of the non-zero entries of $\mathbf{A}$ are contained in the submatrices $\mathbf{A}_i$, which are also block tridiagonal, and the rest of the vectors $\mathbf{x}$ and $\mathbf{y}$ map to the $\mathbf{x}_i$ and $\mathbf{y}_i$ accordingly.

The full system given in equation (2.1) is solved in three stages: first, each block tridiagonal system of size K is solved independently, and the solutions obtained are functions of the unknowns $\boldsymbol{\xi}_i$; second, we construct from these solutions a new, smaller block tridiagonal system of size $P-1$, and determine the $\boldsymbol{\xi}_i$ unknowns; finally, we compose the solutions of the two previous stages to find the solution of the full system.

We now formalize the first step: introducing the $KM \times M$ rectangular matrices

$$\mathbf{U} = \begin{bmatrix} \mathbf{I} \\ \mathbf{0} \\ \vdots \\ \mathbf{0} \end{bmatrix} \quad \text{and} \quad \mathbf{V} = \begin{bmatrix} \mathbf{0} \\ \vdots \\ \mathbf{0} \\ \mathbf{I} \end{bmatrix},$$

we write

$$\begin{array}{l} \mathbf{A}_1\mathbf{x}_1 + \mathbf{V}\boldsymbol{\Theta}_1\boldsymbol{\xi}_1 = \mathbf{y}_1 \\ \mathbf{U}\boldsymbol{\Omega}_{i-1}\boldsymbol{\xi}_{i-1} + \mathbf{A}_i\mathbf{x}_i + \mathbf{V}\boldsymbol{\Theta}_i\boldsymbol{\xi}_i = \mathbf{y}_i, \\ \qquad\qquad\qquad\qquad i = 2, \ldots, P-1, \\ \mathbf{U}\boldsymbol{\Omega}_{P-1}\boldsymbol{\xi}_{P-1} + \mathbf{A}_P\mathbf{x}_P = \mathbf{y}_P. \end{array} \tag{2.3}$$

Defining

$$\begin{aligned}\overline{\mathbf{y}}_i &= \mathbf{A}_i^{-1}\mathbf{y}_i,\\ \overline{\mathbf{U}}_i &= \mathbf{A}_i^{-1}\mathbf{U}\boldsymbol{\Omega}_{i-1} \qquad \text{and}\\ \overline{\mathbf{V}}_i &= \mathbf{A}_i^{-1}\mathbf{V}\boldsymbol{\Theta}_i,\end{aligned}$$

the solutions for $\mathbf{x}_i$ in (2.3) can be written as functions of $\boldsymbol{\xi}_i$, i.e.,

$$\begin{aligned}\mathbf{x}_1 &= \overline{\mathbf{y}}_1 - \overline{\mathbf{V}}_1\boldsymbol{\xi}_1,\\ \mathbf{x}_i &= \overline{\mathbf{y}}_i - \overline{\mathbf{U}}_i\boldsymbol{\xi}_{i-1} - \overline{\mathbf{V}}_i\boldsymbol{\xi}_i, \quad i = 2,\ldots,P-1,\\ \mathbf{x}_P &= \overline{\mathbf{y}}_P - \overline{\mathbf{U}}_P\boldsymbol{\xi}_{P-1}.\end{aligned} \tag{2.4}$$

The objects $\overline{\mathbf{y}}_i$, $\overline{\mathbf{U}}_i$ and $\overline{\mathbf{V}}_i$ can be computed using any method such as those described in the previous two sections.

The second step is to use these solutions to determine the unknowns $\boldsymbol{\xi}_i$. For each row of (2.1) corresponding to a $\boldsymbol{\psi}_i$, we use the $\mathbf{x}_i$ defined in (2.4) to obtain an equation of the form

$$\boldsymbol{\Gamma}_i(\mathbf{V}^{\mathrm{T}}\mathbf{x}_i) + \boldsymbol{\Lambda}_i\boldsymbol{\xi}_i + \boldsymbol{\Phi}_i(\mathbf{U}^{\mathrm{T}}\mathbf{x}_{i+1}) = \boldsymbol{\psi}_i, \quad i = 1,...,P-1. \tag{2.5}$$

Each solution $\mathbf{x}_i$ is a function of $\boldsymbol{\xi}_i$ and $\boldsymbol{\xi}_{i-1}$, except for $\mathbf{x}_1$ and $\mathbf{x}_P$. Applying the expressions for $\mathbf{x}_i$ from (2.4) to (2.5), we obtain

