High Performance Computing
in Material Sciences

—

Maximizing Cache Utilization without Increasing Memory Requirements

Ernst Haunschmid
Institute for Applied and Numerical Mathematics
University of Technology Vienna
ernst@uranus.tuwien.ac.at

Dieter F. Kvasnicka
Institute for Technical Electrochemistry
University of Technology Vienna
kvasnicka@tuwien.ac.at

November 1998

AURORA TR1998-17

The work described in this paper was supported by the Special Research Program SFB F011 "AURORA" of the Austrian Science Fund.

Abstract

This report describes how the program `lapw1` of the computational chemistry package WIEN 97 has been transformed to a high performance version using newly developed routines that make use of Level 3 BLAS and introduce several ways of blocking. The new routines (1) have an increased locality of reference and thus achieve improved floating-point performance, and (2) maintain the storage efficiency of the original version by utilizing a new storage scheme which has only about half the memory requirements of the corresponding LAPACK routines. The new routines speed up the sequential code `lapw1` by a factor of two on many computer systems (cache based as well as vector based architectures), and, equally important, enable the parallelization of the code with nearly linear speedup.

Contents

Chapter 1

Introduction

WIEN 97

The *full potential linearized augmented plane wave* (FLAPW) method, implemented in the material science code WIEN 97, is a widely used method in computational chemistry (see Blaha et al. [4] and Singh [14]). This method has proved to be a very accurate scheme for computing the electronic structure of solids within density functional theory.

In this report various techniques for improving the floating-point performance of the most time consuming part of WIEN 97, the eigensolver `lapw1`, are described for real and complex data.[1]

Synopsis

Chapter 2 gives an overview over the computational requirements of WIEN 97 as well as a short introduction to the fundamentals of density functional theory, as far as needed to understand the principles of WIEN 97.

In Chapter 3 run time measurements of the highly tuned original (real) version of `lapw1` are compared with run times of an `lapw1` version using LAPACK routines. LAPACK routines are not performing well in this context, because of the packed data format. This data format leads to algorithms which rely mostly on Level 2 BLAS, which are less efficient than Level 3 BLAS.

In Chapter 4 methods for increasing storage and run time efficiency are discussed. New storage formats and hierarchically blocked algorithms are introduced.

[1] Solids with inversion symmetry lead to real matrices, whereas solids without inversion symmetry lead to complex matrices.

In Chapter 5 the most time consuming LAPACK routines for real data are reformulated using a block packed storage scheme and utilizing hierarchical blocking. Measured run time values for large matrices show a speedup over the original (highly tuned) program by a factor of two. In Chapter 6 corresponding routines for complex data are described.

The new routines described in this report were developed on an SGI Power Challenge XL (using one MIPS R10000 processor and 2MB cache), but they achieve similar performance improvements on other computer systems (workstations or vector computers) as well. Chapters 7 and 8 present performance results for the newly developed routines as well as for the whole `lapw1` program, running on various computer systems. Chapter 9 deals with the fine tuning of the new routines for different computer architectures.

In various appendices performance differences with respect to various features of computer architecture, the ideas behind the new routines, details about performance tuning, the naming conventions, and the reliability of the time measurement are discussed.

Chapter 2

The Material Science Code WIEN 97

The *linearized augmented plane wave* (LAPW) method is one of the most accurate methods for the computation of the electronic structure of solids based on *density functional theory* (see Singh [14]). A full potential LAPW code for crystalline solids has been developed in the last ten years at the Institute for Technical Electrochemistry of the Vienna University of Technology. In 1990 the first copyrighted version was named WIEN (Blaha et al. [4]).

In the last few years, WIEN 93, WIEN 95, and WIEN 97, significantly improved and updated UNIX versions of the original WIEN code, were developed. The current version, WIEN 97, is the most sophisticated one, especially in terms of speed, versatility, user-friendliness and new features (for example, to compute optical properties, to optimize geometrical properties, and to obtain higher accuracy using spin-orbit coupling).

WIEN 97 is written in Fortran 77 and requires a UNIX operating system since several programs are linked together via C-shell scripts.

Hardware requirements change from case to case. Small cases with 10 atoms per unit cell can be run on a Pentium PC with 64 MB main memory under Linux, but generally a powerful workstation is recommended with 256 MB (or more) main memory and 1–4 GB of disk space.

The first sections of this chapter give a short overview over density functional theory which is the mathematical basis of WIEN 97.

The rest of this chapter describes how the program `lapw1`, which performs eigenproblem computations, is embedded in the WIEN 97 program package, and how its floating-point performance can be improved.

2.1 Density Functional Theory

The theorem upon which *density functional theory* and *local density approximation* are based is that of Hohenberg and Kohn[1] [9]. It states that the total energy E of a non-spin-polarized system of interacting electrons in an external potential (in the following, the Coulomb potential due to the nuclei in a solid) is given exactly as a functional of the ground state electronic density ρ, i. e.,

$$E = E[\rho]. \tag{2.1}$$

Hohenberg and Kohn showed that the true ground state density is the density that minimizes $E[\rho]$, and that the other ground state properties (for example, optical properties or X-ray spectra) are also functionals of the ground state density. The extension to spin-polarized systems is straightforward; E and the other ground state properties become functionals of both the up and down spin densities, i. e.,

$$E = E[\rho_\uparrow, \rho_\downarrow]. \tag{2.2}$$

Unfortunately, the Hohenberg-Kohn theorem provides no guidance as to the form of $E[\rho]$, and therefore the utility of density functional theory depends on the discovery of sufficiently accurate approximations. In order to do this, the unknown functional $E[\rho]$ is rewritten as the Hartree total energy $T_s[\rho]+E_{ei}[\rho]+E_{ii}[\rho]+E_H[\rho]$ plus another, but presumably smaller, unknown functional, called the exchange correlation (xc) functional $E_{xc}[\rho]$:

$$E[\rho] = T_s[\rho] + E_{ei}[\rho] + E_{ii}[\rho] + E_H[\rho] + E_{xc}[\rho]. \tag{2.3}$$

T_s denotes the single particle kinetic energy, E_{ei} is the Coulomb interaction energy between the electrons and the nuclei, E_{ii} arises from the interaction of the nuclei with each other, and E_H is the Hartree component of the electron-electron energy, i. e.,

$$E_H[\rho] = \frac{e^2}{2}\int d^3r d^3r' \frac{\rho(r)\rho(r')}{|r-r'|}. \tag{2.4}$$

In the local density approximation, $E_{xc}[\rho]$ is written as

$$E_{xc}[\rho] = \int d^3r \rho(r)\epsilon_{xc}(\rho(r)) \tag{2.5}$$

[1] Walter Kohn received the Nobel price for chemistry 1998 for his work on density functional theory.

where $\epsilon_{xc}(\rho)$ is approximated by a local function of the density, usually a function which reproduces the known energy of the uniform electron gas.

Kohn and Sham [11] wrote the electron density $\rho(r)$ as a sum of single particle densities, and determined the ground state energy $E[\rho]$ and density ρ. Thus, instead of having to solve a many-body Schrödinger equation, using density functional theory, only the solution of a series of single particle equations (satisfying a self-consistency condition) is required. In particular, Kohn and Sham showed that the correct density $\rho(r)$ is given by the self-consistent solution of a set of single particle Schrödinger equations, known as the *Kohn-Sham equations*, with a density dependent potential,

$$H\phi_i(r) = (T + V_{ei}(r) + V_H(r) + V_{xc}(r))\phi_i(r) = \epsilon_i\phi_i(r), \tag{2.6}$$

with the density ρ given by a sum over the occupied orbitals (Fermi sum), i. e.,

$$\rho(r) = \sum_{occ} < \phi_i(r)|\phi_i(r) > . \tag{2.7}$$

$< \cdot|\cdot >$ denotes an appropriate scalar product. Here the highest occupied orbital is determined by the electron count, the ϕ_i represent single particle orbitals, ϵ_i are the corresponding eigenvalues, T is the kinetic energy operator, V_{ei} is the Coulomb potential due to the nuclei, V_H is the Hartree potential and V_{xc} is the exchange correlation potential. Both V_H and V_{xc} depend on ρ, i. e.,

$$V_H(r) = e^2 \int d^3r' \frac{\rho(r')}{|r - r'|} \tag{2.8}$$

and

$$V_{xc}(r) = \frac{\partial E_{xc}[\rho]}{\partial\rho(r)}. \tag{2.9}$$

In this framework, the self-consistent solution of the Kohn-Sham equations, (2.6) and (2.7), can be obtained. That is, a density ρ must be found which yields an effective potential $E[\rho]$ that when inserted (using equations (2.8) and (2.9)) into the Schrödinger-like equations (2.6) yields orbitals that reproduce the potential.

2.2 Solution of the Single Particle Kohn-Sham Equations

Density functional theory based electronic structure methods are classified according to the representations used for the density ρ, for the potential $E[\rho]$ and, most importantly, for the the Kohn-Sham orbitals ϕ_i. Particular representations are chosen to minimize the computational costs and human (e. g., programming) effort, while maintaining sufficient accuracy. These competing, material and application dependent goals have led to the development and use of a wide variety of techniques, one of them being the LAPW method.

One way is to represent the orbitals ϕ_i in terms of basis functions. It is possible to avoid the explicit use of a basis in constructing the Kohn-Sham orbitals, for example, by numerically solving the differential equation (2.6) on grids. However, nearly all approaches that have been proposed for solids, including the LAPW method, do rely on basis set expansions for the Kohn-Sham orbitals ϕ_i, because well chosen basis sets turn out to be very efficient. In this report the discussion is confined to methods which use an infinite expansion. There is a heuristic method for determining where to truncate the infinite expansion, i. e., for determining the number n_b of basis functions $\Phi_\alpha(r)$. Thus the Kohn-Sham orbitals ϕ_i are represented as

$$\phi_i(r) = \sum_{\alpha \leq n_b} c_{i\alpha} \Phi_\alpha(r) \ , \tag{2.10}$$

where the $c_{i\alpha}$ are expansion coefficients. Given a certain basis, these coefficients are the only variables in the problem (note, according to (2.7) the density ρ depends only on the Kohn-Sham orbitals ϕ_i). Using the theorem of Hohenberg and Kohn, a self-consistent solution to the Kohn-Sham Equation (2.6) can be found by determining the coefficients $c_{i\alpha}$ in Equation (2.10) that minimize the total energy $E[\rho]$.

The energy functional $E[\rho]$ can be rewritten using the single particle eigenvalues ϵ_i to eliminate the unknown functional $T_s[\rho]$ in Equation (2.3):

$$E[\rho] = E_{ii}[\rho] + \sum_{occ} \epsilon_i + E_{xc}[\rho] - \int d^3r \rho(r)(V_{xc}(r) + \frac{1}{2}V_H(r)) \ , \tag{2.11}$$

where the sum is over the occupied orbitals, and ρ, V_H, and V_{xc} are given by Equations (2.7), (2.8), and (2.9), respectively.

The optimization of the $c_{i\alpha}$ and the determination of the self-consistent charge density ρ can be done separately. In this scheme it is necessary to repeatedly determine the $c_{i\alpha}$ that solve the single particle equations (2.11) for a fixed charge density. This may be done using standard techniques from numerical linear algebra. Specifically, given the basis functions $\Phi_\alpha(r)$, the Hamiltonian and the overlap matrices, H and S, are constructed (see Section 2.2.1) and the eigenvalue problem

$$(H - \epsilon_i S)c_i = 0 \ , \tag{2.12}$$

is solved. The rank of the square matrices H and S is equal to the number of basis functions, n_b, and the c_i are eigenvectors containing the n_b coefficients $c_{i\alpha}$.

If the occupied Kohn-Sham orbitals can be expressed as linear combinations of the chosen basis functions, then optimizing the $c_{i\alpha}$ will yield the exact self-consistent solution. In practice, it is impossible to find such a set of basis functions. The quality of the basis chosen determines the accuracy of the solution from this method. Each approximate solution is optimal in the sense that it gives the lowest possible total energy $E[\rho]$ for a given basis. The usefulness of a basis set can, therefore, be measured by the extent to which the total energy evaluated using the orbitals ϕ_i of equation (2.10) differs from the true Kohn-Sham energy $E[\rho]$. Using larger sets of basis functions can improve the accuracy of the solution, at the cost of higher computational effort.

2.2.1 Synthesis of the Hamiltonian and Overlap Matrices

The matrix elements, $S_{GG'}$ and $H_{GG'}$ (overlap matrix and Hamiltonian, respectively) are given as

$$S_{GG'} = < \Phi_G | \Phi_{G'} > \tag{2.13}$$

and

$$H_{GG'} = < \Phi_G | E | \Phi_{G'} > = < \Phi_G(r) | E[\Phi_{G'}(r)] > = < E[\Phi_G(r)] | \Phi_{G'}(r) > \quad . \tag{2.14}$$

These matrices may be decomposed into interstitial and sphere components (see Section 2.3).

The LAPW basis functions allow to exploit inversion symmetry, if present, by choosing the origin of the unit cell to coincide with an inversion center. In general, H and S are complex Hermitian matrices. If inversion symmetry is present, H and S become real symmetric matrices, making their construction and diagonalization easier than for the complex Hermitian case.

2.2.2 Self-Consistency in Density Functional Calculations

As mentioned in Section 2.1, the Hohenberg-Kohn theorem shows that the total energy $E[\rho]$ has a minimum in terms of ρ, and this is the key to its usefulness. The true ground state density ρ is that density which minimizes the energy $E[\rho]$. This is the basis of the standard self-consistency cycle. Given a charge density ρ_{in}, equation (2.12) is diagonalized, ensuring that the orbitals ϕ_i are orthonormal and that no orbitals are missed. This eliminates almost all local extrema of $E[\rho]$. An output charge density ρ_{out} is constructed from the eigenvectors c_i using Equations (2.7) and (2.10), and then mixed with the input charge density ρ_{in}

to yield a refined input charge density ρ_{in} for the next iteration. The simplest mixing scheme is straight mixing

$$\rho_{in}^{i+1} = (1-\alpha)\rho_{in}^{i} + \alpha\rho_{out}^{i} \ , \tag{2.15}$$

where the superscript refers to the iteration number and α is the mixing parameter. For sufficiently small α the iterations converge. However, the region of convergence can be small, and becomes smaller rapidly as the size of the unit cell (see Section 2.3) increases, particularly for metallic systems. As a result, considerable effort has been devoted to devising more sophisticated mixing procedures, using information from previous iterations to accelerate the convergence.

