Parallel implementation of Lanczos bidiagonalization on Convex Exemplar

Rasmus Munk Larsen
Department of Computer Science
University of Århus
Ny Munkegade, building 540
DK-8000 Århus C
e-mail: rmunk@daimi.aau.dk

Per Christian Hansen
Department of Mathematical Modelling
Section for Numerical Analysis
Building 321, Technical University of Denmark,
DK-2800 Lyngby, Denmark.
Email: pch@imm.dtu.dk.
URL: www.imm.dtu.dk/~pch

Oktober 1998

Abstract

This report summarizes our work on parallel iterative algorithms for large-scale sparse or structured ill-posed problems. We describe a regularization method based on approximate singular value decomposition calculated using the Lanczos bidiagonalization. The method is implemented on a shared memory multiprocessor, the Convex Exemplar, and a models of two diferent implementations are derived to analyze mainly their cache performance. The performance models are used to suggest a reordering of the operations in the Lanczos bidiagonalization, which makes it possible to combine the two matrix-vector multiplications with $\boldsymbol{A}$ and $\boldsymbol{A}^T$ that take place in each iteration. The operations in the matrix-vector multiplications are interleaved in a block-wise manner and implemented in a single parallel subroutine, which reduces the number of cache misses by a factor of two. The performance of the parallel implementations is measured for a simple test problem and compared with the model predictions.

1 Introduction

As the scientific and engineering problems get larger, direct methods that compute a decomposition of a matrix become too costly on today's computer systems. Hence, the need for iterative methods for linear systems of equations increases.

Large-scale problems very often lead to sparse or structured (e.g. Toeplitz) matrices, and iterative methods can usually exploit the sparsity or structure better that the direct methods. Iterative methods are also competitive to direct methods when the matrix is dense and matrix-vector products can be computed efficiently such that the matrix itself is never actually stored.

In our present work, we investigate parallel implementations of the Lanczos bidiagonalization method for treating least squares problems with large condition numbers. This iterative algorithm has several important applications in the treatment of inverse problems. For example, it can be used to compute approximations to the largest singular values and vectors of the matrix $\boldsymbol{A}$, and these quantities are important analysis tools. Moreover, regularized solutions to $\min \|\boldsymbol{A}\, x - b\|_2$ can be constructed from the approximate singular values and vectors.

Our implementations were done in the fall of 1994 on a Convex Exemplar with 8 processors located at UNI•C[1]. The Convex Exemplar architecture is still in use, and other machines based on the NUMA architecture have appeared since then. One such example is the Silicon Graphics Origin 2000.

2 Lanczos Bidiagonalization and LSQR

The Lanczos bidiagonalization algorithm with starting vector $u_1 = b/\|b\|_2$ produces two sets of orthogonal Lanczos vectors $\{u_i\}$ and $\{v_i\}$ such that, after k steps and with

$$\boldsymbol{U}_k = (u_1, u_2, \ldots, u_{k+1}), \qquad \boldsymbol{V}_k = (v_1, v_2, \ldots, v_k),$$

we obtain a $(k+1) \times k$ bidiagonal matrix $\boldsymbol{B}_k$ satisfying

$$\boldsymbol{A}\, \boldsymbol{V}_k = \boldsymbol{U}_k\, \boldsymbol{B}_k. \tag{1}$$

Now let the singular value decomposition (SVD) of B_k be denoted by

$$\boldsymbol{B}_k = \boldsymbol{P}_k\, \boldsymbol{\Sigma}_k\, \boldsymbol{Q}_k^T; \tag{2}$$

see [13] for more details about the SVD.

The large singular values of $\boldsymbol{B}_k$, given by $\boldsymbol{\Sigma}_k$, are approximations to the large singular values of $\boldsymbol{A}$. Moreover, approximations to the corresponding left and right singular vectors of $\boldsymbol{A}$ are given by the associated columns of the matrices $\boldsymbol{U}_k \boldsymbol{P}_k$ and $\boldsymbol{V}_k \boldsymbol{Q}_k$. As k increases, the accuracy of some of the approximations increase, while other singular values in $\boldsymbol{\Sigma}_k$ start approximating other singular values of $\boldsymbol{A}$. In this way, we can use Lanczos bidiagonalization to compute (in principle) as many singular values and vectors of $\boldsymbol{A}$ as desired. In practice, we are always satisfied with computing a small fraction of them. This can be used to implement standard regularization methods such as Tikhonov regularization or truncated SVD, see [14].

In the presence of round-off, the convergence properties of the Lanczos bidiagonalization algorithm are more difficult, mainly because the Lanczos vectors loose orthogonality. As

[1] The Danish Computing Center for Research and Education.

a result, the matrix $\boldsymbol{\Sigma}_k$ will contain multiple singular values that approximate an isolated singular value of $\boldsymbol{A}$. However, there are well-known techniques to deal with these results of finite-precision arithmetic, and we use the techniques described by Cullum & Willoughby [7, 8].

The Lanczos bidiagonalization algorithm can also be used to compute approximate least squares solutions by projecting $\boldsymbol{A}$ onto the two subspaces spanned by the columns of $\boldsymbol{U}_k$ and $\boldsymbol{V}_k$. As a results, after k iterations, we write the solution vector as $x^{(k)} = \boldsymbol{V}_k y^{(k)}$, where $y^{(k)}$ is determined from the least squares problem $\min \|\boldsymbol{B}_k y^{(k)} - \boldsymbol{U}_k^T b\|_2$. The iteration vector is therefore $x^{(k)} = \boldsymbol{V}_k \boldsymbol{B}_k^{\dagger} \boldsymbol{U}_k^T b$, where $\boldsymbol{B}_k^{\dagger}$ is the pseudo inverse of $\boldsymbol{B}_k$. This is, in fact, the LSQR algorithm by Paige & Saunders [16, 17], and it is mathematically equivalent to applying the CG algorithm to the system $\boldsymbol{A}^T\boldsymbol{A}x = \boldsymbol{A}^T b$. The regularizing effects of this approach is studies in [15]. Other regularized solutions can be obtained by replacing the least squares problem with other problems, e.g., total least squares [11].

In this paper we describe an implementation of the Lanczos SVD algorithm on the Convex Exemplar, [6, 4]. We focus on the Lanczos bidiagonalization algorithm for computing the largest singular values and corresponding vectors of $\boldsymbol{A}$. We note that the bidiagonalization procedure completely dominates the computing time. The computation of the SVD of $\boldsymbol{B}_k$ including the treatment of multiple singular values (or the computation of $x^{(k)} = \boldsymbol{V}_k \boldsymbol{B}_k^{\dagger} \boldsymbol{U}_k^T b$ via proper updating schemes), requires only a small percentage of the total computing time. Notice that since the underlying problem is a least squares problem, there is no way to avoid the use of both $\boldsymbol{A}$ and $\boldsymbol{A}^T$ in this algorithm.

2.1 The Lanczos Algorithm

The algorithm we use is based on the symmetric Lanczos algorithm for computing the eigenvalues and vectors of a symmetric matrix (see [13, Chapter 9]). It is mathematically equivalent to the Lanczos bidiagonalization algorithm of Golub and Kahan [12], but it has the advantage that the tests for spurious singular values proposed by Cullum & Willoughby [7, Chapter 5], and used in their Fortran package, can be applied directly to the SVD computation.