$$\begin{aligned}&\boldsymbol{\Gamma}_1\big[\mathbf{V}^{\mathrm{T}}(\overline{\mathbf{y}}_1 - \overline{\mathbf{V}}_1\boldsymbol{\xi}_1)\big] + \boldsymbol{\Lambda}_1\boldsymbol{\xi}_1 +\\ &\quad\boldsymbol{\Phi}_1\big[\mathbf{U}^{\mathrm{T}}(\overline{\mathbf{y}}_2 - \overline{\mathbf{U}}_2\boldsymbol{\xi}_1 - \overline{\mathbf{V}}_2\boldsymbol{\xi}_2)\big] = \boldsymbol{\psi}_1,\\ &\\ &\boldsymbol{\Gamma}_i\big[\mathbf{V}^{\mathrm{T}}(\overline{\mathbf{y}}_i - \overline{\mathbf{U}}_i\boldsymbol{\xi}_{i-1} - \overline{\mathbf{V}}_i\boldsymbol{\xi}_i)\big] + \boldsymbol{\Lambda}_i\boldsymbol{\xi}_i\\ &\quad+\boldsymbol{\Phi}_i\big[\mathbf{U}^{\mathrm{T}}(\overline{\mathbf{y}}_{i+1} - \overline{\mathbf{U}}_{i+1}\boldsymbol{\xi}_i - \overline{\mathbf{V}}_{i+1}\boldsymbol{\xi}_{i+1})\big] = \boldsymbol{\psi}_i,\\ &\qquad\qquad i = 2,\ldots,P-2,\\ &\\ &\boldsymbol{\Gamma}_{P-1}\big[\mathbf{V}^{\mathrm{T}}(\overline{\mathbf{y}}_{P-1} - \overline{\mathbf{U}}_{P-1}\boldsymbol{\xi}_{P-2} - \overline{\mathbf{V}}_{P-1}\boldsymbol{\xi}_{P-1})\big] +\\ &\quad\boldsymbol{\Lambda}_{P-1}\boldsymbol{\xi}_{P-1} + \boldsymbol{\Phi}_{P-1}\big[\mathbf{U}^{\mathrm{T}}(\overline{\mathbf{y}}_P - \overline{\mathbf{U}}_P\boldsymbol{\xi}_{P-1})\big] = \boldsymbol{\psi}_{P-1},\end{aligned} \tag{2.6}$$

from which we derive the coefficients of a new tridiagonal system of $P-1$ equations

$$\begin{bmatrix}\overline{\boldsymbol{\Lambda}}_1 & \overline{\boldsymbol{\Phi}}_1 & & & \\ \overline{\boldsymbol{\Gamma}}_2 & \overline{\boldsymbol{\Lambda}}_2 & \overline{\boldsymbol{\Phi}}_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \overline{\boldsymbol{\Gamma}}_{P-2} & \overline{\boldsymbol{\Lambda}}_{P-2} & \overline{\boldsymbol{\Phi}}_{P-2}\\ & & & \overline{\boldsymbol{\Gamma}}_{P-1} & \overline{\boldsymbol{\Lambda}}_{P-1}\end{bmatrix}\begin{bmatrix}\boldsymbol{\xi}_1\\ \boldsymbol{\xi}_2\\ \vdots\\ \boldsymbol{\xi}_{P-2}\\ \boldsymbol{\xi}_{P-1}\end{bmatrix} = \begin{bmatrix}\overline{\boldsymbol{\psi}}_1\\ \overline{\boldsymbol{\psi}}_2\\ \vdots\\ \overline{\boldsymbol{\psi}}_{P-2}\\ \overline{\boldsymbol{\psi}}_{P-1}\end{bmatrix} \tag{2.7}$$

with matrix coefficients given by

$$\begin{aligned}\overline{\boldsymbol{\Lambda}}_i &= \boldsymbol{\Lambda}_i - \boldsymbol{\Gamma}_i(\mathbf{V}^{\mathrm{T}}\overline{\mathbf{V}}_i) - \boldsymbol{\Phi}_i(\mathbf{U}^{\mathrm{T}}\overline{\mathbf{U}}_{i+1}),\\ &\qquad i = 1,\ldots,P-1,\\ \overline{\boldsymbol{\Gamma}}_i &= -\boldsymbol{\Gamma}_i(\mathbf{V}^{\mathrm{T}}\overline{\mathbf{U}}_i),\\ &\qquad i = 2,\ldots,P-1,\\ \overline{\boldsymbol{\Phi}}_i &= -\boldsymbol{\Phi}_i(\mathbf{U}^{\mathrm{T}}\overline{\mathbf{V}}_{i+1}),\\ &\qquad i = 1,\ldots,P-2,\\ \overline{\boldsymbol{\psi}}_i &= \boldsymbol{\psi}_i - \boldsymbol{\Gamma}_i(\mathbf{V}^{\mathrm{T}}\overline{\mathbf{y}}_i) - \boldsymbol{\Phi}_i(\mathbf{U}^{\mathrm{T}}\overline{\mathbf{y}}_{i+1}),\\ &\qquad i = 1,\ldots,P-1.\end{aligned} \tag{2.8}$$

Notice that the inner products involving $\mathbf{U}$ and $\mathbf{V}$ are trivial to obtain, since their computation reduces to isolating either the first or the last entry in a block vector, respectively. The third step is trivial as well: once the system (2.7) is solved, the solution of the full system is computed from the expressions in (2.4).

This algorithm brings its parallelization strategy in its very formulation. The constant P represents the number of processing elements, and the constant K is the size of the block tridiagonal system that each processing element must solve. The matrix $\mathbf{A}_i$ of each subsystem can have its factors computed on a separate processing unit using the serial algorithm presented in [5], and the intermediate results $\overline{\mathbf{U}}_i$, $\overline{\mathbf{V}}_i$ and $\overline{\mathbf{y}}_i$ can be computed locally as well. The values of

$\overline{\mathbf{\Lambda}}_i$, $\overline{\mathbf{\Gamma}}_i$ and $\overline{\mathbf{\Phi}}_i$ in equation (2.8) require all processes to reduce the data to a master process, which will subsequently solve the coupling system (2.7).