2.3 The Full Potential LAPW method

Like most "energy-band methods", the LAPW method is a procedure for solving the Kohn-Sham equations for the ground state density ρ, total energy $E[\rho]$, and (Kohn-Sham) eigenvalues ϵ_i (energy bands; see equation (2.6)) of a many-electron system by introducing a finite basis set which is especially adapted to the problem.

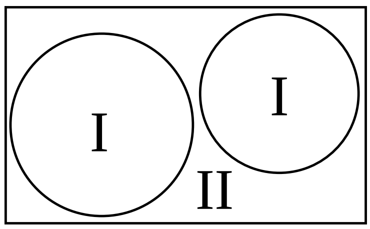

Figure 2.1: Partitioning of the unit cell into atomic spheres (I) and an interstitial region (II).

In the case of a crystal, a unit cell is defined (see Fig. 2.1). The adaptation of the basis set is achieved by dividing the unit cell into (I) non-overlapping atomic spheres (centered at the atomic sites) and (II) an interstitial region. In the two types of regions different basis sets $\{\Phi_\alpha\}$ are used.

(I) Inside an atomic sphere t of radius R_t, a linear combination of radial functions u_l and $\dot{u}_l$ times spherical harmonics $Y_{lm}(r)$ is used (the index t is omitted), i. e.,

$$\Phi_\alpha = \sum_{lm} [A_{lm}^{\alpha} u_l(r, E_l) + B_{lm}^{\alpha} \dot{u}_l(r, E_l)] Y_{lm}(\hat{r}) \ , \tag{2.16}$$

where $u_l(r, E_l)$ is the (at the origin) regular solution of the radial Schrödinger equation for energy E_l (chosen usually at the center of the corresponding band with its main contribution due to quantum number l) and the spherical part of the potential inside sphere t; $\dot{u}_l(r, E_l)$ is the energy derivative of u_l taken at the same energy E_l. A linear combination of these two functions constitutes the linearization of the radial functions; the coefficients A^α_{lm} and B^α_{lm} are functions of α determined by requiring that this basis function matches (in value and slope) the corresponding basis function of the interstitial region; u_l and $\dot{u}_l$ are obtained by numerical integration of the radial Schrödinger equation on a radial mesh inside the sphere.

(II) In the interstitial region plane waves,

$$\Phi_\alpha = \frac{1}{\sqrt{\omega}} e^{i\alpha r} \ , \tag{2.17}$$

are used as basis functions.

2.4 Programs of WIEN 97

A test case, rutile (titanium dioxide TiO_2, a common mineral), provided in the WIEN 97 package, was run on a workstation with an Alpha 21164A processor. The execution times of the programs of WIEN 97 are presented in Figure 2.2, and Tables 2.1 and 2.2.

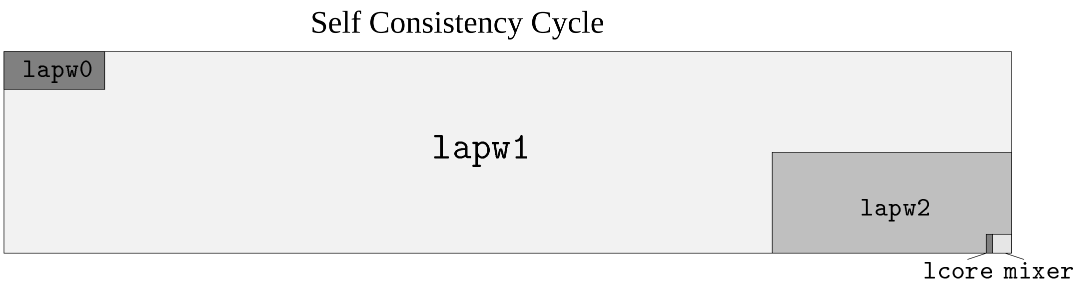

Figure 2.2: Execution times for the WIEN 97 modules of the self-consistency cycle (material: rutile, TiO_2). The area of each box is proportional to the execution time of the respective routine.

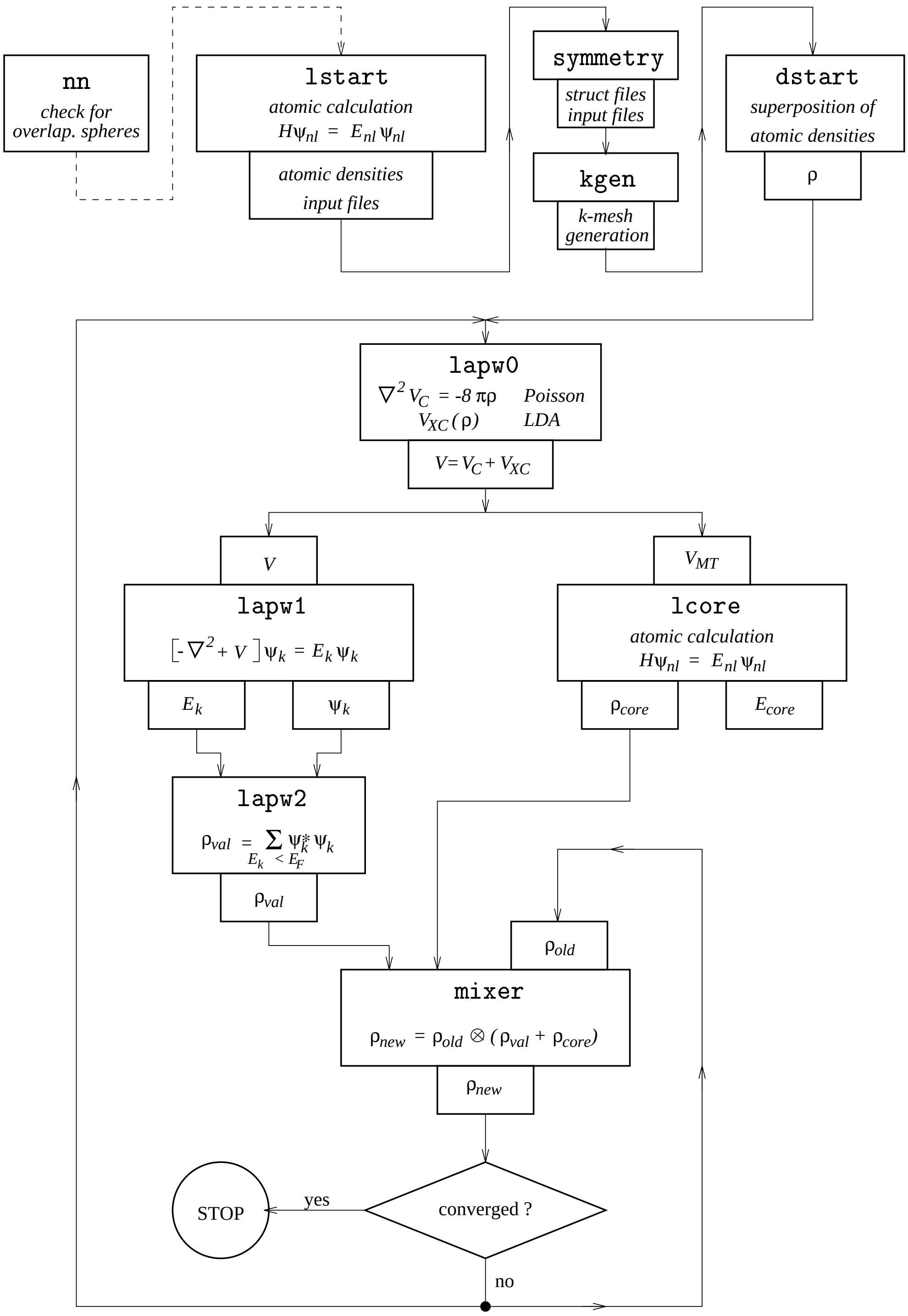

Figure 2.3: Program flow in WIEN 97.

lapw0	lapw1	lapw2	lcore	mixer
3.8 s	197.4 s	13.2 s	0.1 s	0.4 s
1.8 %	92 %	6 %	0.05 %	0.2 %

Table 2.1: Execution times for the programs of the self-consistency cycle (material: rutile, TiO_2).

nn	lstart	symmetry	kgen	dstart
0.02 s	0.6 s	0.03 s	0.02 s	3.2 s

Table 2.2: Execution times (in seconds) for the programs in the initialization phase (material: rutile (TiO_2)). Compared with the self consistency cycle these run times are insignificant.

`lapw1` is one of the programs of the WIEN 97 package used in a self-consistency cycle (see Figure 2.3). This cycle is repeated until convergence criteria are met. The cycle involves the modules `lapw0` (generates potential $E[\rho]$ from density ρ), `lapw1` (calculates valence bands, i. e., eigenvalues ϵ_i and eigenvectors c_i), `lapw2` (computes valence densities ρ from eigenvectors c_i), `lcore` (computes core states and densities), and `mixer` (mixes input ρ_{in} and output densities ρ_{out} to improve the convergence behavior).

An interactive initialization phase is needed before the self-consistency cycle is started. It consists of the routines `nn` (lists the nearest neighbors), `lstart` (generates free atomic densities and determines how the different orbitals are treated in the band structure calculations), `symmetry` (generates group symmetry operations from an input file), `kgen` (generates a k-mesh), and `dstart` (generates a starting density for the self-consistency cycle).

When the self-consistency cycle is finished other programs are needed, e. g., for generating graphical output. `tetra` calculates total and partial density of states. `spaghetti` generates an energy band structure plot (as a PostScript file). `lapw3` calculates X-ray structure factors from the charge density by Fourier transformation. `lapw5` generates the charge density (or the potential) in a specified plane of the crystal. `xspec` calculates near edge structure of X-ray absorption or emission spectra. `mini` performs movements of the atomic positions according to calculated forces. Using this program geometry minimizations can be performed in a big loop that contains the self-consistency cycle. `averx` calculates the average value of the operator X, which might be the number of electrons in the atomic sphere, or an orbital moment, or an orbital moment operator, or a hyperfine coupling operator. `optic` calculates optical properties. `joint` computes the inter-band or the intra-band contributions to the imaginary part of the dielectric tensor, alternatively, the density of states or the joint density of states can be derived. The output of `optic` is used by this program. `kram` carries out a Kramers-Kronig transformation using the imaginary part of the dielectric tensor derived in `joint`.

2.4.1 LAPW1

`lapw1` is the most time consuming module of WIEN 97 (see Table 2.1). In some exceptional cases `lapw2` may run nearly as long as `lapw1`, but all the other programs' run times remain insignificant. This is the reason why this report concentrates on `lapw1`.

The tasks performed by `lapw1` are (i) to set up the matrices H and S, and (ii) to solve the generalized symmetric eigenvalue problem. The first step requires $O(n^2)$ floating-point operations, whereas the second step requires $O(n^3)$ floating-point operations. Therefore, for larger problems the second step is the most time consuming part of `lapw1` and of the whole self-consistency cycle.

2.4.2 Eigenproblem Computation in WIEN 97

The main objective of our work was to improve performance (and efficiency) of the WIEN 97 package and at the same time to retain its accuracy. Our current work focuses on improving the efficiency of numerically solving equation (2.12).

The equation

$$\mathbf{HC} = \mathbf{SCE} \tag{2.18}$$

has to be solved with a large (for example, 4000×4000), dense, symmetric/Hermitian matrix $\mathbf{H}$, a symmetric/Hermitian positive definite matrix $\mathbf{S}$, the diagonal matrix of (unknown) eigenvalues E, and the matrix of (unknown) eigenvectors $\mathbf{C}$. Only the *lower* part (about 10 %) of the eigenpairs (eigenvalues together with the corresponding eigenvectors) is required.

The standard serial algorithms for computing the eigendecomposition of a symmetric matrix have the same two-phase structure: Apply orthogonal transformations to reduce the matrix to a *condensed* form and then apply an iterative algorithm to the condensed matrix to compute its eigendecomposition. Other approaches to the symmetric eigenproblem apply to *dense* matrices instead of *condensed* matrices. The best known is Jacobi's method .

For solving the Schrödinger equation, WIEN 97 switches between a condensed and a dense matrix eigensolver. Compared to the dense matrix eigensolver, the condensed matrix eigensolver involves much more floating-point operations. The condensed matrix eigensolver proceeds as follows:

1. **Reduction** of the generalized eigenvalue problem (2.18) **to a standard eigenvalue problem**

   $$\mathbf{H}'\mathbf{C}' = \mathbf{C}'\mathbf{E}. \tag{2.19}$$

   This involves a Cholesky factorization $\mathbf{S} = \mathbf{U}^\top \mathbf{U}$ (requiring roughly $n^3/3$ operations; see Golub, Van Loan [8]), and a two-sided matrix update $\mathbf{H}' = (\mathbf{U}^{-1})^\top \mathbf{H}\mathbf{U}^{-1}$ (requiring roughly n^3 operations; see Golub, Van Loan [8]).

   We have substantially improved this step. In contrast to the previous version of the code, WIEN 95, it is fully blocked, i. e., a Level 3 algorithm, now. Speedups up to 500 % have been achieved.

2. **Tridiagonalization** of $\mathbf{H}'$ (requiring roughly $4n^3/3$ operations; see Golub, Van Loan [8]).

   This part also has been improved compared to the previous version of the code, WIEN 95. Speedups of up to 100 % have been achieved.

3. **Calculation of the required eigenvalues** in the tridiagonal form (using iterative methods requiring $O(n^2)$ operations).

4. **Calculation of the corresponding eigenvectors** in the tridiagonal form (using iterative methods requiring $O(n^2k)$ operations).