The symmetric Lanczos algorithm applied to a symmetric matrix $\boldsymbol{M} \in \mathbb{R}^{n\times n}$ takes the following form:

Algorithm 1. The symmetric Lanczos algorithm

1. Choose a starting vector $r_0 \in \mathbb{R}^n$, and let
   $\beta_1 = \|r_0\|_2, \ \ \|q_1\|_2 = r_0/\beta_1$ and $q_0 \equiv 0$
2. **for** $j = 1, 2, \ldots, k$ **do**
   $p_j = \boldsymbol{M}\, q_j - \beta_j q_{j-1}$
   $\alpha_j = p_j^T q_j$
   $r_j = p_j - \alpha_j q_j$
   $\beta_{j+1} = \|r_j\|_2$
   $q_{j+1} = r_j/\beta_{j+1}$
   end

The basic iteration of our algorithm for computing the SVD of a rectangular matrix $\boldsymbol{A} \in \mathbb{R}^{m\times n}$, $m \geq n$ is derived by applying Algorithm 1 to the symmetric matrix

$$\boldsymbol{C} = \begin{pmatrix} 0 & \boldsymbol{A} \\ \boldsymbol{A}^T & 0 \end{pmatrix} \in \mathbb{R}^{(n+m)\times(n+m)} \ . \tag{3}$$

There is a close connection between the SVD of $\boldsymbol{A}$ and the eigenvalues and eigenvectors of $\boldsymbol{C}$. Suppose that

$$\boldsymbol{A} = \bar{\boldsymbol{U}}\ \bar{\boldsymbol{\Sigma}}\ \bar{\boldsymbol{V}}^T \tag{4}$$

and define the partitionings of $\bar{\boldsymbol{U}}$ and $\bar{\boldsymbol{\Sigma}}$

$$\bar{\boldsymbol{U}} = \begin{array}{cc} [\ \bar{\boldsymbol{U}}_1 & \bar{\boldsymbol{U}}_2\ ] \\ n & m-n \end{array}\ , \qquad \bar{\boldsymbol{\Sigma}} = \begin{array}{c} n \\ m-n \end{array} \begin{bmatrix} \bar{\boldsymbol{\Sigma}}_1 \\ 0 \end{bmatrix}\ ,$$

where $\bar{\boldsymbol{\Sigma}}_1 = \mathrm{diag}(\sigma_1, \sigma_2, \ldots, \sigma_n)$. From (4), (3) and the orthonormality of $\bar{\boldsymbol{U}}$ and $\bar{\boldsymbol{V}}$ if follows that

$$\boldsymbol{Z} = \frac{1}{\sqrt{2}} \begin{pmatrix} \bar{\boldsymbol{U}}_1 & \bar{\boldsymbol{U}}_1 & \sqrt{2}\bar{\boldsymbol{U}}_2 \\ \bar{\boldsymbol{V}} & -\bar{\boldsymbol{V}} & 0 \end{pmatrix} \tag{5}$$

is an eigenvector basis for $\boldsymbol{C}$ and

$$\boldsymbol{C}\boldsymbol{Z} = \boldsymbol{Z}\boldsymbol{\Lambda}, \quad \text{where} \quad \boldsymbol{\Lambda} = \begin{pmatrix} \bar{\boldsymbol{\Sigma}}_1 & 0 & 0 \\ 0 & -\bar{\boldsymbol{\Sigma}}_1 & 0 \\ 0 & 0 & 0 \end{pmatrix}\ . \tag{6}$$

The task of computing approximations to the singular values and vectors of $\boldsymbol{A}$ is therefore equivalent to computing approximations to the positive eigenvalues of $\boldsymbol{C}$ and their associated eigenvectors. Now since $\boldsymbol{C}$ is symmetric Algorithm 1 is applicable. This approach may seem very inefficient: Working with a matrix of dimension $(n+m) \times (n+m)$ in order to solve a problem of dimension $m \times n$. However, it turns out that by choosing a starting vector of the form $q_1 = (u_1, 0)^T \in \mathbb{R}^{m+n}$ then the Lanczos vectors generated by Algorithm 1 applied to $\boldsymbol{C}$ alternate between the two forms

$$q_{2j-1} = (u_j^T, 0)^T\ , \quad q_{2j} = (0, v_j^T)^T\ , \quad j = 1, \ldots, k\ ,$$

Now, the matrix vector multiplication $\boldsymbol{C}q_i$ never needs to operate on the zero parts of q_i, and by exploiting this and grouping the steps two by two the central steps in Algorithm 1 can be reduced to the compact form:

Algorithm 2. Lanczos bidiagonalization

1. Choose a starting vector $p_0 \in \mathbb{R}^m$, and let
   $\beta_1 = \|p_0\|_2$, $u_1 = p_0/\beta_1$ and $v_0 \equiv 0$
2. **for** $j = 1, 2, \ldots, k$ **do**
   - $r_j = \boldsymbol{A}^T u_j - \beta_j v_{j-1}$
   - $\alpha_j = \|r_j\|_2$
   - $v_j = r_j/\alpha_j$
   - $p_j = \boldsymbol{A} v_j - \alpha_j u_j$
   - $\beta_{j+1} = \|p_j\|_2$
   - $u_{j+1} = p_j/\beta_{j+1}$

   end

Algorithm 2 is generally known as the *Lanczos bidiagonalization* algorithm. We observe that replacing $\boldsymbol{A}$ by $\boldsymbol{C}$ does not increase the computational work per iteration of the Lanczos

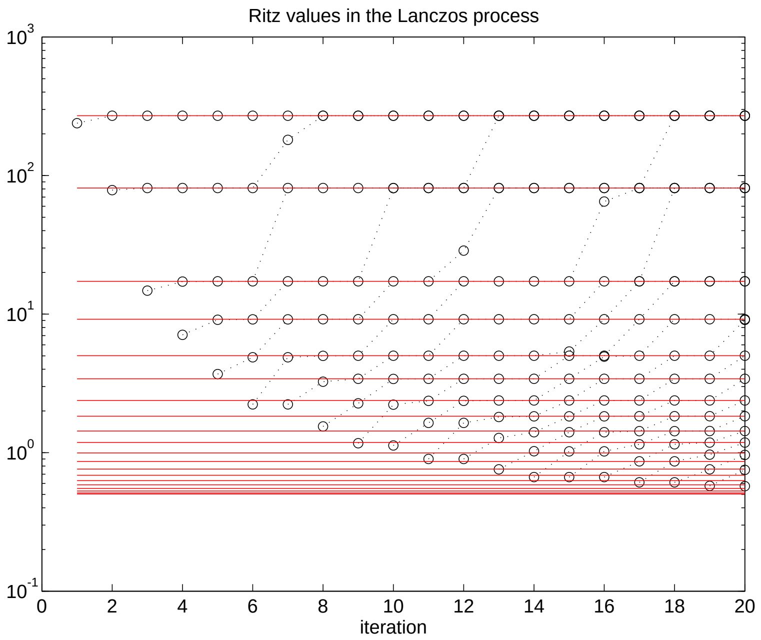

Figure 1: *Illustration of the convergence of eigenvalues values in the Lanczos process in finite precision arithmetic. Horizontal lines indicate the true values of the positive eigenvalues of* $\boldsymbol{C}$ *and circles show the eigenvalues values of* $\boldsymbol{T}_{2k}$. *The circles corresponding to the same eigenvalue of* $\boldsymbol{T}_{2k}$ *at different iterations* $k = 1, 2, \ldots, 20$ *have been connected by a dashed line to show how multiple copies and spurious "ghosts" that lie between the true eigenvalues occur during the iteration.*

algorithm, since only either $\boldsymbol{A}^T u_k$ or $\boldsymbol{A} v_k$ is computed in each iteration. After k iterations of Algorithm 2 we have computed (in theory) orthogonal Lanczos vectors

$$\boldsymbol{Q}_{2k} = (q_1, q_2, \ldots, q_{2k}) = \begin{bmatrix} u_1 & 0 & \cdots & u_k & 0 \\ 0 & v_1 & \cdots & 0 & v_k \end{bmatrix}$$

and the Lanczos matrix

$$\boldsymbol{T}_{2k} = \begin{pmatrix} 0 & \alpha_1 & & & \\ \alpha_1 & 0 & \beta_2 & & \\ & \beta_2 & \ddots & \ddots & \\ & & \ddots & \ddots & \alpha_k \\ & & & \alpha_k & 0 \end{pmatrix}$$

such that

$$\boldsymbol{C}\boldsymbol{Q}_{2k} = \boldsymbol{Q}_{2k}\boldsymbol{T}_{2k} + \beta_{k+1} q_{2k+1} e_{2k}^T$$

It follows from (5) and (6) that by computing the $p \le k$ largest eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_p$ of $\boldsymbol{T}_{2k}$ and the associated eigenvectors $s_1, s_2, \ldots, s_p$, we can obtain approximations to the largest singular values and vectors of $\boldsymbol{A}$ in the following way:

$$\sigma_i \approx \lambda_i \ , \quad \bar{u}_i \approx \sum_{j=1}^{k+1} u_j s_i(2j-1) \ , \quad \bar{v}_i \approx \sum_{j=1}^{k} v_j s_i(2j) \ , \qquad i = 1, \ldots, p \ , \tag{7}$$

and it can be shown that the error in the approximation λ_i can be bounded by

$$\min_{\sigma \in \sigma(\boldsymbol{A})} |\sigma - \lambda_i| \leq \beta_{ik} \equiv |\beta_{k+1} s_i(2k)| \ , \qquad i = 1, \ldots, p$$

where $s_1(2j), s_2(2j), \ldots, s_k(2j)$ are the bottom elements of the normalized eigenvectors of $\boldsymbol{T}_{2k}$, see e.g. [13].