It is not necessary to aggregate all $\overline{\mathbf{U}}_i$ and $\overline{\mathbf{V}}_i$ objects: the computation of the coefficients in (2.8) requires only the $M \times M$ matrices $\mathbf{U}^{\intercal}\overline{\mathbf{U}}_i$, $\mathbf{U}^{\intercal}\overline{\mathbf{V}}_i$, $\mathbf{V}^{\intercal}\overline{\mathbf{U}}_i$ and $\mathbf{V}^{\intercal}\overline{\mathbf{V}}_i$, which for $K \gg M$ represent a much smaller data transfer. However, the whole objects $\overline{\mathbf{U}}_i$ and $\overline{\mathbf{V}}_i$ are required to obtain the solution chunks $\mathbf{y}_i$ after the system (2.7) has been solved. Therefore, it is cheaper to aggregate and broadcast only the coupling data, and each process will compute locally its portion of the solution, which will be spread across the parallel environment, as was the case in the previous algorithm.

The value of N does not have to be of the form $N = KP + (P - 1)$, of course: one processing element can hold a few less equations than others, without significant loss of CPU time. The formulation presented here assumes a homogeneous set of processing units, i.e., the problem of load balancing is not considered. Complexity analysis for this algorithm can be found in [5], along with a comparison with cyclic reduction (as presented in [3] and [2]). A general form allowing for blocks of different sizes M_i is given in [6].

3 Parallel environment used in the tests

As the nomenclature has become rather imprecise with the introduction of multi–core technology, we clarify what meaning we attach to each technical term in this article: a *processor* contains one or more *CPUs* or *cores*, cache memory and other circuitry, all embedded in one single piece of silicon. The processor's cache may be shared by its cores, and usually is. A *shared memory machine* consists of one or more processors and memory, which every CPU may access. This access may be indirect or direct, depending on whether the memory is divided over the processors or not. In the case where it is divided, a processor accesses its own memory directly, but in order to access memory belonging to another it must request the other processor to perform the memory access for it. This is referred to as *non-uniform memory access* — NUMA.

The operating system schedules processes in a given shared memory, multiprocessor/multi–core machine to cores, but a process can migrate from one core to another along its total runtime. A process that does not migrate promotes better cache usage, and a process that only accesses memory local to the processor that hosts it promotes overall greater efficiency.

We performed experiments using message–passing and multithreaded programs on an AMD Opteron–based shared memory machine, with eight dual–core processors, therefore we report results up to sixteen CPUs. The machine runs a custom compiled Linux 2.6 kernel. The operating system provides mechanisms for binding processes/threads to specific CPUs, which we used to ensure optimal usage of cache and NUMA locality, as well as repeatable results. These mechanisms and their interfaces are particular to each operating system that provides them, i.e., they are not standardized and may not be available for a particular parallel environment. The experiments presented herein could not have been performed on a parallel environment without such facilities.

We used GCC 4.1 to compile ScaLAPACK and MPICH2 from source, selecting the mixed shared–memory/network channel for the latter. All optimizations were enabled, without deviating from the defaults, i.e. we used these packages "out–of–the–box" as much as we could. This version of GCC supports OpenMP, which we used to implement the multithreaded version of our solver.

4 ScaLAPACK Reference Benchmark

We present numerical results of experiments analogous to those in [1]. The n-by-n matrix in the system $\mathbf{Ax} = \mathbf{b}$ is not block tridiagonal, but banded with equal lower and upper bandwidths $k_l = k_u \equiv k$; the matrix $\mathbf{A}$ has ones within the band and some value $\alpha > 1$ in the main diagonal. The right hand side $\mathbf{b}$ is $(1, ..., n)^T$. We modified the test program `xddblu` from the ScaLAPACK distribution to perform these test problems.

Results were presented in [1] for problem sizes $(n, k) = (20000, 10)$, $(n, k) = (100000, 10)$ and $(n, k) = (100000, 50)$, with various values for α in the range $[1.01, 100]$, Table 4 is formatted similarly to the ones in reference [1]. The ten-year gap in technology is clearly visible when comparing the execution times of the two benchmarks.