5. Performing all necessary **back-transformations** (requiring roughly $5n^2k/2$ operations). The eigenvalues remain unchanged in the previous steps, but the eigenvectors of the tridiagonal matrix have to be transformed back to the eigenvectors of the original matrix **H**.

The main design goal was to improve the spatial locality of memory references in the condensed matrix eigensolver. On modern computer architectures improved data locality results in an efficiency enhancement, i. e., in reduced execution times.

Several modifications to the original algorithms have been applied. EISPACK routines have been replaced by LAPACK routines. Then the LAPACK routines were modified in a way that combines the memory efficiency of the packed storage scheme and the runtime efficiency of Level 3 BLAS routines. A new storage scheme—block packed storage—has been introduced to guarantee memory efficiency. An additional level of blocking—hierarchical blocking—was introduced to allow use of Level 3 BLAS.

2.4.3 Further Improvement

Tridiagonalization

The tridiagonalization process in Step 2 does not utilize Level 3 BLAS to a satisfactory degree. Thus, attempts will be made to introduce more Level 3 operations (see Gansterer et al. [6, 7] and Kvasnicka et al. [12]).

Improved Eigensolvers

In the future, the whole problem will be considered from at a higher level: *different* methods for solving the linear eigenproblem will be investigated. Finally—as a long-term plan—other high-level modifications (e. g., for convergence acceleration) could be made. These issues, however, are *not* part of the current report.

In the current stage it was the goal to achieve performance improvements based on *existing* software (e. g., using LAPACK routines with only minor code modifications wherever possible), and not to do a lot of coding from scratch. Future work will necessarily attack the problem at a higher level and will have to look into substantially different algorithmic approaches. This will most likely open up a variety of opportunities for further improvement. But at the same time it will require significantly higher coding and implementation effort.

As already mentioned, there is a second, dense matrix eigensolver. At the moment, it is approximately five times faster than the condensed matrix eigensolver, which is based on tridiagonalization. Trading convergence for performance, WIEN 97 switches back and forth between the two methods: Whenever the progress of the dense matrix eigensolver slows down or the iterates proceed in a wrong direction, it switches back to the slower, but much more accurate condensed matrix eigensolver.

The method of the dense matrix eigensolver, which is implemented in WIEN 97, is described in Singh [13]. The method is based on the observation that the exact eigenvectors in the initial iterations (far from self-consistency) are of little interest and approximate eigenvectors would serve as well, while in the final iterations the eigenvectors from the previous iteration are a very close approximation to the true eigenvectors. Thus the eigenvectors can be refined as self-consistency is approached, yielding a substantial reduction in computing time. The advantage of this method over the conventional approach lies in the fact that it solves the diagonalization problem simultaneously with the self-consistency problem.

Provided with accurate starting values computed by the condensed matrix eigensolver, several steps of the dense matrix eigensolver can be performed. The accuracy of the dense matrix eigensolver could be increased by computing a larger subspace, i. e., a larger number of eigenpairs. This, however, would require more time for the whole self-consistency cycle than switching between the two eigensolvers.

Parallel Code Version

Hierarchical blocking has been used to optimize the algorithms for modern cache based sequential computer architecture. A parallel code version, which is under development, will substantially benefit from this blocking, as the ratio of computation to communication can be held fairly high.

Chapter 3

Run Time of LAPW1

In a first series of run time experiments, symmetric generalized eigensolutions have been investigated on a Silicon Graphics (SGI) Power Challenge XL.

3.1 Original Program Version

First experiments with `lapw1` showed that its run time was spent primarily in modified EISPACK routines, which had been extensively fine tuned for several computer architectures. These routines are `redc5` (which consists of `cholesky`, `a_prime`, and `p_compos`), `tred5` (which calls `dsupmv` and `dsupr2`), `solve`, `trbak5` and `rebk5`. All these routines are called from `seclr4.frc`.

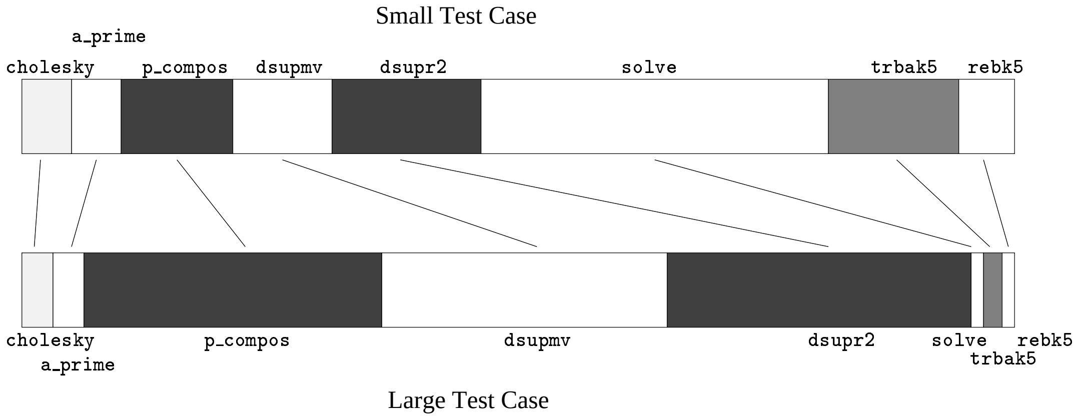

Figure 3.1: Run time proportions for the original `lapw1` routines for a small test case (matrix size $n = 283$) and a larger test case (matrix size $n = 1251$).

n	cholesky	a_prime	p_compos	redc5
283	0.03 s	0.03 s	0.07 s	0.14 s
1251	2.73 s	2.78 s	26.70 s	32.21 s

n	dsupmv	dsupr2	tred5
283	0.07 s	0.10 s	0.18 s
1251	25.55 s	29.23 s	55.17 s

n	solve	trbak5	rebk5
283	0.24 s	0.09 s	0.04 s
1251	0.99 s	1.69 s	0.82 s

Table 3.1: Run time of the original routines for a small test case (matrix size $n = 283$) and a larger test case (matrix size $n = 1251$) in seconds.

Differences of the run time proportions with respect to problem size (as indicated in Figure 3.1 and Table 3.1) are due to more frequent cache misses when solving larger problems. The data of the smaller test case in Table 3.1 can be held in cache, whereas the data the of larger test case do not fit into the cache, which leads to excess time for load and store operations.

Some routines (`cholesky`, `a_prime`) have memory access patterns which perform well on cache based architectures, others (`p_compos`, `dsupmv`, `dsupr2`) access memory in a more global way, which dramatically increases the number of cache misses. The last three routines require less than an $O(n^3)$ effort, so their run times can be neglected when solving larger problems.

3.1.1 REDC5

The subroutine `redc5` transforms the original problem $\mathbf{HC} = E\mathbf{SC}$ to the equivalent problem $\mathbf{H'C'} = E\mathbf{C'}$. It performs first a Cholesky factorization, then two matrix multiplications (`a_prime` and `p_compos`). Of these three subprograms only `p_compos` does not make appropriate use of the cache memory.

3.1.2 TRED5

The subroutine `tred5` performs a tridiagonalization of the matrix H by iteratively calling the subroutines `dsupmv` (which performs a matrix-vector product) and `dsupr2` (which performs a symmetric rank-two update using a matrix and two vectors). Both subroutines do not utilize the cache memory appropriately.

3.1.3 SOLVE, TRBAK5, and REBK5

The three routines `solve`, `trbak5`, and `rebk5` are not performing badly, so they were not considered for improvement.

3.2 Program Version with LAPACK Calls

In a first experiment calls to several LAPACK routines were introduced. The real part of `seclr4.frc` was completely rewritten to utilize existing LAPACK routines.

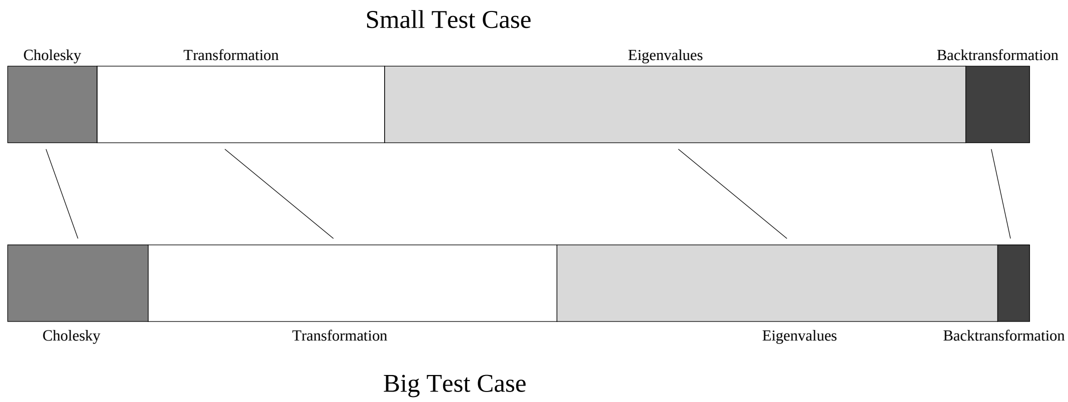

Figure 3.2: Run time proportions for the LAPACK version for a small test case (matrix size $n = 283$) and a larger test case (matrix size $n = 1251$).

n	Cholesky	Transform	Eigenvalues	Backtransform
283	0.07 s	0.23 s	0.46 s	0.05 s
1251	21.62 s	58.47 s	63.66 s	4.05 s

Table 3.2: Run times of the LAPACK version for two test cases.

The LAPACK routines called are `LAPACK/dpptrf` (Cholesky), `LAPACK/dspgst` (Transform), which performs transformation of a generalized to a standard eigenproblem, `LAPACK/dspevx` (Eigenvalues) to solve the standard eigenproblem, and a loop calling `BLAS-2/dtpsv` (Backtransform), which transforms the solution of the standard eigenproblem to the solution of the generalized eigenproblem. The

LAPACK routines use the packed storage scheme and can therefore only use Level 2 BLAS calls, which degrades performance (Table 3.2).

LAPACK supports the following four storage schemes for matrices:

- conventional storage in a two-dimensional array;
- packed storage for symmetric, Hermitian or triangular matrices;
- band storage for band matrices;
- the use of two one-dimensional arrays to store bidiagonal or symmetric tridiagonal matrices.

`lapw1` uses the packed storage scheme because of conflicts between problem size and memory size. In upper packed storage scheme (UPLO='U') a_{ij} is stored in $\texttt{AP}(i + j(j-1)/2)$ for $i \leq j$. In lower packed storage scheme (UPLO='L') a_{ij} is stored in $\texttt{AP}(i + (2n-j)(j-1)/2)$ for $j \leq i$. `lapw1` in the current version uses the upper packed storage scheme.

3.2.1 DPPTRF

`LAPACK/dpptrf` (which requires 14 % of the run time of the big test case) either computes the Cholesky factorization $\mathbf{S} = \mathbf{L}\mathbf{L}^\top$ with a lower triangular matrix $\mathbf{L}$ or the Cholesky factorization $\mathbf{S} = \mathbf{U}^\top\mathbf{U}$ with an upper triangular matrix $\mathbf{U}$ using packed format and therefore only calling Level 2 BLAS. A faster version of this subroutine is `LAPACK/dpotrf`, which uses blocking and is up to a factor of ten faster (on the SGI Power Challenge XL with large matrices). However, this routine uses twice as much memory using full (conventional) storage scheme.

3.2.2 DSPGST

The subroutine `LAPACK/dspgst` (40 % time) transforms the problem

$$\mathbf{HC} = \mathbf{SCE}$$

to the equivalent problem

$$\mathbf{H}'\mathbf{C}' = \mathbf{C}'\mathbf{E}.$$

$\mathbf{H}$ is the Hamiltonian matrix, $\mathbf{C}$ the matrix of eigenvectors, $\mathbf{S}$ the overlap matrix, $\mathbf{E}$ is a diagonal matrix with eigenvalues at the diagonal, $\mathbf{H}' = \mathbf{L}^{-1}\mathbf{H}(\mathbf{L}^{-1})^\top$ or $\mathbf{H}' = (\mathbf{U}^{-1})^\top\mathbf{H}\mathbf{U}^{-1}$, $\mathbf{C}'$ the solution of the transformed problem.

The BLAS routines used to perform this transformation are `BLAS-1/dscal`, `BLAS-1/daxpy`, `BLAS-2/dspr2` (packed rank-2 update), and `BLAS-2/dtpsv` (packed solver). These routines are called in a loop over N. (Other variants of this routine use other BLAS calls.)

3.2.3 DSPEVX

The subroutine `LAPACK/dspevx` (43 % time) computes the eigenvalues **E** (within a given energy range) and the corresponding eigenvectors $\mathbf{C}'$.

The called routines are `LAPACK/dlansp` (returns the element of largest absolute value), `LAPACK/dsptrd` (reduces a real symmetric matrix A stored in packed form to symmetric tridiagonal form T by an orthogonal similarity transformation $T = Q^\top AQ$). `LAPACK/dstebz` is performed to compute the eigenvalues in the interval $(EL1, EU1]$ of the symmetric tridiagonal matrix returned by `LAPACK/dsptrd` and scaled by `LAPACK/dlansp`. Since the eigenvectors are required, the routines `LAPACK/dstein` and `LAPACK/dopmtr` are called afterwards. At last the matrix is scaled back (by `BLAS-1/dscal`) and the eigenvalues are ordered along with the eigenvectors.

DSTEBZ

The routine `LAPACK/dstebz` computes the eigenvalues of a symmetric tridiagonal matrix T.

DSTEIN

The routine `LAPACK/dstein` computes the eigenvectors of a real symmetric tridiagonal matrix T corresponding to specified eigenvalues, using inverse iteration.

DOPMTR

The routine `LAPACK/dopmtr` overwrites the matrix Z with $Q \times Z$ (see Section 5.3.3).

3.2.4 DTPSV

The subroutine `BLAS-2/dtpsv` (3 % time) transforms the vector $\mathbf{C}'$ back to $\mathbf{C}$, which is the solution of the original problem. This routine is called in a loop to transform the whole matrix of vectors $\mathbf{C}'$.