In our implementation we use the standard bisection algorithm from [1] based on Sturm sequences to compute the positive eigenvalues of $\boldsymbol{T}_{2k}$. Then standard inverse iterations are used to compute the eigenvectors, see [18]. This approach is well suited for parallel implementations since each eigenvalue/vector pair can be computed independently of all the others.

It is well known that the Lanczos vectors q_1, q_2, ... loose their orthogonality in finite precision computations. As a consequence, isolated eigenvalues of $\boldsymbol{C}$ are approximated by numerically multiple eigenvalues of the tridiagonal matrices $\boldsymbol{T}_{2k}$. Moreover, spurious eigenvalues appear in $\boldsymbol{T}_{2k}$ in the sense that these eigenvalues are not approximations to any of those in $\boldsymbol{C}$. These phenomena are illustrated in Figure 1, which show eigenvalue approximations found in the first 20 iterations of the Lanczos of applied to the testmatrix used in Section 5.2. Regarding the multiple eigenvalues of $\boldsymbol{T}_{2k}$, we follow the standard approach and consider any multiple eigenvalue of $\boldsymbol{T}_{2k}$ as a converged eigenvalue of $\boldsymbol{C}$. The treatment of the spurious eigenvalues of $\boldsymbol{T}_{2k}$ is a more delicate problem. We use the identification test suggested by Cullum & Willoughby [7] based on a comparisons of the eigenvalues of $\boldsymbol{T}_{2k}$ and those of the $(2k-1) \times (2k-1)$ tridiagonal matrix $\hat{\boldsymbol{T}}_{2k}$ obtained from $\boldsymbol{T}_{2k}$ by deleting the first row and the first column. If an eigenvalue of $\boldsymbol{T}_{2k}$ is repeated in $\hat{\boldsymbol{T}}_{2k}$ then it is considered as a spurious eigenvalue, we refer to [7] for details.

The starting vector p_0 should preferably be chosen in such a way that it has large components along the left singular vectors that correspond to the large singular values of $\boldsymbol{A}$. In connection with eigenvalue problems, it is standard practice to use a random starting vector. In our applications, which arises in connection with least squares problems $\min \|\boldsymbol{A}x - b\|_2$ derived from first-kind integral equations, a good choice of p_0 is the right-hand side b. This choice ensures that p_0 has the desired SVD components, due to the fact that b satisfies a so-called "discrete Picard condition", cf. [14].

The complete algorithm, including the test for spurious eigenvalues, is outlined in Algorithm 3. The choice of the tolerance parameters ϵ_{spur} and ϵ is also discussed in [7]. The stopping criterion depends on the regularization used. If truncated SVD is used, the criterion could be that the fraction λ_p/λ_1 of the smallest to the largest converged, non-spurious eigenvalue of $\boldsymbol{T_{2k}}$, which gives the reciprocal of the effective condition number of the regularized system, should be smaller than some user defined tolerance. To reduce the work involved in monitoring the eigenvalues, they are only computed to test the stop criterion every k_{check} iterations.

Algorithm 3. Lanczos SVD

1. Choose a starting vector $p_0 \in \mathbb{R}^m$, and let
   $\beta_1 = \|p_0\|_2, \; u_1 = p_0/\beta_1$ and $v_0 \equiv 0$
2. stop=FALSE, $k = 1$
   while stop = FALSE **do**
   $r_k = \boldsymbol{A}^T u_k - \beta_k v_{k-1}$
   $\alpha_k = \|r_k\|_2$
   $v_k = r_k/\alpha_k$
   $p_k = \boldsymbol{A} v_k - \alpha_k u_k$
   $\beta_{k+1} = \|p_k\|_2$
   $u_{k+1} = p_k/\beta_{k+1}$
   if k **mod** $k_{\text{check}} = 0$ **then**

   Compute the k positive eigenvalues $\lambda_1, \lambda_2, \ldots \lambda_k$ of $\boldsymbol{T}_{2k}$

   Compute the eigenvalues $\hat{\lambda}_1, \hat{\lambda}_2, \ldots \hat{\lambda}_k$ of $\hat{\boldsymbol{T}}_{2k}$ and
   mark λ_i where $\min_j |\lambda_i - \hat{\lambda}_j| < \epsilon_{\text{spur}}$ as spurious

   Compute eigenvectors and error estimates β_{ik} for
   isolated, non-converged, non-spurious eigenvalues

   Mark eigenvalues λ_i which are either numerically multiple or
   non-spurious with $\beta_{ik} < \mu$ as converged

   stop = stopping_criterion($\{\lambda_i \,|\, \lambda_i$ is converged $\}$)

   end
   $k = k + 1$
   end
   Compute singular vectors according to (7)

3 The Convex Exemplar

The Lanczos bidiagonalization has been implemented on the Convex Exemplar installation at UNI•C in Århus. This computer has a relatively new architecture of which we give a brief overview in the following.

The Convex Exemplar belongs to the category of Non-Uniform Memory Access (NUMA), Memory coherent multiprocessors (cf. [19, p. 370]. It is based on the HP PA-RISC processors (with cycle time of 10ns) in configurations ranging from 8 to 128 processors in a system. The fundamental building block of the Exemplar is the *hypernode*. A hypernode consists of 8 processors, each with a 1 Mbyte *CPU cache*, connected by a crossbar to a 512 Mbytes shared *Hypernode private memory* (HPM). Several hypernodes can be interconnected using a high-speed network with a peak transfer rate of 667 Mbytes/second (the machine in Århus is a single hypernode system with 8 processors, hence the details concerning the inter-hypernode communication is of less interest in the present paper). The network and special memory coherence hardware provides the programmer with a view of the HPM's of all hypernodes as

Table 1: *Approximate latencies for access to data residing at different levels of the memory hierarchy.*

Level in hierarchy	**Latency**
CPU cache	10 ns
Hypernode private memory	500 ns
Global memory (in network cache)	500 ns
Global memory (*not* in network cache)	2μs

a single (globally) shared memory. Sharing is implemented on a *cache-line* basis, i.e. memory is allocated in chunks of 32 byte CPU cache lines which are interleaved across 8 independent memory banks, and a cache-line is also the amount of data transfered across the network when load or store operations are performed on data residing in other hypernodes. The caching is done on two levels:

1. The *CPU cache* holds data read by a single processor from the HPM. The CPU cache line size is 32 bytes.

2. The *Network cache* which holds data previously read from other hypernodes is located in the HPM, and is thus shared by all the processors of the hypernode. The size of the network cache is not fixed and can be reconfigured even while the system is running. Two CPU cache lines combine to make a single network cache line of 64 bytes.

On both caching levels coherence control is provided by hardware. This is done by ensuring that the following rules are met:

- Any number of read encachements of a cache line may be made at a single time.

- To store data into a cache line, the particular cache line must be owned exclusively. In this case a *purge* or *flush* command is broadcasted forcing copies residing in the cache of any other node to be deleted or written back to memory respectively.

- Modified cache lines are written back to memory before being deleted, or the contents are forwarded to the new owner.

For further details see [6].

The latency associated with loading and storing to memory vary significantly depending on where in the memory hierarchy the associated cache line is currently situated (hence the term *Non-Uniform Memory Access*). Table 1 shows the latency for each layer in the memory hierarchy. These are the most important specifications to keep in mind, when designing parallel applications for the Convex Exemplar.

The algorithm is implemented in ANSI C using the optimized BLAS-1 kernels in the CONVEX MLIB library, and the CPS subroutine library, see [4], [5]. The CPS library provides the basic primitives for thread management, synchronization and mutual exclusion, the latter implemented using either cache- or memory based semaphores. The most important operations available in the CPS library are summarized below.