	$\alpha = 10$			$\alpha = 5$			$\alpha = 1.01$		
P	t	S	ϵ	t	S	ϵ	t	S	ϵ
Size(n,k) = (20000,10)									
1	37	1.00	8.14e-5	37	1.00	5.23e-4	37	1.00	4.23e-3
2	83	0.44	8.17e-5	58	0.63	1.45e-3	58	0.63	6.09e-3
4	42	0.86	8.85e-5	30	1.21	1.38e-3	30	1.21	8.10e-3
8	19	1.95	7.74e-5	19	1.95	1.87e-3	19	1.95	1.30e-2
16	14	2.61	8.00e-5	14	2.63	3.00e-3	14	2.61	2.38e-4
Size(n,k) = (100000,10)									
1	184	1.00	1.73e-5	184	1.00	7.58e-4	185	1.00	5.48e-1
2	280	0.66	1.84e-5	292	0.63	3.82e-4	293	0.63	2.73e-3
4	158	1.16	1.62e-5	149	1.24	5.25e-4	149	1.25	3.72e-3
8	102	1.81	1.65e-5	81	2.29	3.18e-4	81	2.30	6.34e-3
16	77	2.39	1.78e-5	50	3.73	3.47e-4	50	3.76	1.53e-3
Size(n,k) = (100000,50)									
1	2099	1.00	1.85e-3	2098	1.00	4.76e-3	2100	1.00	1.07e-3
2	4580	0.46	4.63e-3	4579	0.46	4.94e-3	4580	0.46	1.49e-3
4	2293	0.92	9.18e-3	2290	0.92	7.63e-3	2291	0.92	9.33e-4
8	1162	1.81	6.40e-3	1162	1.81	7.51e-3	1163	1.81	5.77e-4
16	690	3.04	9.41e-3	691	3.04	1.17e-2	692	3.03	1.65e-4

Table 4.1: Execution time (in milliseconds) for ScaLAPACK on Opteron–based system, for tests similar to those found in [1].

5 Comparison between ScaLAPACK and our solver

The problem sizes $(n, k) = (20000, 10)$, $(n, k) = (100000, 10)$ and $(n, k) = (100000, 50)$ are named "small", "medium" and "large" in [1], but nowadays they can all be considered small. However, we are interested in small problems.

The size of a block tridiagonal system is defined by a number N of *block equations* of M unknowns each, totaling NM scalar equations. The narrowest bandwidths sufficient to cover the three block diagonals would be $k_l = k_u = 2M - 1$. Therefore, when applied to a block tridiagonal system of size (N, M), a general banded solver such as the one provided in ScaLAPACK would deal with a problem of size $(n, k) = (NM, 2M - 1)$.

Our choices of M and N are justified as follows: in the target application, i.e. numerical simulation of physical systems, the size M of the blocks is defined by the physical model and is usually in the range $[2, 12]$. We chose $M \in \{2, 6, 12, 26\}$ for the test problems, corresponding to $k \in \{3, 11, 23, 51\}$. The first three values cover the range of interest, and the fourth pushes the algorithm's space requirement by a factor of about 6.5 (the ratio $26^2/12^2$). We expect different cache behaviour to occur, if at all, for this last value of M.

In the target application, the number of block equations N is defined by the spatial grid resolution, which can be increased arbitrarily in convergence and stability analyses, within the available computational limits. For the selected values of k, we found that $n = 2^{20}$ was the largest power of two for which the test program provided with ScaLAPACK would run without major intervention in its memory allocation. Thus, we set N in the experiments with our solver to the smallest value such that $NM \geq 2^{20}$.

In Table 5 we present estimates for the condition numbers of the matrices of each test problem. These numbers were computed using the MATLAB implementation of the algorithm found in [4], and agree with the values reported in [1]. However, the estimates for the new problem sizes are much less reliable, as can be seen by comparing the last three rows (the values should increase both row- and column-wise).

The average execution times of the two solvers, for the selected problem sizes, are shown in Tables 5.2 through 5.5. Times are reported in milliseconds; speedups for each program are relative to the time taken by the same program on a single core; and the error norm is the same computed in the ScaLAPACK test program. The relative machine precision was taken to be 1.1×10^{-16}. The variation of the execution times for consecutive runs was negligible.

In these test problems, the lightweight algorithm we propose outperforms ScaLAPACK's solver on serial execution by a factor in the range $[4.03, 6.29]$. Figure 5.1 gives the visual comparison for two particular problem sizes: qualitatively, the plots for all problem sizes are quite similar. If any different behaviour can be can be observed, it is

(n,k)	$\alpha = 10$	$\alpha = 5$	$\alpha = 1.01$
$(20000, 10)$	9.1	$4.2e+4$	$2.9e+6$
$(100000, 10)$	9.1	$4.3e+5$	$3.8e+6$
$(100000, 50)$	$1.7e+5$	$6.0e+6$	$4.7e+8$
$(1048576, 3)$	2.7	6.1	$8.2e+6$
$(1048576, 13)$	$1.5e+1$	$5.4e+6$	$5.8e+7$
$(1048576, 23)$	$9.2e+5$	$1.0e+7$	$1.4e+8$
$(1048576, 51)$	$2.1e+6$	$4.9e+6$	$1.2e+8$

Table 5.1: Estimated condition numbers.