3.3 Comparison of the Original Version and the LAPACK Version

For larger matrices (2457 or 1781) the results are equivalent to the larger test case (matrix size 1251), which is shown in Figure 3.3.

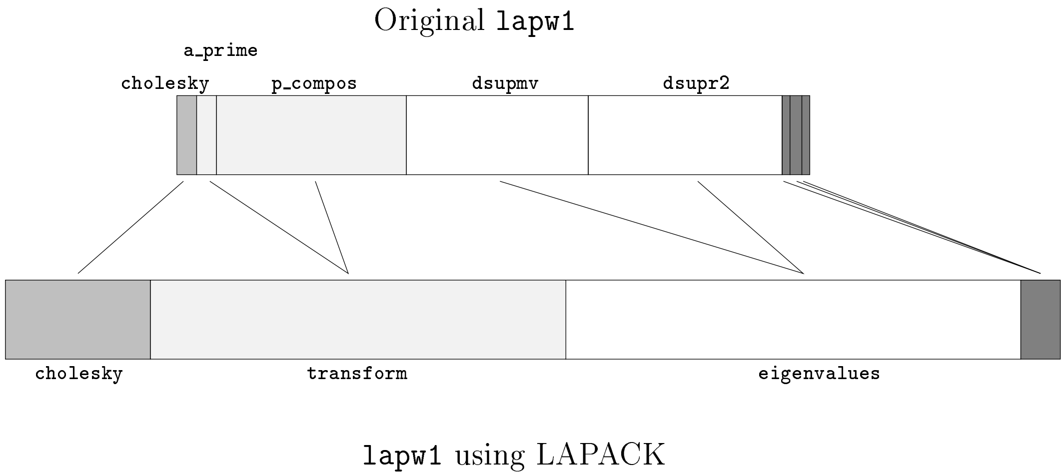

Figure 3.3: Comparison of the original version of `lapw1` and the LAPACK version of `lapw1` for a test case with a matrix size $n = 1251$.

The routines which have to be compared with respect to their run times are `cholesky` (2.8 s versus 21.6 s), `a_prime` and `p_compos`, which are the counterparts of the routine `transform` (29.5 s versus 58.5 s), `tred5`, which is the counterpart of the routine `eigenvalues` (55.2 s versus 63.7 s). The other routines may be neglected in this run time comparison. The LAPACK version is always slower than the (highly tuned) original version. Some routines of the original program version show significantly lower performance than others. This indicates the lack of a good cache strategy. These routines achieve only an efficiency of 13 % (50 MFlop/s on a processor with a peak performance of 390 MFlop/s), which indicates possibilities for improvement using better tuned algorithms.

Chapter 4

Methods for Efficiency Improvement

4.1 Storage Schemes

Lapack provides different storage schemes for dense matrices. Symmetric matrices can either be stored using conventional storage or packed storage. As both storage schemes have some drawbacks, a more suitable storage scheme, the block packed storage, is proposed. Figure 4.1 illustrates the different storage schemes, and Figure 4.3 compares their storage efficiency.

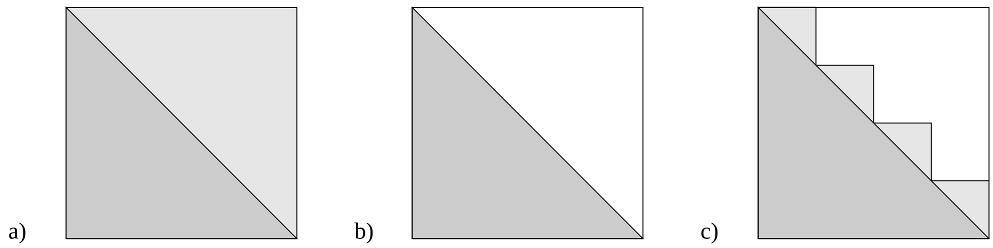

Figure 4.1: Storage schemes; a) conventional (unpacked) storage, b) packed storage, c) block packed storage format. The darker shaded blocks indicate matrix elements whose values are used by the algorithms; the lighter shaded parts indicate matrix elements for which redundant storage is required.

4.1.1 Conventional Storage

Conventional storage is the default scheme for storing matrices: a matrix A is stored column wise in a two-dimensional array `A`, with matrix element a_{ij} stored in array element `A(i,j)`. LAPACK routines that handle symmetric or Hermitian matrices allow for either the upper or lower triangular part of matrix (as specified by the parameter `UPLO`) to be stored in the corresponding elements of the array. The remaining elements of the array need not be set (see Fig. 4.1a).

4.1.2 Packed Storage

Symmetric, Hermitian or triangular matrices can be stored more compactly if the relevant triangular part is packed by columns in a one-dimensional array `AP`. If the upper triangular part of a matrix A is packed column wise, the matrix element a_{ij} $(i \leq j)$ is stored in

$$\mathtt{AP}(i + j(j-1)/2)$$

If the lower triangular part of a matrix A is packed column wise, the matrix element a_{ij} $(j \leq i)$ is stored in

$$\mathtt{AP}(i + (2n-j)(j-1)/2).$$

Conventional storage requirements can be halved (see Fig. 4.1b) in this way. A major drawback is that Level 3 BLAS for matrices in packed storage format do not exist. Using only Level 2 BLAS, however, can dramatically decrease performance on some machines.

4.1.3 Block Packed Storage

Block packed storage tries to combine the advantages of the two previous storage schemes, thus making it possible to use Level 3 BLAS and to minimize storage requirements at the same time. The main idea behind this storage scheme is to extend the packed storage format by adding some additional storage locations so that Level 3 BLAS can operate on submatrices of suitable size (see Fig. 4.1c).

The elements of the block diagonal and the lower block triangular part of a matrix A are packed column wise into a one-dimensional array `AP`. The diagonal blocks are all of order h_b (the last block may possibly be smaller). The values of the superdiagonal elements (within the diagonal blocks) are not stored, but storage

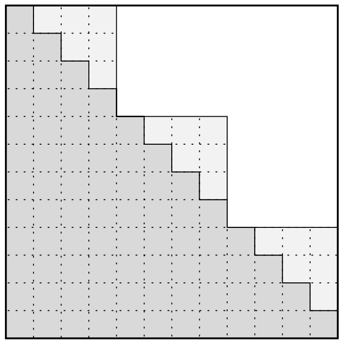

Figure 4.2: A matrix of order 12 in block packed storage format ($h_b = 4$). Each square represents a matrix element. The darker shaded parts indicate matrix elements whose values are used by the algorithm. The lighter shaded parts indicate matrix elements for which only storage locations are reserved. All shaded elements are stored column wise in a one-dimensional array.

locations are reserved for them. If the lower triangular part of a matrix A is packed column wise, the matrix element a_{ij} ($j \leq i$) is stored in

$$\texttt{AP}(i + (2n - j)(j - 1)/2 + (h_b(h_b - 1)/2)((j - 1)/h_b) + h(h + 1)/2)),$$

where

$$h = \bmod\ (j - 1, h_b).$$

Fig. 4.2 shows a symmetric matrix of order 12 in block packed storage format with $h_b = 4$. Note, that both the conventional and the packed storage format are special cases of the block packed storage format. Specifying $h_b = 1$ is equivalent to using packed storage; specifying $h_b = n$, where n is the order of the matrix is equivalent to using conventional storage.

Packing the upper triangular part of real or complex symmetric matrices by columns is equivalent to packing the lower triangular part by rows; packing the lower triangular part by columns is equivalent to packing the upper triangular part by rows.

4.2 Hierarchically Blocked Algorithms

A blocked algorithm merely divides a task into subtasks which work on smaller matrices. The operations on the submatrices may themselves be described by blocked algorithms, which leads to the notion of *hierarchical blocking*.

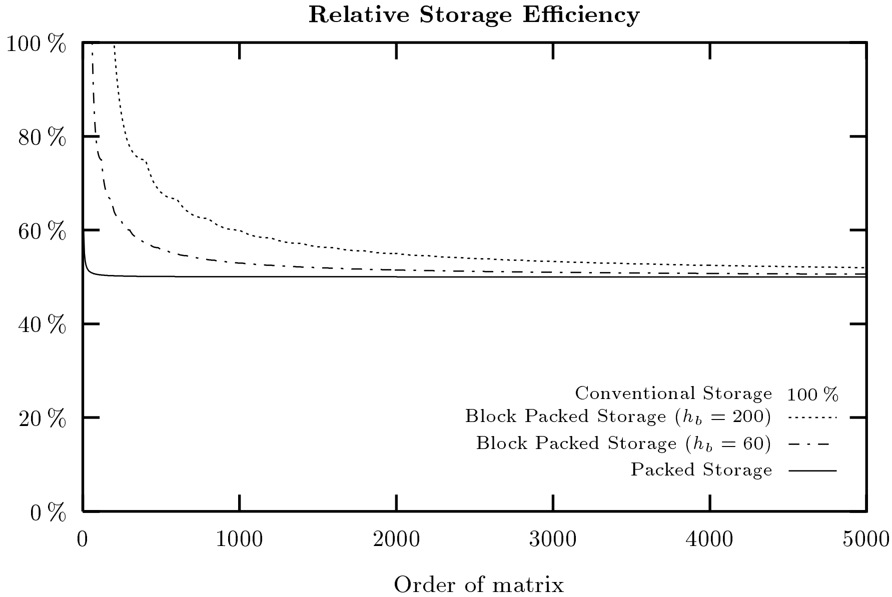

Figure 4.3: Storage requirement of several storage formats. The ratio of required storage locations (words) to the total number of matrix elements (100 %) is plotted.

If several levels of the memory hierarchy are to be exploited, then hierarchically blocked algorithms can be used; each blocking level exploits a corresponding level of the memory hierarchy.

The program package LAPACK, used for the solution of systems of linear equations and eigenvalue problems, comprises the blocked as well as the unblocked versions of many algorithms. However, blocked LAPACK algorithms employ just *one* blocking level. Thus, page faults may lead to substantial losses in performance even for the optimal block size, if the respective matrices do not fit into main memory. A second blocking level may significantly improve the locality of reference for large matrices which cannot be held in main memory and, hence, significantly increase floating-point performance. Some examples of hierarchically blocked algorithms can be found in Ueberhuber [15].

Furthermore, hierarchical blocking has been demonstrated to increase efficiency by about 10 to 30 % even if the respective matrices fit into main memory. This improvement is due to the fact that hierarchical blocking improves overall data locality. The following pseudo-code fragment shows why:

The innermost loop of blocked algorithms normally looks like

```
DO i = 1, n
   CALL operation_one(large_matrix_A)
   CALL operation_two(large_matrix_A)
   CALL operation_three(large_matrix_A)
END DO
```

The single operations may have a high degree of data locality, but for every loop cycle each matrix element has to be transferred at least three times from main memory into cache memory. Hierarchically blocked algorithms avoid unneccesary data transfers by splitting operations acting on large matrices into a series of operations on small submatrices. The corresponding hierarchically blocked pseudo-code fragment may look like

```
DO i = 1, n, number_of_block_columns
   DO j = i, min(n,i+number_of_block_columns-1)
       CALL operation_one(block_column_A(j))
       CALL operation_two(block_column_A(j))
       CALL operation_three(block_column_A(j))
   END DO
END DO
```

If the block size is chosen suitably so that all required data can be kept in cache memory, the overall data locality can be increased significantly.

4.3 Performance Quantification

The assessment of scientific software requires the use of numerical values, which may be determined analytically or empirically, for the quantitative description of performance. Which parameters are used and how they are to be interpreted depends on *what* is to be assessed.

The user of a computer system who waits for the solution of a particular problem is mainly interested in the time it takes for the problem to be solved. This time depends on workload and performance:

$$\begin{aligned}\text{time} &= \frac{\text{workload}}{\text{performance}_{\text{effective}}} \\ &= \frac{\text{workload}}{\text{performance}_{\text{maximum}} \times \text{efficiency}} \,.\end{aligned}$$

The computation time is therefore influenced by the following three quantities:

1. The amount of work (*workload*) which has to be done. This depends on the nature and complexity of the problem as well as on the properties of the algorithm used to solve it. For a given problem complexity, the workload is a characteristic of the algorithm. The workload (and hence the required time) may be reduced by improving the algorithm.

2. *Peak performance* characterizes the computer hardware independently of particular application programs. The procurement of new hardware with high peak performance usually results in reduced time requirements for the solution of a problem.

3. *Efficiency* is the percentage of peak performance achieved for a given computing task. It tells the user what share of the potential peak performance is actually exploited and thus measures the quality of the implementation of an algorithm. Efficiency may be increased by optimizing the program.

Chapter 5

Improved Algorithms

To improve the LAPACK version of the generalized symmetric eigenvalue problem (GSEP; Chapter 3) hierarchically blocked algorithms (Chapter 4.2) have been developed. These algorithms rely on the new block packed storage scheme (Chapter 4.1). Thus they combine the benefits of the LAPACK routines using full storage scheme (using fast Level 3 BLAS) and the ones using packed storage scheme (requiring only half the storage size). The floating-point performance of some of these routines is even better than that of the fast LAPACK versions. This speed-up is a result of the additional blocking level which has been introduced.

Figure 5.1 shows the calling structure of the most time consuming routines of the program `lapw1`. The program `lapw1` calls (among others) the routine `seclr4` to solve general eigenproblems (compute the eigenvalues and eigenvectors). This routine first calls the (new) Cholesky factorization routine `dphtrf` (Chapter 5.1), then the (new) transformation routine `dshgst` (Chapter 5.2), and finally the eigenproblem routine `dshevx` (Chapter 5.3). The routine `dshevx` calls other subprograms for the tridiagonalization of the matrix, computing the eigenvalues and eigenvectors of the tridiagonal matrix and transforming the eigenvectors back to the solution of the original symmetric eigenvalue problem.