Thread management

`cps_pp_call`: Spawn a specified number of threads (in our case one on each processors, but starting several threads on the same processor is possible) and make each one start executing a subroutine `func` supplied as an argument to `cps_pp_call`. An implicit synchronization is done after the execution of `func`; when all threads have exited `func` they are stopped, and the execution of the thread that called `cps_pp_call` is resumed.

`cps_stid`: Return thread id of the current thread.

`cps_nsthreads`: Return the number of threads in the current spawn context.

Synchronization

`cps_barrier`: Implements barrier synchronization. The calling thread is blocked until a specified number of threads have called `cps_barrier`.

Mutual exclusion

`cps_mutex_lock`: Can be used to implement critical regions. If the lock is already acquired by another thread the calling thread is blocked until the owner calls `cps_mutex_unlock`.

`cps_mutex_trylock`: Nonblocking version of `cps_mutex_lock`, with which a lock can be tried without blocking the thread. If the mutex is currently unlocked the thread acquires the lock, otherwise it returns immediately without waiting.

We will not use the low-level primitives listed above in the description of our implementations. Instead we will write our algorithm in Pascal-like pseudo code and use the parallel loop construct

```
for p = 1 to P pardo
    ...
endpar
```

which represents a parallelized `for`-loop where the P processors numbered 1 through P execute the body of the loop concurrently and with p equal to the local processor number. The parallel `for ... pardo` is implemented by placing the code corresponding to the body of a loop in a function, which is then called by all processors using the `cps_pp_call` function from the CPS library. Inside the function, the local processor number is computed using the `cps_stid` function (notice that we use the term processor instead of thread, since in our implementation one thread is executed on each processor).

In our pseudo code we will also use the construct

$$\texttt{accumulate}\ x = \sum_{j=1}^{P} x^{local(j)}$$

which computes in parallel in the vector x the sum of the vectors $x^{local(1)}, x^{local(2)}, \ldots, x^{local(P)}$. Although one cannot talk of "local" data on a shared memory machine, the superscript is used to indicate that the vectors $x^{local(1)}, x^{local(2)}, \ldots, x^{local(P)}$ are the results of independent calculations performed by each processor, and therefore $x^{local(i)}$ is likely to reside in the cache of processor number i. The construct is implemented using the standard recursive doubling algorithm (see e.g. [20]), which accumulates the local vectors by mapping the processors onto a binary tree as shown for the four processor case in Figure 2. During the recursive doubling $\lceil \log_2 P \rceil$ barrier synchronizations take place.

Both of the parallel constructs are followed by a barrier synchronization.

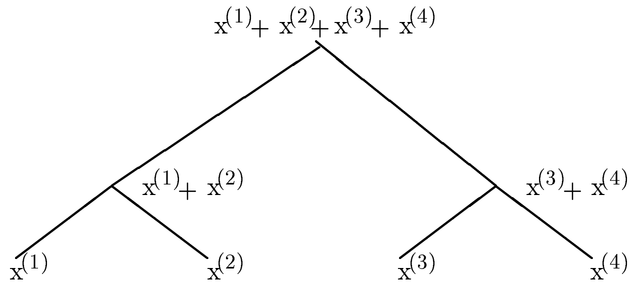

Figure 2: *Accumulation of "local" vectors.*

4 Parallelization of the Lanczos bidiagonalization

To create an efficient parallel implementation it is clearly important to understand the factors which limit the achievable performance. In all of todays high performance computers, including the Convex Exemplar, and increasingly powerful workstations the processor speed significantly exceeds memory bandwidth, [2]. This has lead to a widespread use of multi-layered memory hierarchies, which are designed to disguise this speed-gap by inserting small and fast (and expensive) buffer-memories between the main memory and the processor(s). This introduces a new factor to consider in the analysis of algorithm complexity, namely the cost of moving data between the different levels of the memory hierarchy. Thus for a hierarchical memory to provide any gains in computational performance a large amount of data reuse is crucial. As an example, Table 1 shows that the Convex Exemplar in the worst case, when each new element referenced is in a cache line located in another hypernode, runs at only 1/200 of its peak performance. Therefore, the number of data transports between layers in the memory hierarchy is perhaps *the* most important performance measure to consider when designing a new algorithm or, as in our case, when implementing a well known algorithm on a new machine.

Returning to the Lanczos algorithm we notice that the main computational work usually lies in the two matrix-vector products $v = \boldsymbol{A}^T u - \beta v$ and $u = \boldsymbol{A} v - \alpha u$ in Algorithm 3. In large scale applications the matrix is often sparse or structured and the matrix-vector multiplications are implemented as one or more subroutines that compute $\boldsymbol{A}x$ and $\boldsymbol{A}^T y$ without $\boldsymbol{A}$ being stored explicitly in memory. Examples include sparse matrices represented in a standard sparse matrix storage format, Toeplitz or circulant matrices where only the first row and the first columns are stored and the matrix-vector multiplication is be implemented using the fast Fourier transform algorithm. Another example is when the effect of the linear operators $\boldsymbol{A}$ and $\boldsymbol{A}^T$ is computed by solving a (partial) differential equation, say. Thus, in many cases, the parallelization of the matrix-vector multiplication is completely application independent. However, we have chosen to focus on a class of problems which still encompasses many applications. More specifically, we consider problems where either

1. the matrix is stored in some sparse matrix storage format, and the total amount of storage required by $\boldsymbol{A}$ is much larger than the amount of cache memory on the machine, or

2. the entries of $\boldsymbol{A}$ are generated on the fly in the matrix-vector multiplication subroutine. This includes applications where the matrix elements are given by an analytical expression, or can be constructed from a number of smaller matrices, e.g. when $\boldsymbol{A}$ is a Kronecker product.

For this class of problems we may use the Lanczos bidiagonalization with a dense matrix as a simple yet representative model problem. In the following, the cost associated with "cache-misses" can generally be thought of as the "the cost of accessing $\boldsymbol{A}$" – either by generating the entries on the fly or by copying them from main memory into the cache.

In the model problem, the two matrix-vector multiplications at the heart of the recursion are simple BLAS-2 operations i.e. they perform $O(mn)$ flops on $O(mn)$ data. Therefore the amount of data reuse that we can hope for is limited, and we must strive to reduce the number times data is loaded from main memory to the cache during the algorithm to a minimum. It turns out that the Lanczos algorithm can be restructured such that the number of cache misses is halved compared to a "naive" implementation of Algorithm 3. In the following let P denote the number of processors, S_c the number of double precision floating point numbers that fits in a single CPU cache and K the number of iterations. Further, assume that matrix $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ can be partitioned into $M \times N$ rectangular blocks $\boldsymbol{A}_{ij}$ of size $m_b \times n_b$ such that $Mm_b = m$ and $Nn_b = n$:

$$\boldsymbol{A} = \begin{pmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} & \dots & \boldsymbol{A}_{1N} \\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22} & \dots & \boldsymbol{A}_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{A}_{M1} & \boldsymbol{A}_{M2} & \dots & \boldsymbol{A}_{MN} \end{pmatrix},$$

and the vectors $u\ \mathbb{R}^m$ and $v \in \mathbb{R}^n$ are partitioned accordingly:

$$u = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_M \end{pmatrix}, \qquad v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_N \end{pmatrix},$$

where $u_i \in \mathbb{R}^{m_b},\ i = 1, \dots, M$ and $v_i \in \mathbb{R}^{n_b},\ i = 1, \dots, N$.

If the matrix in the problem is really a sparse matrix, the partitioning should be chosen on the basis of its non-zero pattern to balance the workload in the matrix-vector multiplication evenly among the processors. This is a well known problem, see [9], and is not important for our analysis. Notice: in the pseudo codes describing the parallel implementation, subscripts on the variables denote the block and not the iteration number which we omit.

4.1 Implementation 1

A fairly simple and straightforward implementation is obtained by dividing $\boldsymbol{A}$ block columnwise among the processors such that $N = P$, $n_b = n/N$, and choosing M such that a block of $\boldsymbol{A}$ and corresponding segments of u and v fit in the cache:

$$m_b n_b + n_b + m_b \leq S_c$$

This implies that we should choose $M < \frac{mn}{S_c}\frac{1}{P}$. Now the computation of $v = \boldsymbol{A}^T u - \beta v$ is straight forward. Processor p computes $v_p = \sum_{i=1}^{M} \boldsymbol{A}_{ip}^T u_i - \beta v_p$ in a block inner product fashion as described in Algorithm 4.