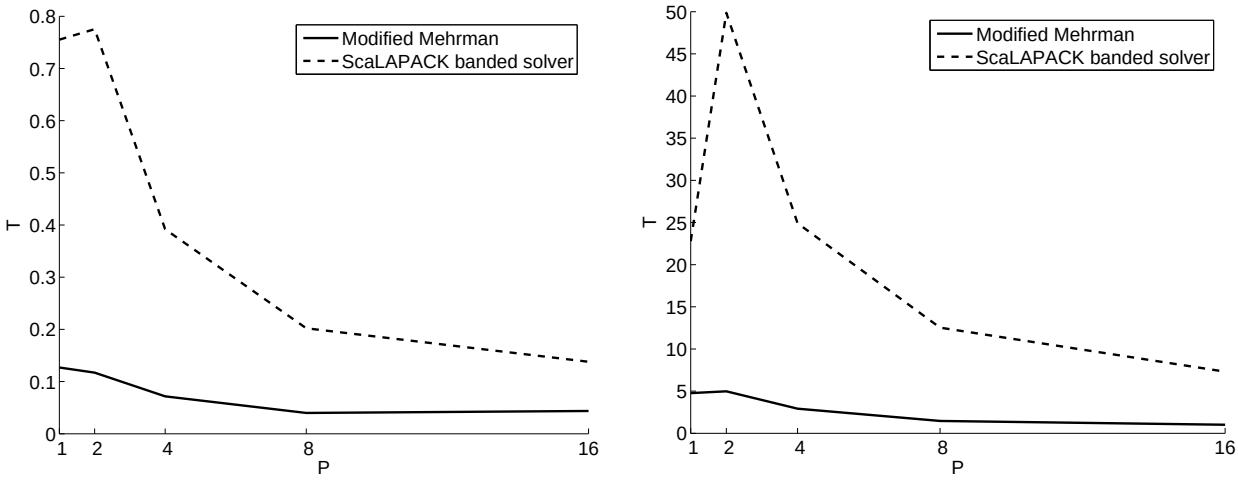
$(n,k) = (1048576, 3) \leftrightarrow (N,M) = (524289, 2)$									
	$\alpha = 10$			$\alpha = 5$			$\alpha = 1.01$		
P	t	S	ϵ	t	S	ϵ	t	S	ϵ
ScaLAPACK Implementation									
1	754	1.00	7.54e-7	756	1.00	1.09e-6	756	1.00	2.76e-4
2	724	1.04	7.46e-7	724	1.04	1.04e-6	776	0.97	8.01e-4
4	365	2.07	7.65e-7	364	2.08	1.07e-6	392	1.93	3.09e-4
8	192	3.94	7.46e-7	192	3.95	1.05e-6	203	3.73	2.45e-4
16	128	5.89	7.62e-7	129	5.87	1.02e-6	138	5.48	2.34e-4
Modified Mehrmann									
1	121	1.00(6.27)	8.90e-7	121	1.00(6.29)	1.01e-6	127	1.00(5.95)	3.44e-3
2	88	1.37(8.58)	8.90e-7	88	1.37(8.61)	1.01e-6	118	1.08(6.45)	4.87e-3
4	54	2.23(13.96)	8.90e-7	55	2.22(13.98)	1.01e-6	72	1.77(10.52)	2.23e-2
8	31	3.95(24.76)	8.90e-7	31	3.93(24.72)	1.01e-6	40	3.18(18.91)	1.80e-4
16	33	3.71(23.23)	8.90e-7	33	3.67(23.11)	1.01e-6	44	2.91(17.30)	1.78e-4

Table 5.2: Execution times for both programs, for the problem size $(n,k) = (2^{20}, 3)$. Times are given in milliseconds. The speedups are relative to the time on one CPU; the speedup in between parentheses for the Modified Mehrmann solver is relative to the time taken by ScaLAPACK on one CPU.

$(n,k) = (1048576, 13) \leftrightarrow (N,M) = (149797, 7)$									
	$\alpha = 10$			$\alpha = 5$			$\alpha = 1.01$		
P	t	S	ϵ	t	S	ϵ	t	S	ϵ
ScaLAPACK Implementation									
1	2833	1.00	1.63e-6	2836	1.00	3.17e-5	2839	1.00	1.10e-3
2	3882	0.73	1.65e-6	5019	0.56	2.29e-4	5020	0.57	1.02e-3
4	2034	1.39	1.68e-6	2538	1.12	1.51e-4	2538	1.12	1.16e-3
8	1108	2.56	1.61e-6	1303	2.18	5.08e-5	1304	2.18	2.11e-3
16	722	3.93	1.63e-6	845	3.36	1.12e-4	845	3.36	3.50e-3
Modified Mehrmann									
1	611	1.00(4.64)	1.26e-6	611	1.00(4.64)	1.41e-6	691	1.00(4.11)	2.19e-2
2	425	1.44(6.67)	1.26e-6	432	1.41(6.57)	1.41e-6	674	1.03(4.22)	1.66e-1
4	253	2.42(11.24)	1.26e-6	262	2.33(10.83)	1.41e-6	406	1.70(7.01)	1.06e-2
8	136	4.50(20.88)	1.26e-6	145	4.22(19.57)	1.41e-6	211	3.28(13.48)	3.23e-2
16	134	4.58(21.25)	1.26e-6	142	4.32(20.07)	1.41e-6	176	3.94(16.19)	3.04e-3