5.1 Blocked Cholesky Factorization

LU factorization and Cholesky factorization are the simplest blocked algorithms to derive. Derivation of blocked algorithms from the corresponding vectorizable algorithms is straightforward. No extra floating-point operations or extra working storage is required. For many computations in linear algebra, there are several vectorizable variants (often referred to as i-, j, and k-variants; see Dongarra [5]). Similiar variants exist for the corresponding blocked algorithms.

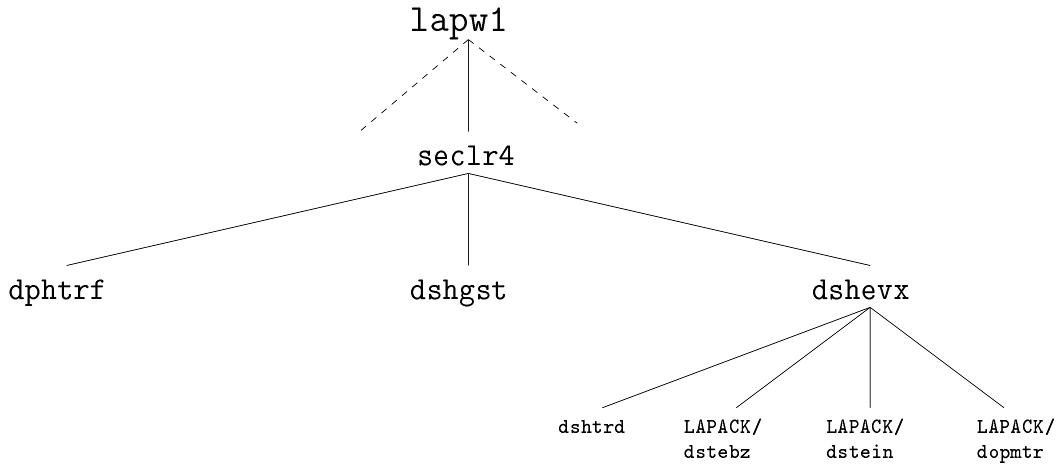

Figure 5.1: Calling structure of the most time consuming routines of `lapw1`.

The variant of Cholesky factorization actually used in LAPACK is the j-variant, which does rely on matrix-matrix multiplication (done by the routine `BLAS-3/dgemm`). The i-variant is based on solving triangular systems (done by the routine `BLAS-3/dtrsm`), the k-variant is based on symmetric matrix-matrix operations (performed by `BLAS-3/dsyrk`). On most computer systems matrix-matrix multiplication is implemented more efficiently than the other operations are.

Cholesky factorization decomposes an $n \times n$, symmetric, positive definite matrix A into the product of a lower matrix L and its transpose, i. e., $A = LL^\top$ (or $A = U^\top U$, where U is upper triangular).

In the rest of this paper, only the case in which the lower part of A is stored will be discussed; algorithms using the upper part of A can be derived similarly.

A partitioned form (k-variant) of the Cholesky factorization can be derived inductively. At the k-th step of the computation, it is assumed that the $n \times n$ matrices $A^{(k)}$, $L^{(k)}$, and ${L^{(k)}}^\top$ can be partitioned as follows

$$\begin{aligned}\begin{pmatrix} A_{11} & A_{21}^\top \\ A_{21} & A_{22} \end{pmatrix} &= \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} L_{11}^\top & L_{21}^\top \\ 0 & L_{22}^\top \end{pmatrix} \\ &= \begin{pmatrix} L_{11}L_{11}^\top & L_{11}L_{21}^\top \\ L_{21}L_{11}^\top & L_{21}L_{21}^\top + L_{22}L_{22}^\top \end{pmatrix}.\end{aligned}$$

First A_{11} has to be factorized. As Fortran 77 does not allow recursion, a separate routine `dpotf2` must be called (using Level 2 BLAS rather than Level 3 routines). Hence, if L_{11}, the lower triangular factor of A_{11}, is known, L_{21} can be computed from the relation

$$L_{21} = A_{21}(L_{11}^{\top})^{-1}$$

by calling `BLAS-3/dtrsm`; then A_{22} must be updated as

$$\tilde{A}_{22} = A_{22} - L_{21}L_{21}^{\top} = L_{22}L_{22}^{\top}.$$

Now the factorization can be continued by recursively applying the steps outlined above to the submatrix $\tilde{A}_{22}$.

A snapshot of the k-variant of the Cholesky factorization algorithm in Fig. 5.2 illustrates how the lower factor L is computed.

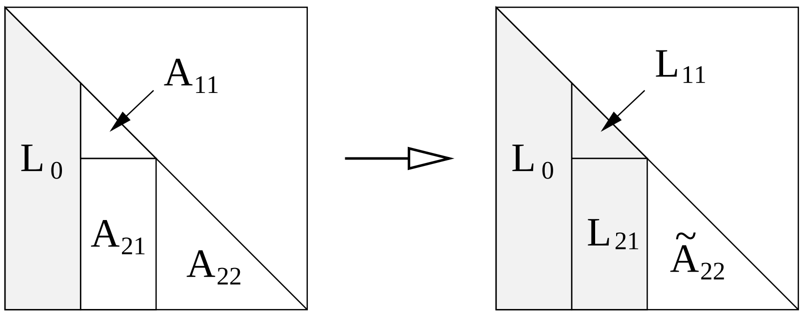

Figure 5.2: A snapshot of blocked Cholesky factorization (k-variant). The shaded parts of L have, already, been computed.

In a similar way the i-variant and the j-variant can be derived. Performance rates of the three variants of the Cholesky factorization are given in Fig. 5.3 (using the lower triangle of A) and Fig. 5.4 (using the upper triangle of A).

In the first case, the k-variant is superior to the other ones, especially for large matrices. In the latter case, the i-variant is slightly faster than the other variants. In both cases, the j-variant, which is implemented in `LAPACK/dpotrf` is significantly slower. Similar results have been obtained on all computer systems tested. For that reason the k-variant has been taken as a basis for further improvements.

The large problem of Chapter 3 ($n = 1251$) shows a run time decrease for the Cholesky factorization from 21.6 s to 3.5 s (including 0.2 s needed for the conversion to block packed storage). This routine reduces the run time fraction of the Cholesky factorization from 14 % to a mere 2 % of the total execution time needed for `lapw1`. However, the total execution time for the LAPACK version is still 60 % higher than the time needed for the original version of `lapw1`. Figure 5.6 shows that the difference between upper and lower storage format as well as optimizing the i-, j-, and k-variant (which could sum up to about 0.3 s) is negligible as long

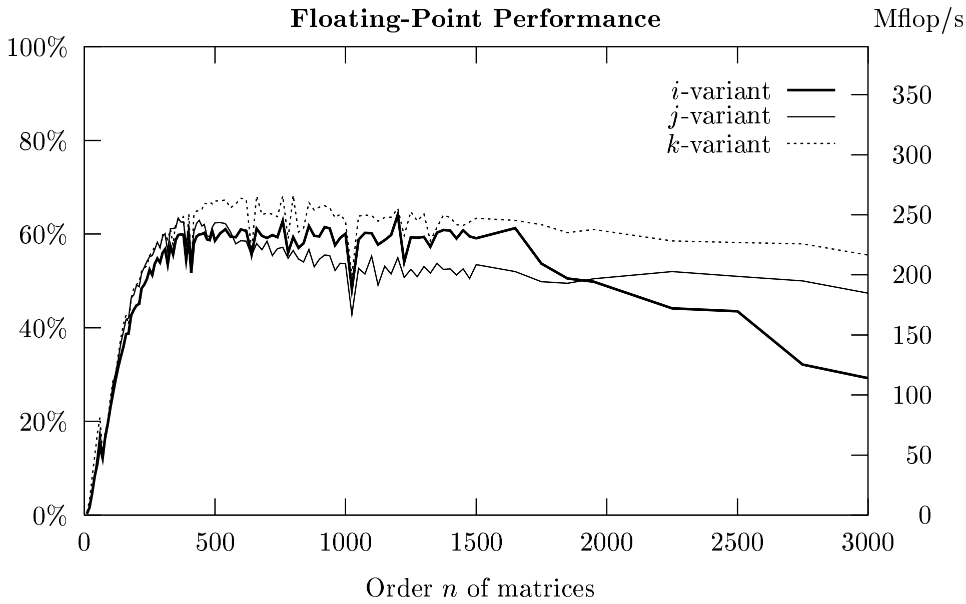

Figure 5.3: Floating-point performance of the three variants of the blocked Cholesky factorization (blocksize $h_b = 64$) when run on an SGI Power Challenge XL. The *lower triangle* of the array `A` was used to store the matrix A.

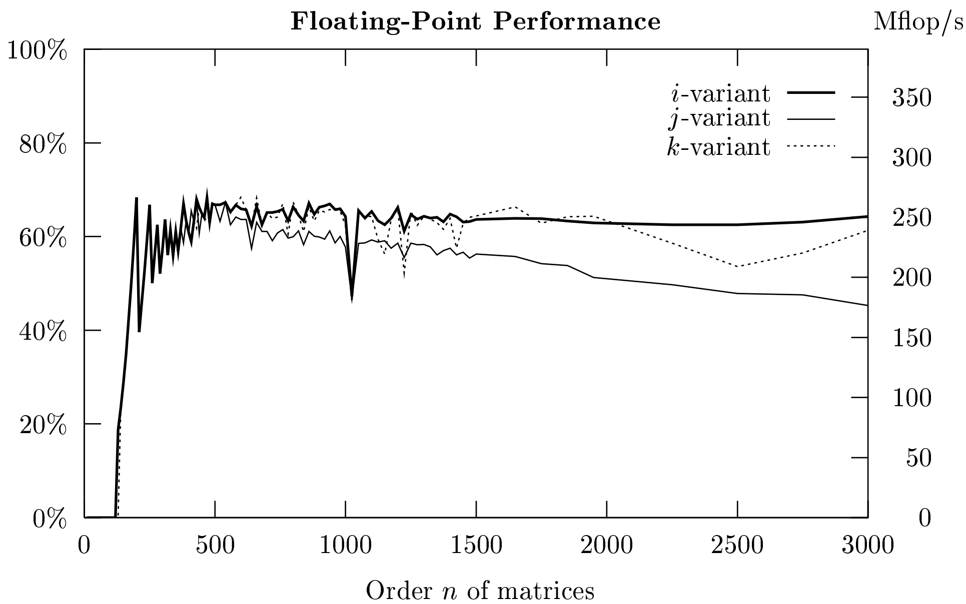

Figure 5.4: Floating-point performance of the three variants of the Cholesky factorization (blocksize $h_b = 64$) when run on an SGI Power Challenge XL. The *upper triangle* of the array `A` was used to store the matrix A.

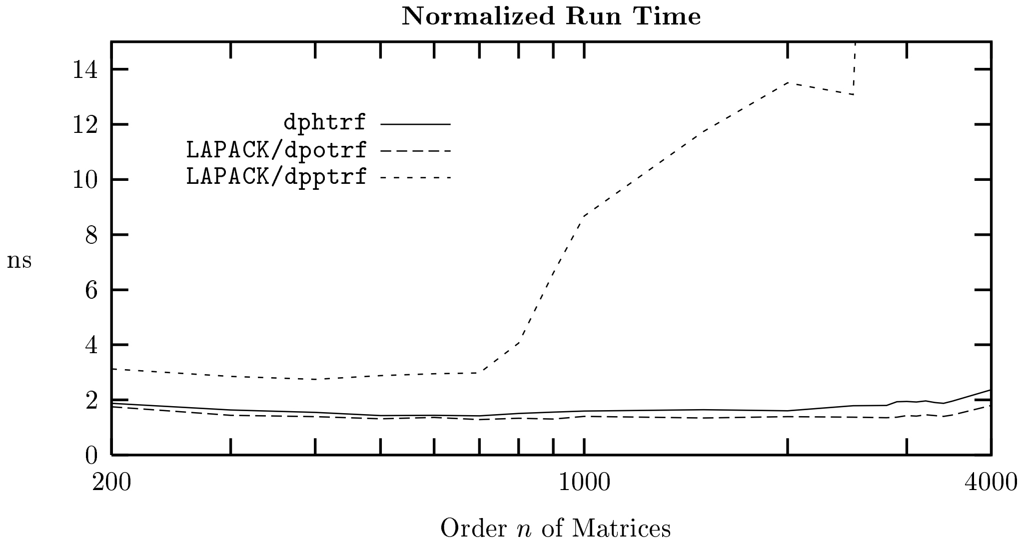

Figure 5.5: Normalized execution time (run time/n^3 in nanoseconds) for the Cholesky factorization using conventional storage (`LAPACK/dpotrf`), packed storage (`LAPACK/dpptrf`), and block packed storage (`dphtrf`) (using the blocking factor $h_b = 64$).

as the transformation to a standard eigenproblem and the eigenproblem solver are not performing better.

5.2 Reduction of a GSEP to a SSEP

A generalized symmetric eigenproblem (GSEP) can be reduced to a standard symmetric eigenvalue problem (SSEP) via factorizing B into $LL^\top$ or $U^\top U$ and applying the factor to the matrix A. Given the Cholesky factorization $B = LL^\top$ the GSEP $Ax = \lambda Bx$ can be written as

$$(L^{-1}A(L^{-1})^\top)(L^\top x) = \lambda(L^\top x).$$

Hence the eigenvalues of $Ax = \lambda Bx$ are identical with those of $Cy = \lambda y$, with the symmetric matrix $C = L^{-1}A(L^{-1})^\top$ and $y = L^\top x$. Analogously, with $B = U^\top U$ one gets

$$((U^{-1})^\top AU^{-1})(Ux) = \lambda(Ux).$$

The reduction to a SSEP is implemented in the routines `LAPACK/dsygst` (for conventional storage) and `LAPACK/dspgst` (for packed storage). Both matrices A and C are stored in the array `A`.