Algorithm 4. Matrix-vector multiplication $v = \boldsymbol{A}^T u - \beta v$

```
for p = 1 to P pardo
    v_p = -βv_p
    for i = p to p + M do
        q = i mod M
        v_p = v_p + A^T_qp u_q
    enddo
endpar.
```

The only data which is shared is u, but since multiple *read*-encachements do not cause any cache flushes this has no effect on performance. The computations in each processor can be regarded as being completely independent. The inner `for`-loop has a starting index that depends on the processor number, such that processor p visits the blocks in order $q = p, p+1, \ldots, P, 1, 2, \ldots, p-1$. This means that one processor tries to read a given segment of u at a time and delays caused by bank conflicts, where different processors try to read from the same memory address over the crossbar, are avoided.

If we keep the same blocking of $\boldsymbol{A}$ then $u = \boldsymbol{A}v - \alpha u$ can be computed in a block saxpy fashion as described in Algorithm 5.

Algorithm 5. Matrix-vector multiplication $u = \boldsymbol{A}v - \alpha u$

```
for p = 1 to P pardo
    for i = M + p to p step -1 do
        q = i mod M
        x_q^{local(p)} = A_qp v_p - αu_q
    enddo
endpar.
accumulate u = Σ_{j=1}^{P} x^{local(p)}
```

Notice that the order in which the segment of u is updated is reversed in Algorithm 5 as compared to Algorithm 4. In the Lanczos iteration Algorithm 5 and Algorithm 4 are executed alternately, and therefore this little trick saves approximately $2S_c$ loads from memory in every iteration because first block of $\boldsymbol{A}$ needed by Algorithm 5 is left in the cache by Algorithm 4 and vice versa.

4.2 Implementation 2

For large problem sizes ($mn \gg PS_c$) the implementation in Section 4.1 of the two matrix-vector products in the Lanczos algorithm gives rise to approximately $2mn/P$ cache misses – the block row used in each processor is loaded once for each of the matrix-vector products. Intuitively it should somehow be possible to load one block of $\boldsymbol{A}$ and then perform all operations in *both* matrix-vector products which involves elements of that particular block. One obvious problem is the fact that in the original formulation of the Lanczos bidiagonalization (Algorithm 2), the result of $\boldsymbol{A}v$ is used in the calculation of $\boldsymbol{A}^T u$ and vice versa. We discovered that with a simple reformulation of the recursion, it is possible to compute a segment of v_j and immediately afterwards use this intermediate result to accumulate a term in the sum for each segment in u_{j+1}. By doing this any given block of $\boldsymbol{A}$ only need to be loaded once per iteration. The modified recursion takes the following form:

Algorithm 6. Reordered Lanczos applied to $\boldsymbol{C}$

1. Choose a starting vector $y_0 \in \mathbb{R}^m$, and let
   $\beta_1 = \|y_0\|_2,\ \ u_1 = y_0/\beta_1$ and $v_0 \equiv 0$
2. **for** $j = 1, 2, \ldots, k$ **do**
   $x = \boldsymbol{A}^T u_j - \beta_j v_{j-1}$
   $y = \boldsymbol{A}x$
   $\alpha_j = \|x\|_2$
   $v_j = x/\alpha_j$
   $y = y/\alpha_j - \alpha_j u_j$
   $\beta_{j+1} = \|y\|_2$
   $u_{j+1} = u_{j+1}/\beta_{j+1}$
   end

By introducing the temporary vectors $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$, where

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} \quad \text{and} \quad y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_M \end{pmatrix} ,$$

we avoid overwriting results computed in one matrix-vector product which are needed in the other. Here $\boldsymbol{A}$ must be divided among the processors by block rows instead of block columns for the interleaving scheme to work, hence

$$M = P \ , \quad N = \frac{mn}{S_c - 2(n+m)} \frac{1}{P} \ .$$

The implementation of the parallel interleaved matrix-vector products for computing $x = \boldsymbol{A}^T u - \beta v$ and $y = \boldsymbol{A}x$ follows the pseudo-code in Algorithm 7. The correctness of the implementation can be proved by verifying that the statements in the two invariant are true after each iteration of the outer loop.

Algorithm 7. Parallel interleaved matrix-vector multiplications

for $p = 1$ to P pardo
    $y_p = 0$
endpar
for $q = 1$ to N do
    for $p = 1$ to P pardo
        $x_q^{local(p)} = \boldsymbol{A}_{pq}^T u_p$
    endpar
    accumulate $x_q = \sum_{j=1}^{P} x_q^{local(j)}$
    $x_q = x_q - \beta v_q$
    (* Invariant: $\forall j \in 1 \ldots q:\ x_j = \sum_{k=1}^{M} \boldsymbol{A}_{kj}^T u_k - \beta v_j$ *)
    for $p = 1$ to P pardo
        $y_p = y_p + \boldsymbol{A}_{pq} x_q$
    endpar
    (* Invariant: $\forall j \in 1 \ldots M:\ y_j = \sum_{k=1}^{q} \boldsymbol{A}_{jk} x_k$ *)
end.

5 Performance on the Convex Exemplar

In this section we discuss the performance of our implementation of the Lanczos bidiagonalization on the Convex Exemplar. The numerical results, i.e. the computed singular values and singular vectors, are quite satisfactory when compared to results obtained from a full SVD and we will not discuss this issue here.

5.1 Performance models

In this paragraph we present models for the execution time of the two implementations. The models are not intended to give accurate predictions of the execution times, but to help understanding the qualitative differences between the different implementations. Therefore, we have focused on the cost of memory-cache transports and the floating point operation (flops) involved. In the model we estimate the number of cache misses, the number of floating point operation and the number of synchronizations. Hence the models contain three machine dependent constants:

τ_{cm}: The average time to load a cache-line from memory and into the cache.

τ_a: The average time to perform an arithmetic operation, e.g., a floating point addition or multiplication.

$\tau_{sync}(P)$: The average time to synchronize P processors using the `cps_barrier` system call.

In constructing the models we have made the simplifying assumption that the bank conflicts (several CPU's trying to read from the same of the 8 memory banks) occur at randomly distributed points in time and that on the average they slow down each processor by approximately the same amount, such that the net result can be modeled by a higher memory latency, i.e. a higher value of τ_{cm}. Furthermore we have assumed that the synchronization time grows linearly with the number of processors, i.e. $\tau_{sync}(P) \equiv \tau_{sync} * (P-1)$.

In the models we must distinguish between to situations:

1. The problem is small enough to fit into the cache(s), $(PS_c \leq nm + n + m)$. The matrix is loaded only the first iteration, and remain there for the remaining iterations of the algorithm. Only the shared vectors, where forced cache flushes are necessary to ensure cache coherence, give rise to cache misses.

2. The problem does not fit into the cache(s) $(PS_c > nm + n + m)$. Part of the data are evicted from the cache and have to be reloaded every iteration.

Regarding cache misses, one should furthermore notice that data are fetched from memory by 32 byte cache line which equals 4 double precision floating point numbers. This means that if the elements of $\boldsymbol{A}$, say, are accessed in order (with unit stride), we get 3 "free" elements loaded for each cache miss. The expressions below were obtained by analyzing the code of the actual implementations.