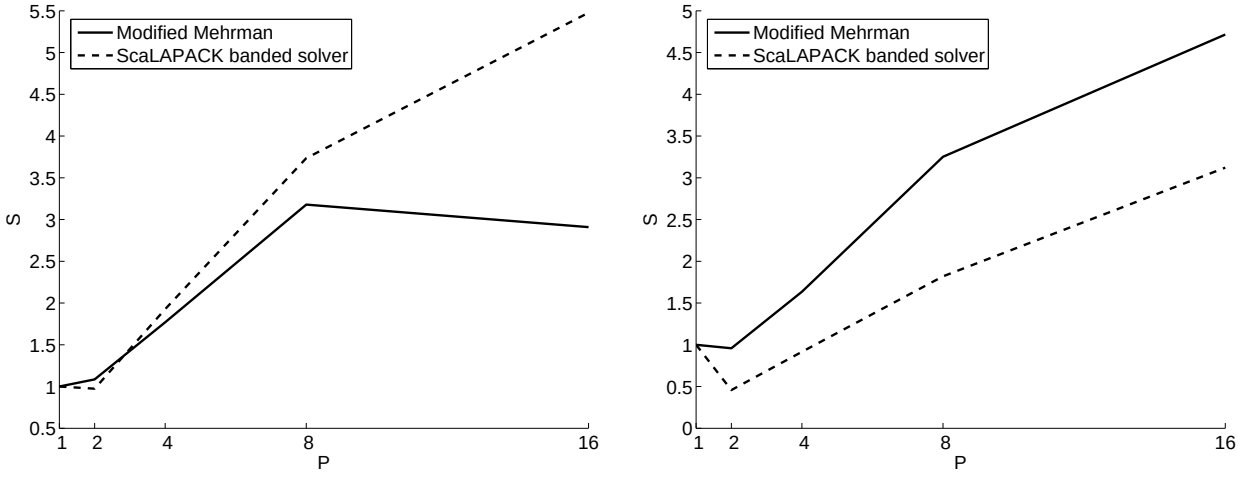
Table 5.3: Analogous to Table 5.2, for problem size $(n,k) = (2^{20}, 13)$.

that the narrower the bandwidth, the smaller is the relative increase of ScaLAPACK from one to two cores: the two plots in Figure 5.1 are extreme opposites.



(a) (b)

Figure 5.1: Wall time (in seconds) of the two solvers for two problem sizes: on the left, for $N = 2^{20}$, $K = 3$, and $\alpha = 1.01$, and on the right for the same N and α but with $K = 51$.



(a) (b)

Figure 5.2: Relative speedup the two solvers for two problem sizes: on the left, for $N = 2^{20}$, $K = 3$, and $\alpha = 1.01$, and on the right for the same N and α but with $K = 51$.

$(n,k) = (1048576, 23) \leftrightarrow (N,M) = (87382, 12)$									
	$\alpha = 10$			$\alpha = 5$			$\alpha = 1.01$		
P	t	S	ϵ	t	S	ϵ	t	S	ϵ
ScaLAPACK Implementation									
1	6300	1.00	3.24e-4	6296	1.00	8.75e-3	6305	1.00	6.85e-4
2	12851	0.49	7.30e-4	12848	0.49	8.11e-4	12848	0.49	4.89e-4
4	6415	0.98	7.79e-4	6415	0.98	1.96e-3	6414	0.98	6.06e-4
8	3241	1.94	3.24e-4	3241	1.94	8.19e-4	3240	1.95	4.32e-4
16	1879	3.35	4.72e-4	1881	3.35	1.50e-3	1881	3.35	4.63e-4
Modified Mehrmann									
1	1398	1.00(4.51)	1.45e-6	1426	1.00(4.42)	6.57e-5	1565	1.00(4.03)	1.35e-2
2	917	1.52(6.87)	1.45e-6	1524	0.94(4.13)	1.47e-4	1602	0.98(3.94)	2.13e-2
4	543	2.58(11.61)	1.45e-6	905	1.58(6.96)	2.61e-4	944	1.66(6.68)	3.34e-2
8	301	4.65(20.96)	1.45e-6	462	3.09(13.65)	3.22e-5	481	3.26(13.12)	6.14e-4
16	293	4.77(21.52)	1.45e-6	364	3.92(17.31)	2.32e-5	371	4.23(17.03)	3.40e-4

Table 5.4: Analogous to Table 5.2, for problem size $(n,k) = (2^{20}, 23)$.

$(n,k) = (1048576, 51) \leftrightarrow (N,M) = (40330, 26)$									
	$\alpha = 10$			$\alpha = 5$			$\alpha = 1.01$		
P	t	S	ϵ	t	S	ϵ	t	S	ϵ
ScaLAPACK Implementation									
1	22786	1.00	6.66e-4	22799	1.00	1.38e-3	22790	1.00	9.45e-4
2	49845	0.46	1.13e-3	49867	0.46	2.40e-3	49865	0.46	1.11e-3
4	24906	0.91	1.20e-3	24906	0.92	9.77e-4	24906	0.92	2.47e-3
8	12530	1.82	4.15e-4	12529	1.82	1.07e-3	12530	1.82	2.90e-3
16	7302	3.12	1.17e-3	7302	3.12	8.09e-4	7303	3.12	8.20e-4
Modified Mehrmann									
1	4260	1.00(5.35)	4.32e-5	4358	1.00(5.23)	5.90e-4	4767	1.00(4.78)	1.15e-3
2	4698	0.91(4.85)	6.29e-5	4752	0.92(4.80)	8.75e-5	4976	0.96(4.58)	4.00e-2
4	2777	1.53(8.21)	3.12e-5	2805	1.55(8.13)	9.26e-5	2915	1.64(7.82)	8.50e-2
8	1397	3.05(16.31)	4.62e-5	1411	3.09(16.17)	9.21e-5	1466	3.25(15.55)	1.23e-1
16	999	4.27(22.83)	4.32e-5	1002	4.35(22.76)	2.27e-4	1011	4.72(22.55)	2.03e-1

Table 5.5: Analogous to Table 5.2, for problem size $(n,k) = (2^{20}, 51)$.