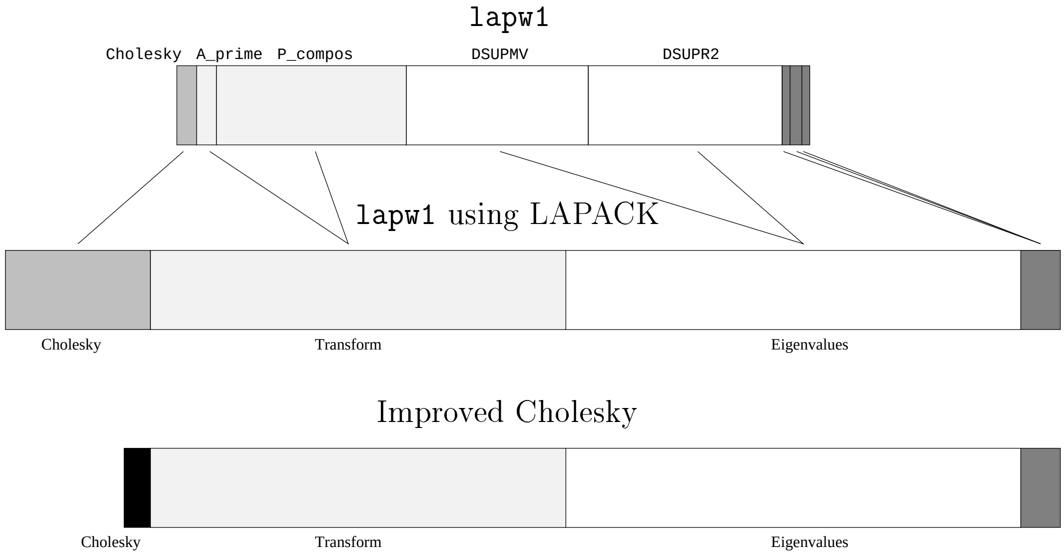

Figure 5.6: First improvement of `lapw1`.

A block row of A is computed in the following steps.

1. Blocked reduction of A_{11} using the unblocked routine `LAPACK/dsygs2`.
2. Solution of the triangular systems
$$B_{11}^{\top} X = A_{12}$$
using `BLAS-3/dtrsm`. A_{12} is overwritten by the solution X.
3. Symmetric rank-k update of A_{12} using `BLAS-3/dsymm`
$$A_{12} := A_{12} - 0.5 \cdot A_{11} B_{12}.$$
4. Symmetric rank-$2k$ update of A_{22} using `BLAS-3/dsyr2k`
$$A_{22} := A_{22} - A_{12}^{\top} B_{12} - B_{12}^{\top} A_{12}.$$
5. Symmetric rank k update of A_{12} using `BLAS-3/dsymm`
$$A_{12} := A_{12} - 0.5 \cdot A_{11} B_{12}.$$
6. Solution of the triangular systems
$$X B_{22} = A_{12}$$
using `BLAS-3/dtrsm`. A_{12} is overwritten by the solution X.

Figure 5.7 shows the flow of information in the routine `dshgst`. The k-loop runs over all the blocks of the matrix, the kk-loop runs from k to the rest of the blocks, and the kkk-loop runs from k to the rest of the blocks, too. In every line of the figure the grey fields are used to update the dark fields of other matrices. The matrices are drawn in the sequence as they are written in the subroutine call. Matrix A is the Hamilton matrix, matrix B is the secular matrix (in factorized form).

A snapshot of the execution of `LAPACK/dsygst` in Fig. 5.8 illustrates how a block row of A is computed.

Fig. 5.9 shows the floating-point performance for some routines which reduce a GSEP to a SSEP. The unblocked routine `LAPACK/dspgst` shows very poor performance rates. The blocked routine `LAPACK/dsygst` is much more efficient, but has doubled memory requirements. The hierarchically blocked routine `dshgst` uses the block packed storage. Thus `dshgst` is as memory efficient as `LAPACK/dspgst`. For larger matrices the performance rates of `dshgst` are superior to all other tested routines. In addition, the performance rates of the hierarchically blocked routine `dsygs4`, which uses full storage format, are given. The performance gains of hierarchical blocking over standard blocking are evident.

Figure 5.10 shows the run time behaviour of the newly developed routine `dshgst`. A speedup of about six over `LAPACK/dspgst` is achieved and the new routine performs even better than the unpacked LAPACK routine, which consumes twice as much memory.

Figure 5.11 depicts the improvement of `dshgst` in `lapw1` when compared with the previous version.

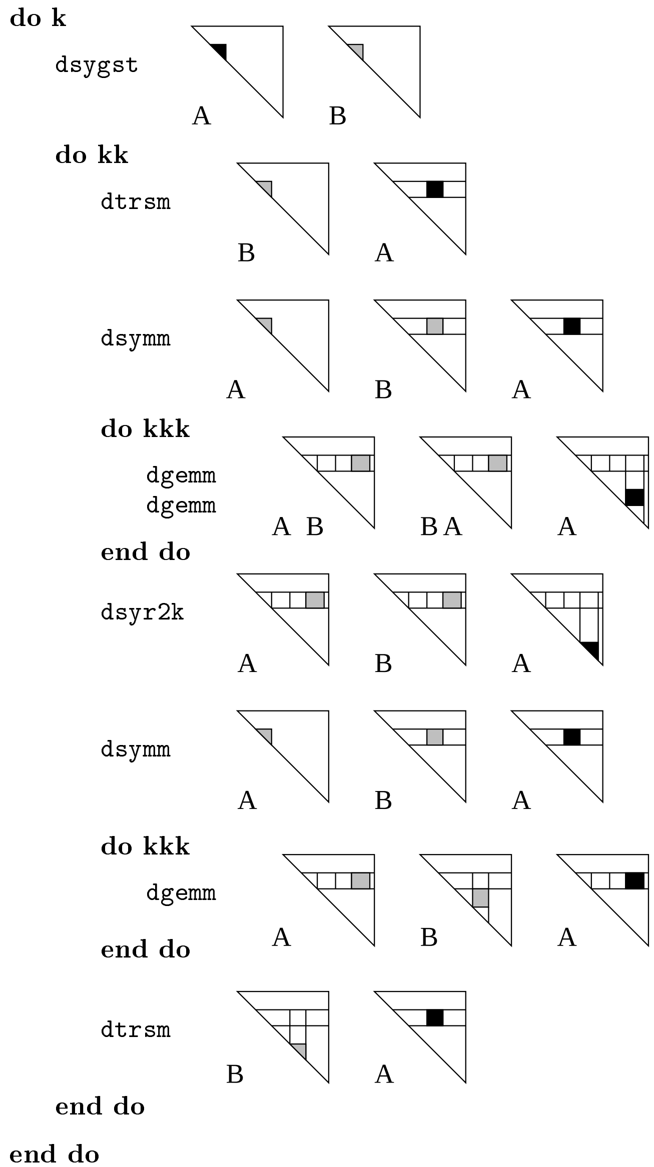

Figure 5.7: Matrix access in the routine `dshgst`. Grey sections of the triangular matrices A (the Hamiltonian) and B (the secular matrix) are used to update the black sections of matrix A.

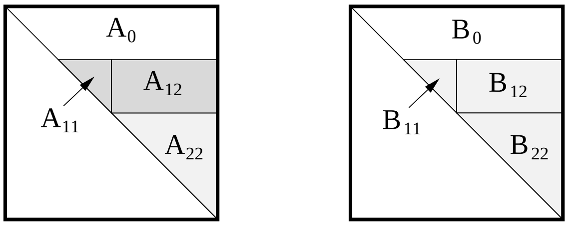

Figure 5.8: A snapshot of the execution of the blocked routine `LAPACK/dsygst` (which reduces a GSEP to a SSEP). The block column A_0 has already been computed. The darker shaded parts of A are computed at this stage. All other shaded blocks of A and B are referenced during the computation of the block row of A.

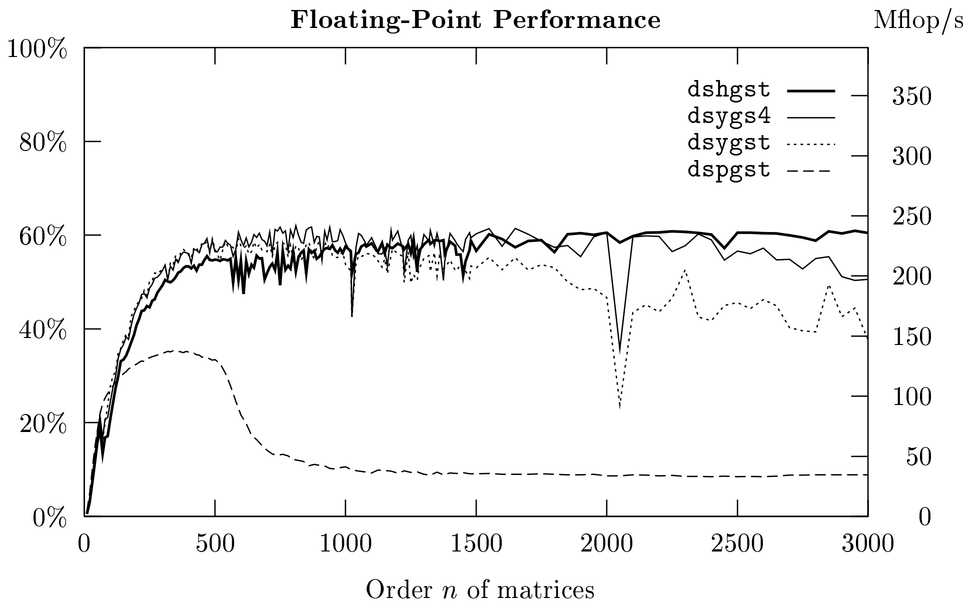

Figure 5.9: Performance results for the hierarchically blocked routines `dshgst` (using block packed storage format), `dsygs4` (using conventional storage format), the blocked routine `LAPACK/dsygst` (using conventional storage format) and the unblocked routine `LAPACK/dspgst` (using packed storage format) on an SGI Power Challenge XL.

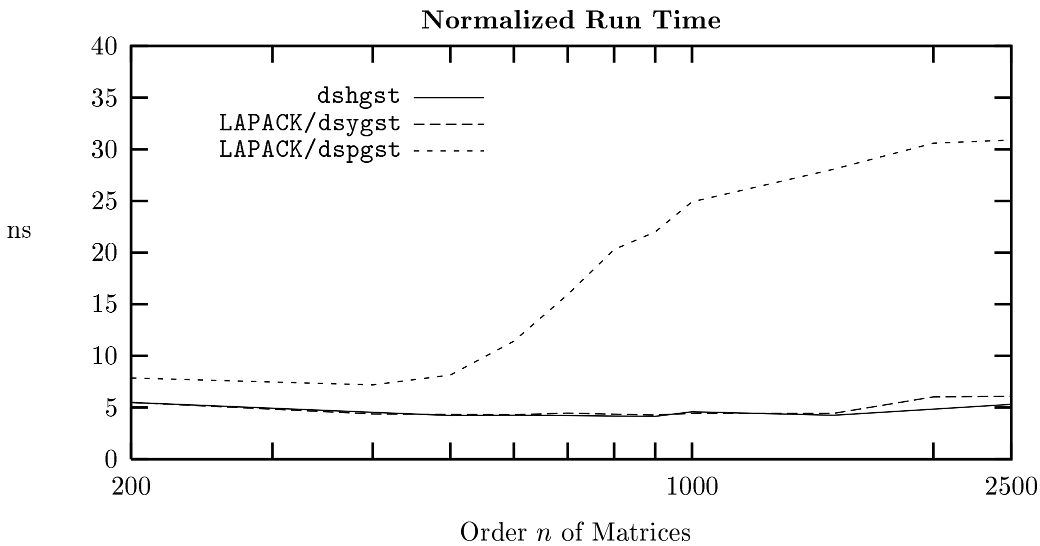

Figure 5.10: Normalized execution time (run time/n^3 in nanoseconds) of the transformation routine using conventional store (`LAPACK/dsygst`), packed storage (`LAPACK/dspgst`), and block packed storage (`dshgst` blocking factor $h_b = 64$).

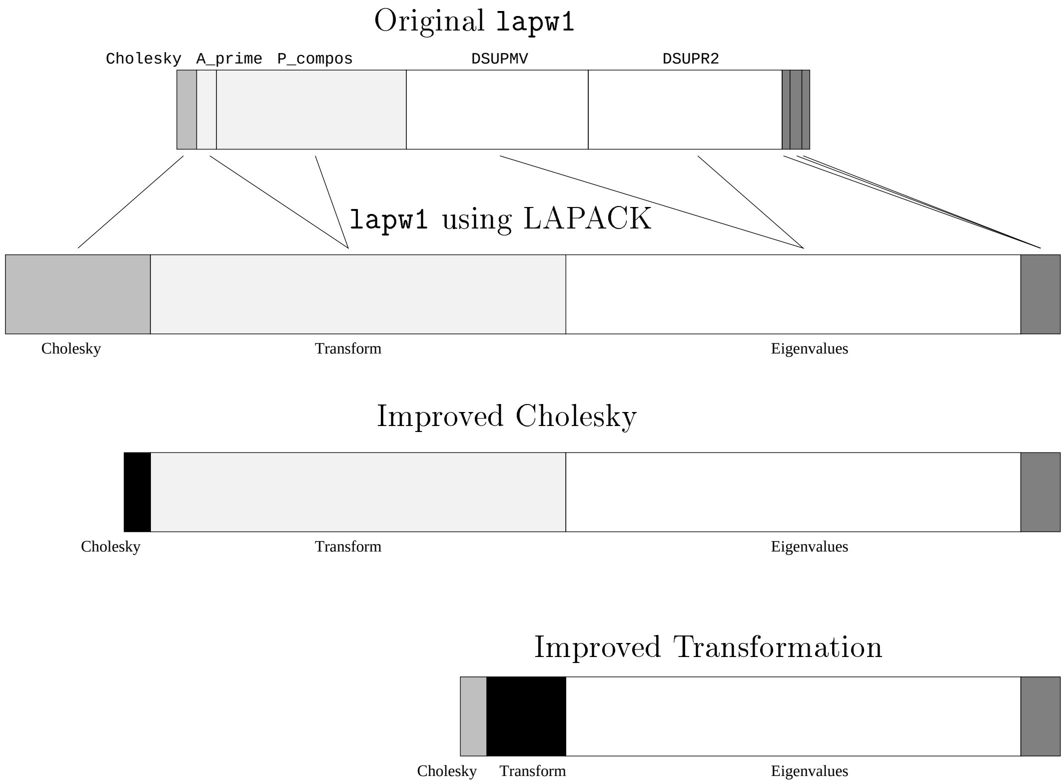

Figure 5.11: Second improvement of `lapw1`.