Implementation 1: We recall that $N = P$, $M = \max(\frac{mn}{S_c-(n+m)}\frac{1}{P}, P)$. We get the following expressions for contributions to the execution time:

$$\begin{aligned} T_a &\approx \tau_a K((4nm + 7m + 3n)/P + 12n) \\ T_{sync} &= \tau_{sync} K(9 + \log_2 P) \end{aligned}$$

$PS_c \leq nm + n + m$:

$$T_{cm} \approx \frac{\tau_{cm}}{4} K(\max(0, 2mn/P - S_c) + m + n) + \frac{\tau_{cm}}{4} S_c$$

$PS_c > nm + n + m$:

$$T_{cm} \approx \frac{\tau_{cm}}{4}(\min(mn/P, S_c) + m + n)$$

Implementation 2: Here $N = \frac{mn}{S_c-(n+m)}\frac{1}{P}$, $M = P$, and we get the following model:

$$\begin{aligned} T_a &\approx \tau_a K((4nm + 7m + 3n)/P + 12n) \\ T_{sync} &= \tau_{sync} K(9 + N(\log_2 P + 3)) \end{aligned}$$

$PS_c \leq nm + n + m$:

$$T_{cm} \approx \frac{\tau_{cm}}{4} K(\max(0, mn/P - S_c) + m + n + n\log_2 P) + \frac{\tau_{cm}}{4} S_c$$

$PS_c > nm + n + m$:

$$T_{cm} \approx \frac{\tau_{cm}}{4}(\,\min(mn/P, S_c) + K(m + n + n\log_2 P)\,)$$

In both models the linear terms in n and m for T_{cm} are caused by forced cache flushes, issued when shared variables, e.g. u and v, are updated by one of the processors. The terms proportional to $\log_2 P$ in the model for the block-interleaved implementation 2 comes from the accumulation of the block inner products in Algorithm 7.

Comparing the two models we notice two major differences. The main difference lies in the number of cache misses where the dominating term (for loading the elements of $\boldsymbol{A}$) which is halved in implementation 2 compared to the implementation 1. For large problems we therefore expect a reduction in execution time compared to the implementation 1 by a similar amount. This is confirmed by our measurements in the next paragraph. Another conclusion which can be made on the basis of the models is that implementation 1 may be superior to implementation 2 in terms of speedup. This is a result of the extra forced cache flushes and the extra synchronizations needed during the accumulation of the N block inner products. However, since the terms involved are only proportional to $n \log_2 P$ this cost will be dwarfed by the terms proportional to nm as the problem size increases. In the next paragraph we present measured execution times and speedup for the two implementations, and compare this to the conclusions drawn from the simple models.

5.2 Experiments

In this section we present test results for the two implementations of the Lanczos SVD calculation on the Convex Exemplar. In the test we have used a simple $n \times n$ test matrix of the form

$$\boldsymbol{A} = \begin{pmatrix} 1 & \frac{n-1}{n} & \cdots & \frac{1}{n} \\ \frac{n-1}{n} & 1 & \ddots & \frac{2}{n} \\ \vdots & \ddots & \ddots & \vdots \\ \frac{1}{n} & \cdots & \frac{n-1}{n} & 1 \end{pmatrix}.$$

The discussion in this section are based on measurements of the execution time for 200 iterations of the Lanczos algorithm for varying problem size n and number of processors P. As mentioned earlier, the time spent computing and "filtering" the singular values is (especially for large problems) insignificant. Therefore the execution times reported are for the "bare" Lanczos process. The measured execution times tend to fluctuate a bit due to the interference from operating system processes running on the nodes.

The constants τ_a and τ_{cm} in the models were estimated by fitting the models below to the measured execution time on a single processor. Then τ_{sync} was estimated by keeping τ_a and τ_{cm} fixed and adjusting τ_{sync} to make the model fit the execution times with different number of processors. This gave the values $\tau_a \approx 20ns$, $\tau_{cm} \approx 650ns$ and $\tau_{sync} \approx 5\mu s$. The value of τ_a, which according to according to Table 1 indicates that on average a floating point instruction takes 2 clock cycles to execute, which in only 50% of the peak performance of the processor. However, in the model we have ignored all integer operation and load and store operation for operands in the cache, hence the value τ_a really includes these contributions. The value of τ_{cm} is also somewhat higher than the value in Table 1. This is more difficult to explain. Either the higher latency must be attributed to bank conflicts due to processes running on the remaining nodes or else we have underestimated the number of cache misses in the model. Another probable explanation is that the value of $500ns$ provided by the manufacturer is a little too optimistic.

Figure 3 shows the execution time of implementation 2 for 1, 3 and 7 processors respectively in a log-log plot. Crosses show the measured data and the solid line show the values predicted by the model. In each case we notice the abrupt transition at the critical point $n_c \approx \sqrt{PS_c}$, where the problem does no longer fit in the cache. The overall form of the plots are as one would expect from the models: for problems smaller than n_c a straight line mainly

Table 2: Relative reduction in running time from implementation 1 to implementation 2 in percent.

	$P = 1$	$P = 2$	$P = 3$	$P = 4$	$P = 5$	$P = 6$	$P = 7$
100×100	60	55	43	39	-15	8	-5
500×500	31	23	45	52	40	51	46
1000×1000	50	35	35	38	25	16	-4
2000×2000	59	62	30	11	10	10	7
5000×5000	67	47	45	49	50	40	34

dominated by the cost of the arithmetic operations ($T_a \propto n^2$ i.e. slope is 2), with fairly large contributions from the parallel overhead (synchronization) as the number of processors increase. For problems sizes above the transition zone (the region where a substantial part of, but not all of the problem fits in the cache), the asymptotic behavior is also a straight line with slope 2. The constant of proportionality, however, is larger since now the execution time is dominated by the term proportional to the number of cache misses ($\tau_{cm} \approx 50\tau_a$) – the machine is running at "memory speed". This should be compared with Figure 4 where we have plotted the contributions from the different terms in the model for a single and 7 processors. The solid line is T_a, the dashed line is T_{cm} and the dotted is T_{sync}.

Table 2 illustrates the reduction in execution time when replacing implementation 1 by 2. What is most striking is the great improvement when solving large problems on a small number of processors. This because we benefit the most from the interleaving scheme in implementation 2 when the total amount of cache memory, which is proportional to the number of processors, is small compared to the problem size. In this case the execution time is almost completely dominated by the time spent moving data between the cache and the main memory and the reduction in the number of cache misses is felt most strongly. For small problems (and who would use a parallel computer for them anyway?), where the total amount of cache memory is comparable to the problem size, implementation 2 suffers from the increased parallel overhead introduced in rewriting the recursion; implementation 1 is almost embarrassingly parallel and has therefore a higher speedup rate (see below).

In Figure 5 we have plotted the measured speedup, which was calculated as

$$S = \frac{T(P)}{T(1)} \ ,$$

where $T(P)$ is the execution time of the parallel program using P processors. The measured speedup for the 2 implementations is also listed in Table 3. Comparing the two panels in 5 we notice how the parallel overhead in implementation 2 (caused by the many synchronization points) degrades the speedup rate compared to that of implementation 1, even for large problems; but still the *execution time* is reduced by 30-60%, as can be seen more clearly in Figure 6. Some of the results for implementation 2 look a little peculiar; for example we do not have any explanation for the relative poor speedup for $n = 2000$. It is not unlikely that some of the explanation is disturbances from processes running on the remaining nodes.

We also observe another interesting phenomenon, namely the superlinear speedup in the left panel Figure 5. This is a characteristic behavior for this type of machine, and is due to the fact that as we increase the number of processors we also increase the total amount of cache memory available. The matrix in a 500×500 problem, say, occupies $500 \times 500 \times 8 = 2$ Mbyte

Table 3: Measured speedup.

Implementation 1						
Size	$P=2$	$P=3$	$P=4$	$P=5$	$P=6$	$P=7$
100×100	0.91	0.87	0.87	1.2	0.77	0.71
500×500	7.5	9.0	10.0	14.2	11.6	11.9
1000×1000	2.1	3.2	4.5	7.3	10.1	15.0
2000×2000	1.8	3.3	4.7	5.7	6.8	7.7
5000×5000	2.4	3.9	5.6	7.2	8.5	10.0
Implementation 2						
Size	$P=2$	$P=3$	$P=4$	$P=5$	$P=6$	$P=7$
100×100	0.78	0.59	0.52	0.4	0.33	0.26
500×500	6.7	11.1	14.3	16.3	16.2	15.2
1000×1000	1.6	2.4	3.6	4.8	6.0	7.3
2000×2000	2.0	1.9	2.2	2.6	3.1	3.4
5000×5000	1.5	2.4	3.7	4.8	4.8	5.0

of memory. It does not fit in the cache of a single processor and we get a considerable contribution to the execution time from T_{cm} - the machine runs at "memory speed". Using 2 processors (or more), the entire matrix fits in the caches and will only be loaded from the shared memory once. Subsequent calculations can load their operand directly from the cache and are thus performed at "processor speed". This obviously gives quite extreme speedup values when the matrix dimension is greater than 350, which is the largest matrix fitting in a single cache. As the problem size increases the matrix size will eventually exceed the total amount of cache, and this effect will gradually disappear. As can be seen in Table 3 this effect is most pronounced for implementation 1, which is to be expected since this formulation of the algorithm does not utilize the cache as efficiently as implementation 2 and thus has worse single processor performance on large problems.