Relative speedups of each program are shown in Figure 5.2. The speedup of each program is relative to its own serial execution time: the net gain of the modified Mehrmann solver over ScaLAPACK's is given between parentheses in Tables 5.2 through 5.5. These graphs illustrate the scalability of both solvers: ScaLAPACK scales linearly up to sixteen cores, whereas our solver stagnates at eight, particularly for narrow bandwidths. Figure 5.2a is the worst case, where the execution time actually increases when the number of cores is increased from eight to sixteen. Figure 5.2b shows the behaviour for systems with larger bandwidth, where the slope of the speedup curve changes at eight cores, but there is still a gain as P increases.

However, even assuming that ScaLAPACK would scale linearly for $P > 16$, it would still require over seventy CPUs to be faster than our solver, because of the factor of at least four between the serial execution times — i.e. the absolute speedup of 18.9 for $P = 8$, given by the lower right corner of Table 5.2, would be achieved by ScaLAPACK only at $P > 77$.

6 Programming model impact

The tests presented in the previous section were conducted using MPI on a shared memory machine with eight dual-processors. Although applicable to this architecture, the message passing model was developed and is required only for distributed memory environments. ScaLAPACK is based on the message passing paradigm, which translates into a legacy overhead. We refactored our solver using thread based fork-join parallelism, as provided for by the OpenMP standard, and compared the performance of the two implementations of the same algorithm. The code base for both solvers is the same, using conditional compilation, so all difference in performance can be attributed to the parallelization middleware.

$(n,k) = (1048578, 3) \leftrightarrow (N,M) = (524289, 2)$						
	$\alpha = 10$		$\alpha = 5$		$\alpha = 1.01$	
P	t	S	t	S	t	S
MPI Implementation						
1	121	1.00	121	1.00	127	1.00
2	88	1.37	88	1.37	118	1.08
4	54	2.23	55	2.22	72	1.77
8	31	3.95	31	3.93	40	3.18
16	33	3.71	33	3.67	44	2.91
OpenMP Implementation						
2	89	1.36	89	1.35	100	1.28
4	47	2.61	46	2.62	50	2.55
8	31	3.92	31	3.93	34	3.84
16	15	8.17	15	8.38	16	7.98

$(n,k) = (1048580, 51) \leftrightarrow (N,M) = (40330, 26)$						
	$\alpha = 10$		$\alpha = 5$		$\alpha = 1.01$	
P	t	S	t	S	t	S
MPI Implementation						
1	4260	1.00	4358	1.00	4767	1.00
2	4698	0.91	4752	0.92	4976	0.96
4	2777	1.53	2805	1.55	2915	1.64
8	1397	3.05	1411	3.09	1466	3.25
16	999	4.27	1002	4.35	1011	4.72
OpenMP Implementation						
2	3064	1.39	3115	1.40	3316	1.44
4	1526	2.79	1551	2.81	1652	2.89
8	822	5.18	834	5.22	884	5.39
16	428	9.97	434	10.06	458	10.41

Table 6.1: Wall times (in milliseconds) and speedups for the MPI and OpenMP implementations of the modified Mehrmann algorithm. The information for one CPU is identical for both programs, and presented only once. The information for the MPI implementation is identical to that contained in Tables 5.2 and 5.5, but we repeat here for ease of comparison: the solver errors are identical for both programs, since they perform exactly the same operations, and are not repeated for brevity (see Tables 5.2 and 5.5).

For the narrowest bandwidth, performance is similar up to eight cores; for the largest, the OpenMP solves the same problems in about 30% less time. However, the saturation observed from eight to sixteen cores disappears completely in the OpenMP solver, as illustrated by the nice straight lines shown in Figure 6.2.