5.3 Eigensolver DSPEVX

As the solution of the symmetric eigenvalue problem is now the most time consuming routine of `lapw1`, any improvements to this routine would largely increase overall program efficiency.

The solution of the symmetric eigenvalue problem (SEP)[1] is carried out in the following steps:

1. The real symmetric matrix A is reduced to a real symmetric tridiagonal matrix T by using orthogonal similarity transformations:

$$A = QTQ^\top . \tag{5.1}$$

2. The real symmetric tridiagonal matrix T is then decomposed into

$$T = PDP^\top , \tag{5.2}$$

where P is orthogonal and D is a diagonal matrix. The diagonal elements of D are the eigenvalues of T (and of A), and the columns of P are the eigenvectors of T. The eigenvectors of A are the columns of QP.

Table 5.1 shows an interesting run time behavior for the packed and unpacked routines. If only selected eigenvectors are required, the packed routine performs badly. If all eigenvectors are required, the unpacked routine performs even worse. For the parameters used in `lapw1` a speedup of 25 % to 50 % should be possible to be achieved by combining the advantages of the two routines.

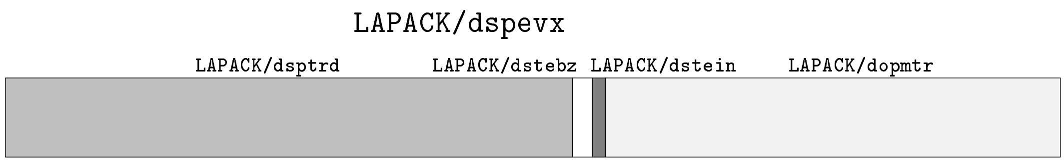

Figure 5.12: Execution time for `LAPACK/dspevx`, matrix size $n = 1500$, number of computed eigenvalues 500.

If the number of desired eigenvalues is approximately $n/10$, which is representative for WIEN 97, nearly all the time is spent in the tridiagonalization

[1] The LAPACK routine for solving a standard eigenvalue problem using unpacked storage is `LAPACK/dsyevx`, the corresponding routine using packed storage is `LAPACK/dspevx`, the newly developed routine described in this chapter is called `dshevx`.

Matrix size n	Number of Eigenvectors	`LAPACK/dsyevx`	`LAPACK/dspevx`	`dshevx`
900	100	13.8 s	15.0 s	10.4 s
900	300	16.7 s	19.9 s	13.1 s
900	800	26.7 s	66.9 s	21.9 s
900	899	28.3 s	76.6 s	23.3 s
900	900	93.0 s	59.9 s	89.0 s

Table 5.1: Execution times (in seconds) for `LAPACK/dsyevx` (unpacked storage), `LAPACK/dspevx` (packed storage), and `dshevx` (block packed storage).

rescaling	`LAPACK/dsptrd`	`LAPACK/dstebz`	`LAPACK/dstein`	`LAPACK/dopmtr`	scaling	ordering
90 ms	111.3 s	4.0 s	2.7 s	88.8 s	0.6 ms	5.1 ms

Table 5.2: Execution time for `LAPACK/dspevx`, matrix size $n = 1500$, number of computed eigenvalues 500.

step (`LAPACK/dsptrd`). The biggest section of the remaining time is spent in the back-transformation step (`LAPACK/dopmtr`), especially for large matrices and when many eigenvalues are to be computed. If the number of required eigenvalues is about $n/3$, the times of `LAPACK/dopmtr` and `LAPACK/dsptrd` are almost equal (Fig. 5.12 and Table 5.2). If nearly all eigenvalues are required, the time for `LAPACK/dopmtr` is five times higher than the time for `LAPACK/dsptrd`. Both the tridiagonalization (`LAPACK/dsptrd`) and the back-transformation (`LAPACK/dopmtr`) can be improved using the hierarchical blocking strategy. The routine `LAPACK/dsptrd` shows some algorithmic limits to this improvement, but for `LAPACK/dopmtr` a speedup comparable to the one of the Cholesky factorization should be achieved.

Table 5.2 and Figure 5.12 illustrate the run times spent in subroutines called by `LAPACK/dspevx`: first the matrix is scaled, then tridiagonalized (`LAPACK/dsptrd`), then the eigenvalues (`LAPACK/dstebz`) and the eigenvectors (`LAPACK/dstein`) are computed, afterwards the eigenvectors are transformed back to the solution of the input matrix (`LAPACK/dopmtr`) and at last the solution is rescaled and the eigenvalues and eigenvectors are ordered.

5.3.1 Tridiagonalization

The routine `LAPACK/dsptrd`, which is the most time consuming subroutine of the eigenproblem solver `LAPACK/dspevx`, served as a basis for the block packed routine `dshtrd`.

The tridiagonalization

$$T = Q^{-1}AQ \tag{5.3}$$

does not necessarily require that the matrix Q is actually computed. The method implemented in LAPACK computes Householder updates which give

$$Q = H_1 H_2 \dots H_{n-1} \ . \tag{5.4}$$

The effective size of these real elementary reflectors H_i decrease with ongoing computation. The matrices H_i are represented as

$$H_i = I - \tau_i \begin{pmatrix} 1 \\ v_i \end{pmatrix} \begin{pmatrix} 1 & v_i^\top \end{pmatrix} \tag{5.5}$$

with τ_i chosen appropriately so that $H_i^\top H_i = I$. The Householder vector v and the scalar β are computed, such that

$$H \begin{pmatrix} \alpha \\ x \end{pmatrix} = \begin{pmatrix} \beta \\ 0 \end{pmatrix} \tag{5.6}$$

with the vector $(\alpha \quad x)^\top$ being the first column vector of the matrix A.

The H_i do not need to be really computed, but may be applied to the matrix A as vector updates, whose complexity is only $O(n^2)$ instead of $O(n^3)$.

The operation count of `LAPACK/dsytrd` and its relatives is

$$\frac{4}{3}n^3 + 3n^2 - \frac{17}{6}n - 19$$

(Anderson et al. [1]). So a matrix size of 1000 would sum up to 1336 Mflop, which should be computed in about 6.5 s on a 200 Mflop/s processor (which is an average sustained performance of the SGI for many LAPACK routines). The measured time is over 20 s for the LAPACK routines. Figure 5.13 shows the run time for the routines `LAPACK/dsptrd` and `dsytrd-imp`, an improved blocked version of `LAPACK/dsytrd`, and, for comparison, the newly developed routine `dshtrd`, which is based on the ideas presented in this chapter. For $n \geq 700$ cache effects are clearly visible. The unblocked routines are most heavily effected by poor main memory performance, the influence of the blocked routines is much less noticeable. Cache effects cannot be removed completely, as it was possible for the

Cholesky factorization (Section 5.1) or for the reduction to a standard eigenvalue problem (Section 5.2). The reason for this impossibility are data dependencies which hinder full blocking of similarity transformations[2].

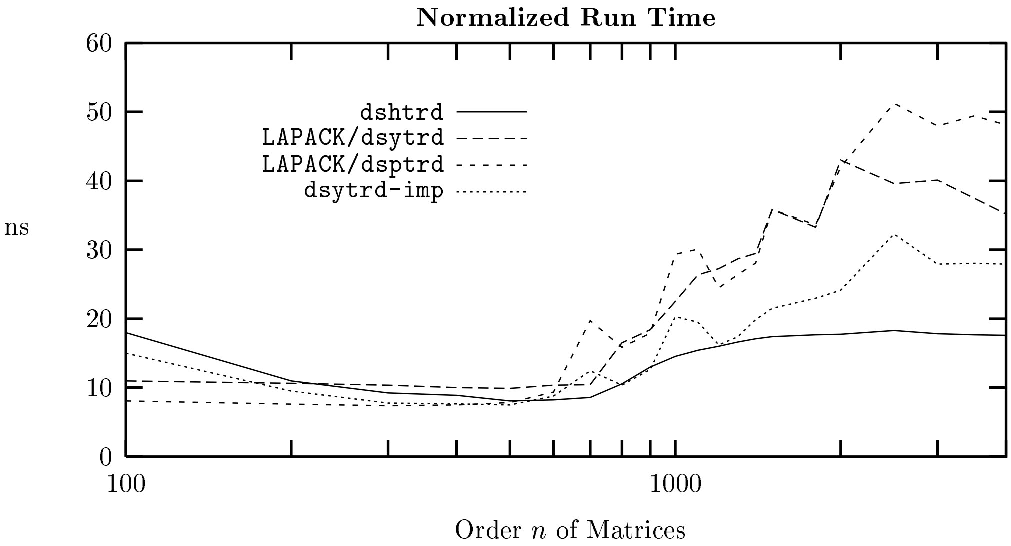

Figure 5.13: Normalized run time of tridiagonalization routines using block packed (`dshtrd`), full (`LAPACK/dsytrd`), and packed (`LAPACK/dsptrd`) storage scheme (blocking factor $h_b = 64$). Run time values are normalized to time/n^3 (in nanoseconds).

The blocked routine `LAPACK/dsytrd` consists mainly of two subroutine calls: one to `LAPACK/dlatrd` performs a partial tridiagonalization of the matrix, and a second to `BLAS-3/dsyr2k` which applies the transformation matrix obtained from `LAPACK/dlatrd` to the rest of the matrix. Figure 5.14 illustrates that for optimum block sizes—between 20 and 50—nearly all of the run time is spent in `LAPACK/dlatrd`, a routine which uses only Level 2 BLAS calls and thus cannot be improved easily.

DLATRD

The routine `LAPACK/dlatrd` consists mainly of calls to Level 2 BLAS: 6 calls to `BLAS-2/dgemv` (matrix-vector product), one call to `LAPACK/dlarfg` (to generate an elementary Householder reflector), some calls to Level 1 BLAS (`BLAS-1/dscal`

[2]Similarity transformations are of the form $B = X^{-1}AX$ with non-singular matrix X. A and B have the same eigenvalues.

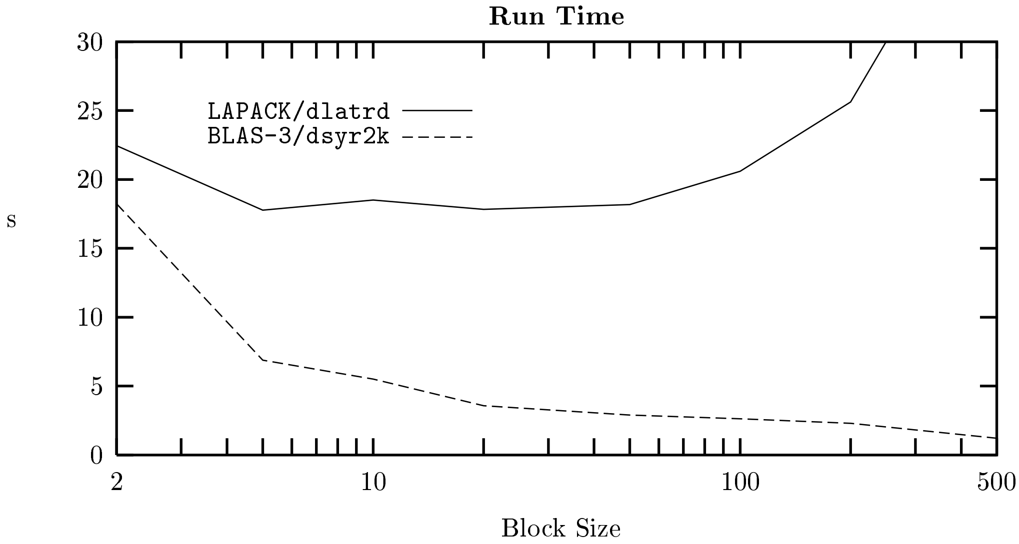

Figure 5.14: Execution time (in seconds) for partial reduction (LAPACK/dlatrd) and update of the rest of the matrix (dsyr2k) in the routine LAPACK/dsytrd.

and BLAS-1/daxpy), and a call to BLAS-2/dsymv (symmetric matrix-vector product). The call to BLAS-2/dsymv is the one where more than 80 % of the run time of LAPACK/dlatrd is spent. The reason for this high percentage is that this matrix-vector multiplication has to access the whole matrix.

Figure 5.15 shows that a huge amount of time is spent in BLAS-2/dsymv. For large matrices the time spent in BLAS-2/dsymv is mainly due to bad cache utilization. A block size of 64 would lead to best blocking results. To avoid the time consuming subroutine BLAS-2/dsymv, however, smaller block sizes have to be chosen.

The routines cannot be improved by full blocking because the BLAS-2/dsymv step cannot be avoided due to data dependencies.

5.3.2 Eigensystem Computation

If only a subset of all eigenvalues is required, the desired eigenvalues are computed using bisection in the routine LAPACK/dstebz. Eigenvectors are computed by inverse iteration in the routine LAPACK/dstein and transformed back by LAPACK/dormtr. The maximum number of iterations in LAPACK/dstein is invariably set to five (in parameter MAXITS). If the process does not converge in MAXITS iterations, the return parameter INFO is set to an error code.