6 Conclusion

One of the conclusions which can be drawn from this work is that when designing efficient algorithms for a parallel machine like the Convex Exemplar, one cannot rely entirely on traditional performance measures such as the number of arithmetic operations (flops) and speedup. Instead the cost associated with memory hierarchy transports, e.g. cache misses, must be taken into account. This is not possible without a certain degree of insight into the architectural peculiarities of the specific machine. In our case, reducing the number of cache misses and obtaining high speedup rates are mutually conflicting demands. In fact, the superlinear speedup obtained with the simple implementation 1, is primarily an indication of its poor cache performance.

7 Acknowledgments

The authors would like to thank Peter Kaae from Convex Denmark for supplying us with documentation of the CPS library, and Niels Carl Hansen from UNI•C for creating a dedicated partitioning on the Convex Exemplar for our experiments.

References

[1] W. Bath, R. S. Martin & J. H. Wilkinson, *Calculation of the Eigenvalues of a Symmetric Tridiagonal Matrix by the Method of Bisection*, Numerische Mathematik **9**, 1967.

[2] F. Baskett and J. L. Hennesy, *Microprocessors: From Desktops to Supercomputers*, Science, **261** (1993), 864–870.

[3] J. Christensen-Dalsgaard, P. C. Hansen and M. J. Thompson, *GSVD analysis of helioseismic inversions*, Mon. Not. R. Astr. Soc. **264** (1993), 541–465.

[4] Convex Computing Corporation, *Convex Exemplar Programming Guide*, March 1994, DSW-067.

[5] Convex Computing Corporation, *cps - parallel compiler support library*, Manual pages, March 1994.

[6] Convex Computing Corporation, *Exemplar Architecture*, November 1993, DHW-014.

[7] J. K. Cullum and R. A. Willoughby, *Lanczos Algorithms for Large Symmetric Eigenvalue Computations. Vol. I Theory*, Birkhäuser, Boston, 1985.

[8] J. Cullum, R. Willoughby and M. Lake, *A Lanczos algorithm for computing singular values and vectors of large matrices*, SIAM J. Sci. Statist. Comput. 4 (1983), no. 2, 197–215.

[9] Dongarra, J., Duff, I. S., Sorensen, D. C., van der Vorst, H. A., 1991, *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM, Philadelphia.

[10] J. J. Dongarra & H. A. van der Vorst, *Performance of various computers using standard techniques for solving sparse linear equations*, SuperComputer **51**, IX–5 (1992).

[11] R. D. Fierro, G. H. Golub, P. C. Hansen, D. P. O'Leary, *Regularization by truncated total least squares*, submitted to SIAM J. Sci. Comput., **18** (1997), 1223–1241.

[12] G. H. Golub and W. Kahan, *Calculating the singular values and pseudo-inverse of a matrix*, J. Soc. Indust. Appl. Math. Ser. B Numer. Anal. **2** 1965 205–224.

[13] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 2. Ed., Johns Hopkins University Press, Baltimore, 1989.

[14] M. Hanke and P. C. Hansen, *Regularization methods for large-scale problems*, Surv. Math. Ind. **3** (1993), 253–315.

[15] P. C. Hansen, *Experience with Regularizing CG Iterations*, Report UNIC-94-05, UNI•C, 1994.

[16] C. C. Paige and M. A. Saunders, *LSQR: an algorithm for sparse linear equations and sparse least squares*, ACM Trans. Math. Software **8** (1982), 43–71.

[17] C. C. Paige and M. A. Saunders, *Algorithm 583. LSQR: sparse linear equations and least squares problems*, ACM Trans. Math. Software **8** (1982), 195–209.

[18] G. Peters and J. H. Wilkinson, *The Calculation of Specified Eigenvectors by Inverse Iteration*, in J. H. Wilkinson and C. Reinsch (eds.) “Handbook for Automatic Computation. Volume II”, Springer Verlag, Berlin, Heidelberg, New York, 1971.

[19] A. S. Tannenbaum, *Modern Operating Systems*, Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[20] E. F. Van de Velde, *Concurrent Scientific Computing*, Springer-Verlag, New York, 1994.

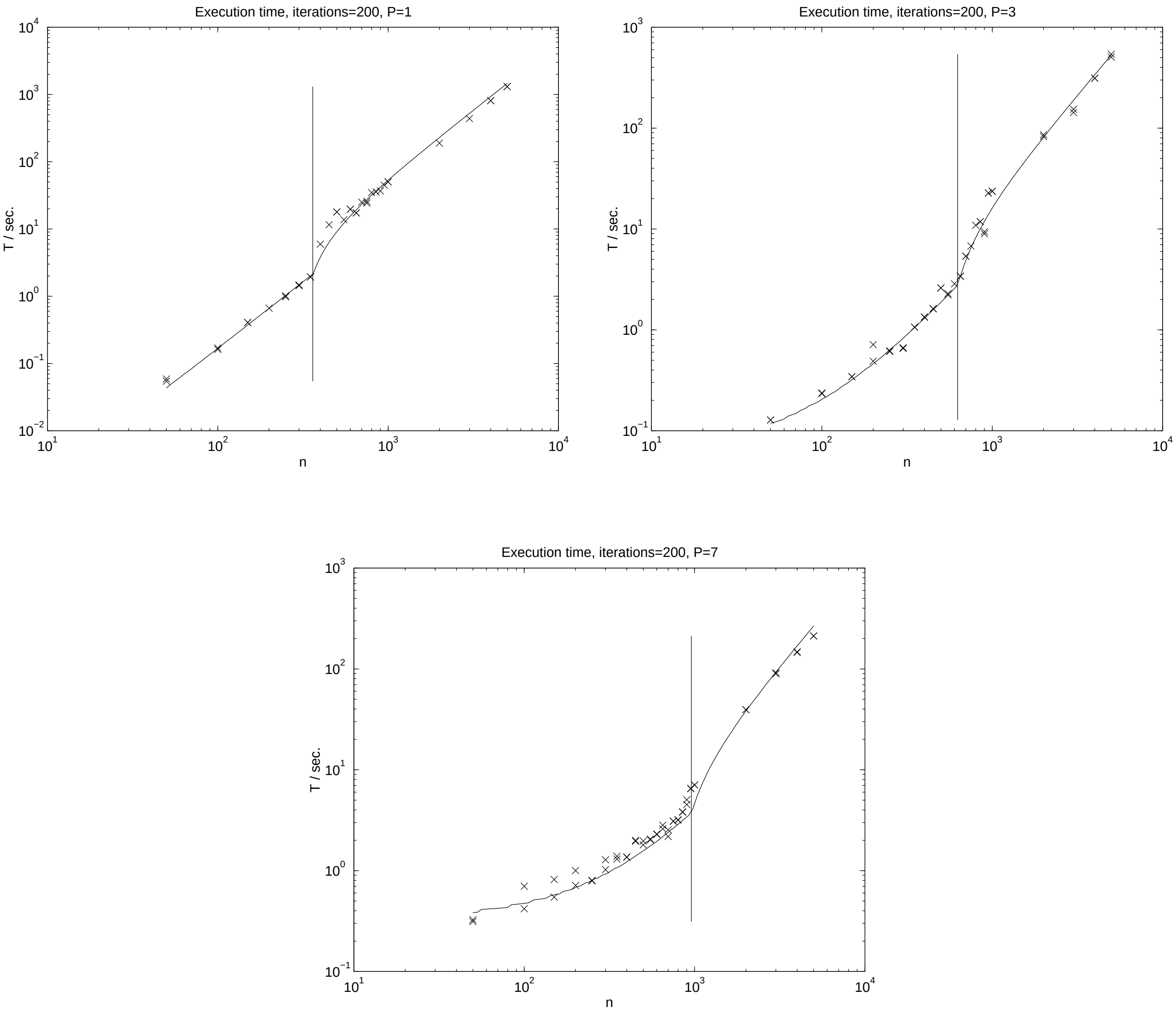

Figure 3: *Execution time versus problem size. Implementation 2. Crosses show measurements and the solid line shows model predictions. Top left panel is for* $P = 1$ *processor, top right is for* $P = 3$ *and the lower panel is for* $P = 7$.