7 Conclusions

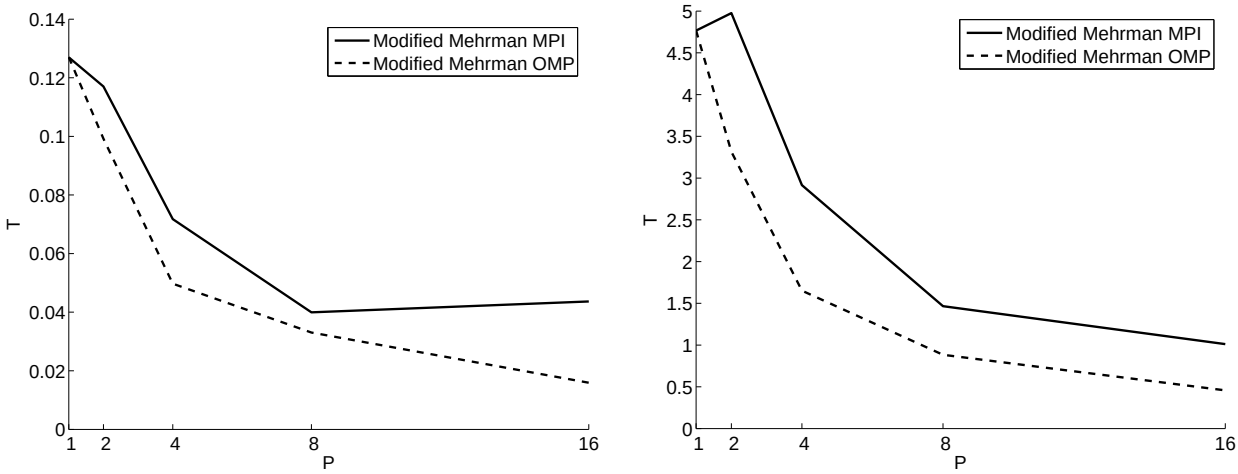
In this work we present an investigation of the performance of parallel solvers for block tridiagonal system. We focus on machines and problem sizes that are considered small in the context of the currently available technology.

The divide and conquer algorithm proposed in this work is fairly easy to implement and outperforms ScaLAPACK in the problem size range considered. It has been implemented using message passing for comparison with ScaLAPACK, and with OpenMP for advantageous use of shared memory hardware. On our test machine, using all sixteen cores, the factor by which our OpenMP based solver was faster than ScaLAPACK was in the range $[8.6, 17.1]$.

We found the precision of our implementation adequate for the numerical problems in flow simulation that motivated this work. The results presented herein justified our development of an alternative to the MPI based solvers available in the scientific community. We have contributed a MATLAB package to the Mathworks repository, including the OpenMP source code, which can be downloaded and used for free.

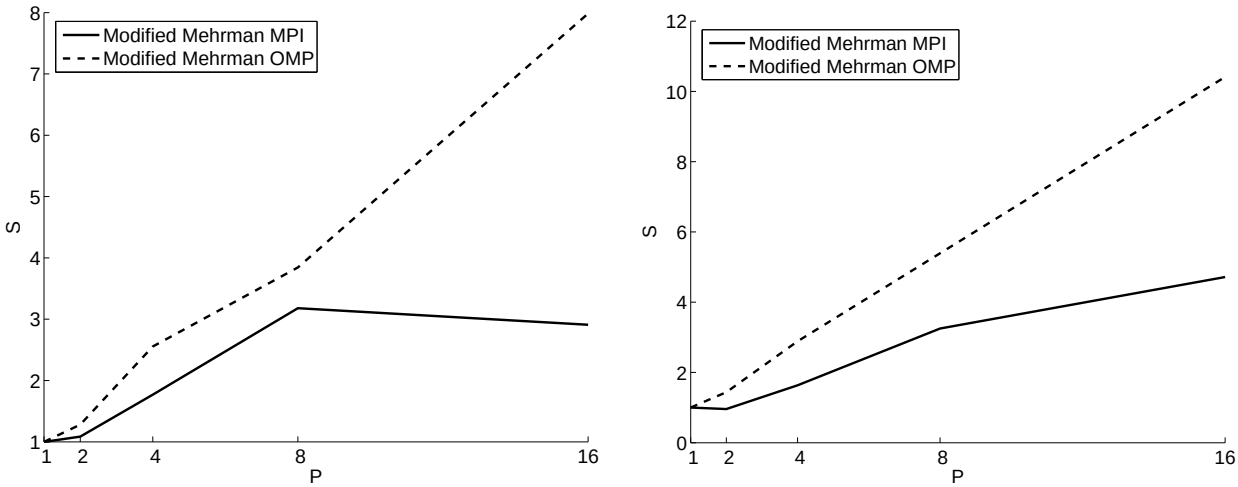
References

[1] P. Arbenz, A. Cleary, J. Dongarra, and M. Hegland. A comparison of parallel solvers for diagonally dominant and general narrow-banded linear systems II. In *European Conference on Parallel Processing*, pages 1078–1087,



(a) (b)

Figure 6.1: Wall times (in seconds) of the MPI and OpenMP versions of our solver for the same problem sizes as in the previous figures.



(a) (b)

Figure 6.2: Relative speedup of the MPI and OpenMP versions of our solver for the same problem sizes as in the previous figures.

1999.

[2] B. L. Buzbee, G. H. Golub, and C. W. Nielson. On direct methods for solving Poisson's equations. *SIAM J. Numer. Anal.*, 7(4):627–656, 1970.

[3] I. N. Hajj and S. Skelboe. A multilevel parallel solver for block tridiagonal and banded linear systems. *Parallel Computing*, 15:21–45, 1990.

[4] N. J. Higham and F. Tisseur. A block algorithm for matrix 1-norm estimation, with an application to 1-norm pseudospectra. In *SIAM Journal Matrix Anal. Appl.*, volume 21, pages 1185–1201, 2000.

[5] G. Hime. Parallel solution of nonlinear balance systems. Master's thesis, LNCC, 2007.

[6] V. Mehrmann. Divide and conquer methods for block tridiagonal systems. *Parallel Computing*, 19(3):257–279, 1993.