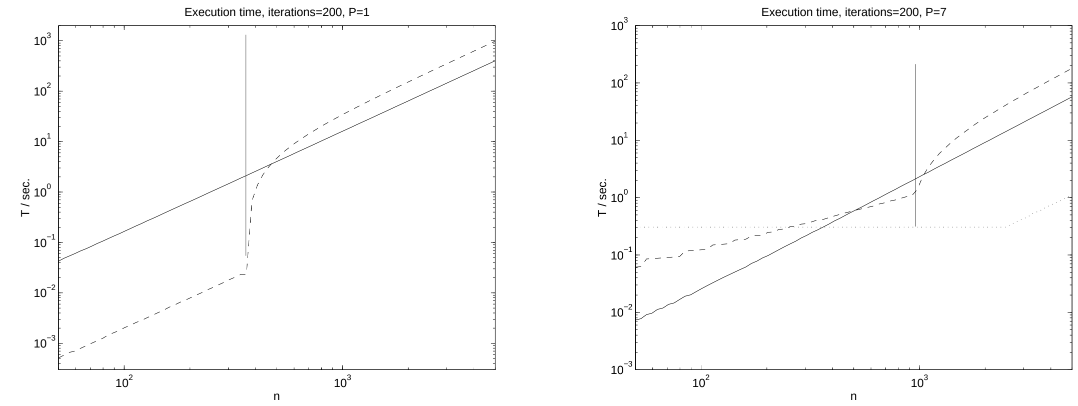

Figure 4: *Contributions to the execution time of implementation 2 from different terms in the model versus problem size. Left panel shows T_a (solid), T_{sync} (dashed) and T_{cm} (dashed) for 1 processor. Right panel shows the same for 7 processors.*

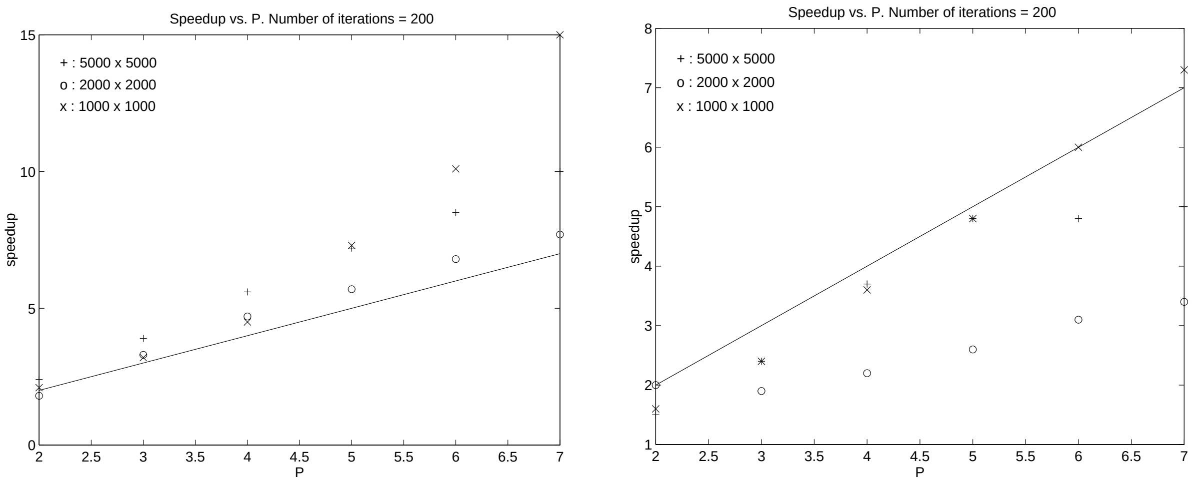

Figure 5: *Speedup versus number of processors. Left panel shows measured speedup for implementation 1. Right panel shows measured speedup for implementation 2. The solid line corresponds to linear speedup $S = P$.*

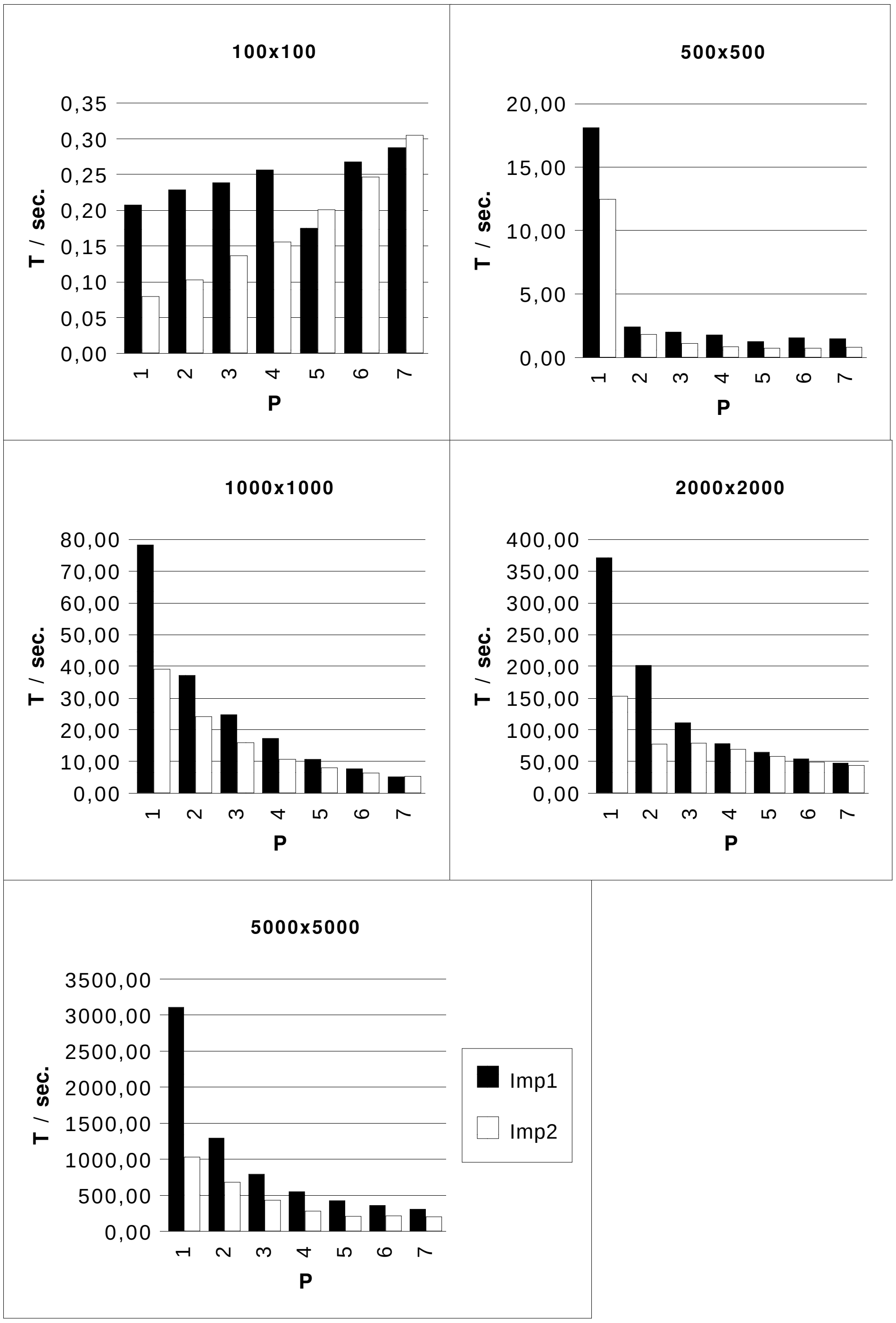

Figure 6: *Execution time for 200 iterations. Each subplot shows the execution time of implementation 1 and implementation 2 for a given problem of fixed size executed on 1 through 7 processors.*