The starting vector is a random vector (computed in LAPACK/dlarnv). Then the

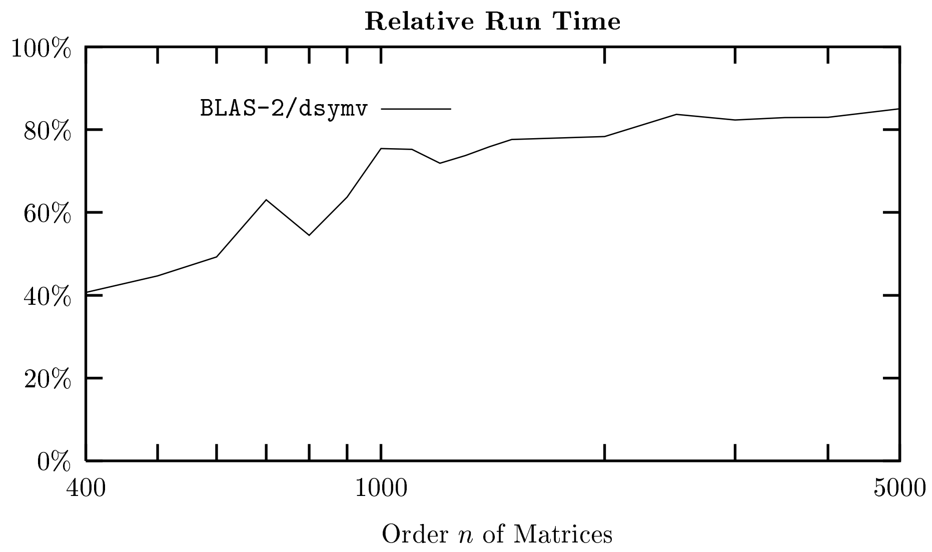

Figure 5.15: The relative amount of time spent in `BLAS-2/dsymv`. The figure is scaled so that the execution time needed for `dlatrd` is 100 %.

routine `LAPACK/dstein` first performs an LU factorization of the tridiagonal matrix for each eigenvector (using `LAPACK/dlagtf`). Then $LU = Pb$ is solved using the routine `LAPACK/dlagts` (P denotes the pivot matrix of the LU factorization). The infinity norm of the iteration step is computed and checked for convergence (using the variable `dtpcrt` and `BLAS-1/idamax`).

The computation of eigenvalues and eigenvectors is done on a (similar) symmetric tridiagonal matrix. Thus only $(2n-1)$ elements have to be stored. These fit fully into the caches of modern processors, so no additional blocking is necessary.

5.3.3 Back-transformation

The eigenvectors y of the tridiagonal matrix T have to be transformed back to the eigenvectors x of A by computing

$$x = Qy \tag{5.7}$$

with

$$Q = H_1 H_2 \cdots H_{n-1} \; . \tag{5.8}$$

The matrix Q is not stored explicitly, only the vectors v_i and the scalars τ_i are stored. The back-transformation

$$x = H_1 H_2 \cdots H_{n-1} y \tag{5.9}$$

is performed stepwise, starting with H_{n-1} and finishing with H_1. The advantage of this procedure is that H_i is a vector of length only $n-i$. So the first back-transformation steps can be performed very quickly.

The routine `LAPACK/dopmtr` computes the back-transformation of the eigenvectors of T to the eigenvectors of the standard eigenproblem.

`LAPACK/dopmtr` and `LAPACK/dormtr` overwrite the general real $m \times n$ matrix C with $Q \times C$, if called with `side=left` and `uplo=upper` and `trans=no`. Q is contained in the array A in the form of elementary reflectors. This is a packed array used in the routine `LAPACK/dopmtr`. The calling structure of the packed as well as the unpacked routine is found in Figure 5.16.

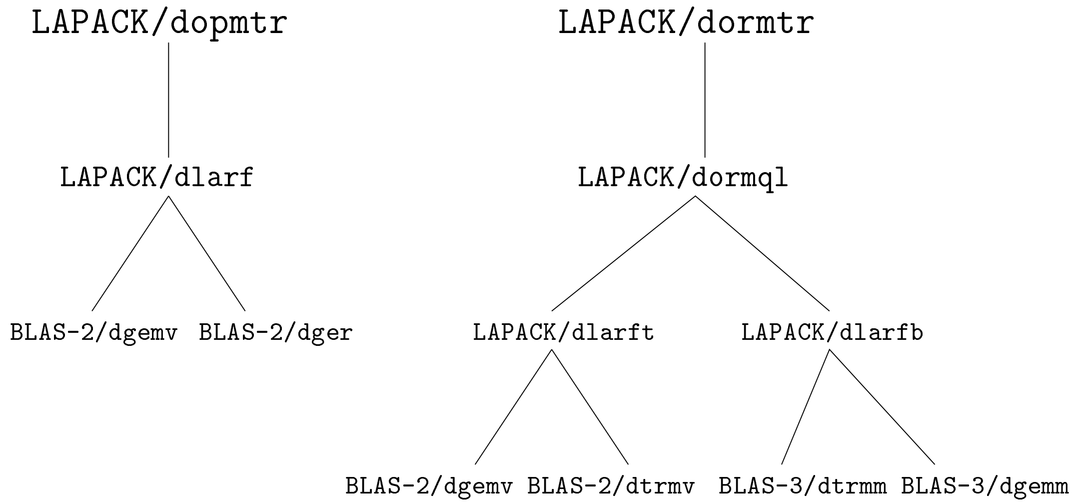

Figure 5.16: Calling structure of the routine `LAPACK/dopmtr` (packed storage scheme) and the unpacked routine `LAPACK/dormtr`.

The blocked, unpacked routine `LAPACK/dormtr` serves as a starting point towards a hierarchically blocked version of this algorithm. Figure 5.16 shows that many routines have to be rewritten in order to enable hierarchically blocked algorithms.

- The blocked version of the back-transformation routine `LAPACK/dormtr` is much faster if many eigenvectors are required. Since the back-transformation time is about the same as the triangularization time if the number of eigenvalues is a third of the order of the matrix, much time can be gained.
- The routine `LAPACK/dormtr` calls `LAPACK/dormql` (when called with the specific parameters of `lapw1`). This routine runs a loop over the blocks of the matrix, calling `LAPACK/dlarft` and `LAPACK/dlarfb` which do the computation.

	n	n_e	Triang.	Backtr.
LAPACK/dsyevx	1500	500	108.3 s	11.9 s
LAPACK/dspevx	1500	500	105.2 s	82.5 s
LAPACK/dsyevx	1000	100	29.3 s	0.9 s
LAPACK/dspevx	1000	100	22.7 s	1.5 s
LAPACK/dsyevx	1000	900	28.5 s	13.1 s
LAPACK/dspevx	1000	900	23.1 s	62.3 s

Table 5.3: Execution times for the back-transformation using packed and unpacked storage. n is the matrix size, n_e the number of eigenvalues computed. Only the times (in seconds) of triangularization and back-transformation are given. Other times are negligible.

- `LAPACK/dlarft` forms the triangular factor T of a real block reflector H of order n, where H is defined as a product of k elementary reflectors. This routine calls `BLAS-2/dgemv` and `BLAS-2/dtrmv` in a loop over k, which is at most the block size of the routine `LAPACK/dormql`.

  This routine is always applied to a matrix block which is equal to a storage block of the block packed storage scheme. So this routine may be used unchanged for the block packed routine `LAPACK/dohmtr`.

- `LAPACK/dlarfb` applies a real block reflector H or its transpose H' to a real $m \times n$ matrix C. The BLAS called are `BLAS-3/dtrmm` (three times) and `BLAS-3/dgemm` (twice). Both are routines of complexity $O(n^3)$.

  This routine is fortunately easy to decompose (see Figure 5.17) in order to apply block packed storage.

The run time of the routine `LAPACK/dlarft` is independent of the number of eigenvectors, whereas the run time `LAPACK/dlarfb` increases linearly with the number of eigenvectors. The break even point is at about ten eigenvectors. Since more than ten eigenvectors are required by WIEN 97, fortunately most of the time of `LAPACK/dormtr` is spent in the routine `LAPACK/dlarfb`.

The routine `LAPACK/dormql`, which performs the blocking, is the routine where the introduction of blocked storage mode and hierarchical blocking requires the most substantial coding effort. Since only a small performance gain is possible, a hierarchically blocked back transformation has not yet been implemented.

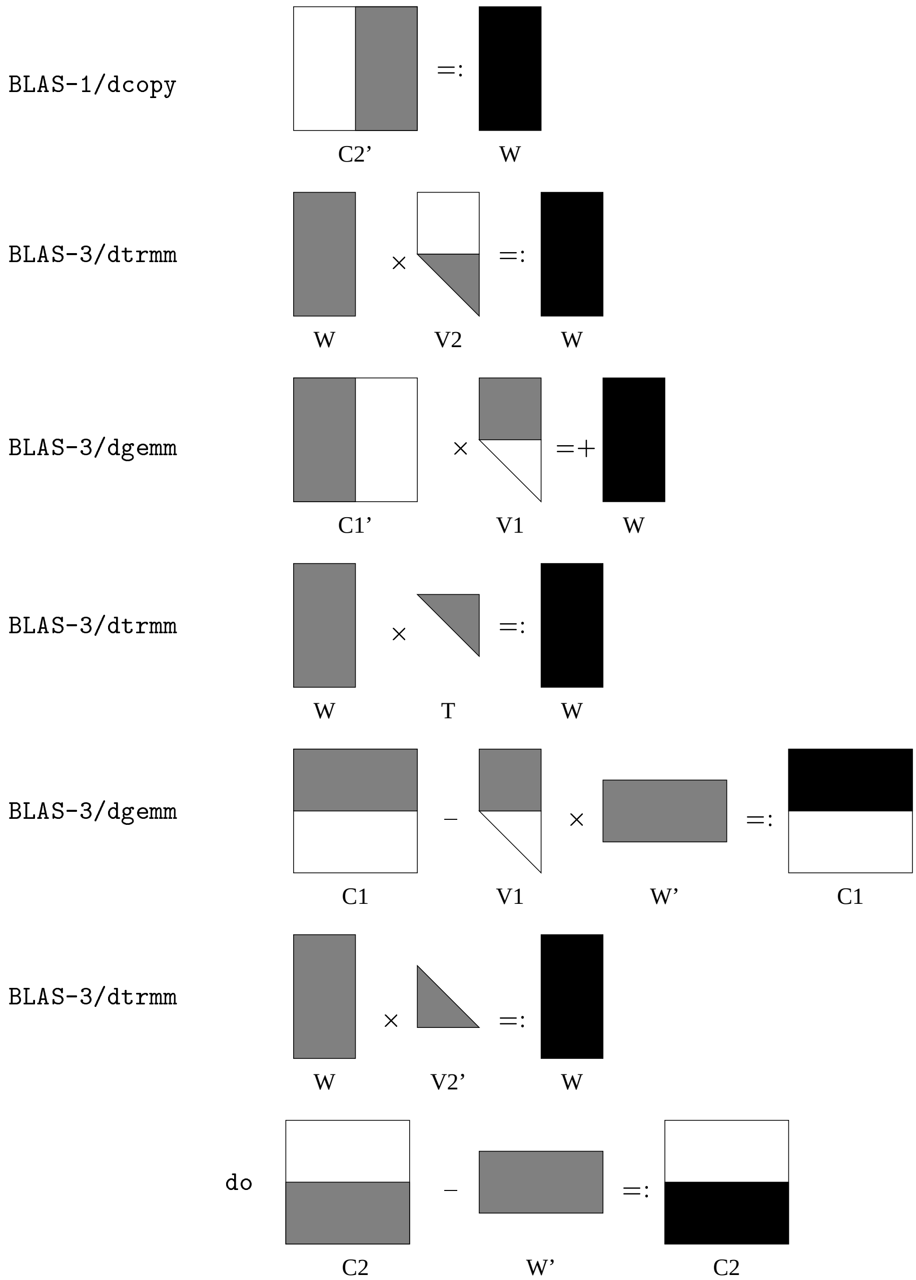

Figure 5.17: Matrix access patterns in the routine LAPACK/dlarfb.

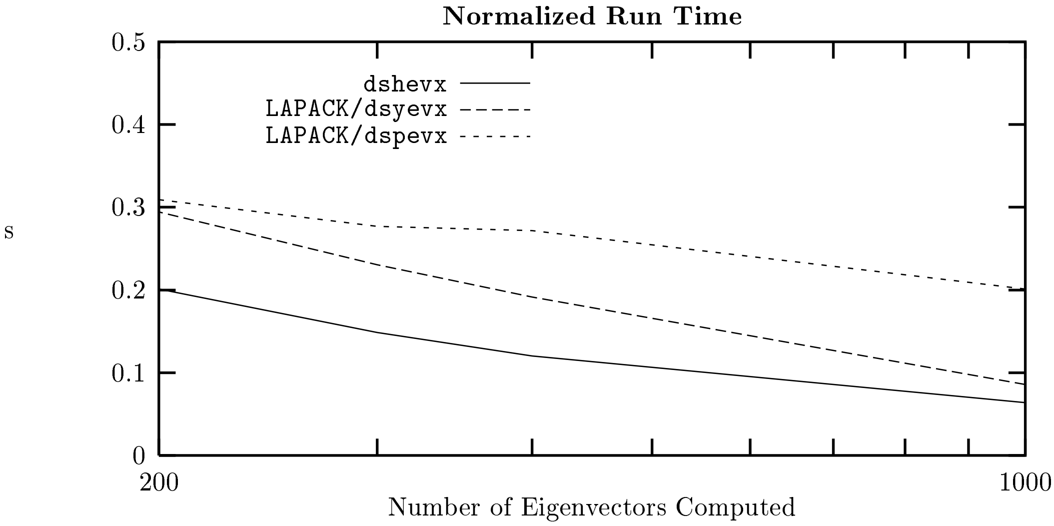

Figure 5.18: Eigenproblem computation using full storage (`LAPACK/dsyevx`), packed storage (`LAPACK/dspevx`), and the block packed storage scheme (`dshevx`) (blocking factor $h_b = 64$). Matrix size $n = 1251$. Time is normalized to time/number of eigenvectors.

5.3.4 Performance Improvement

Figure 5.18 shows that for small numbers of eigenvectors—as required by WIEN 97—the new routine `dshevx` is performing better than the corresponding LAPACK routines. If many eigenvectors are required, some modifications to improve the back-transformation are still possible.

The run time improvements resulting from applying the blocked storage scheme and the concept of hierarchical blocking are even better than for the blocked, but not packed routine. Results are shown in Figure 5.19 and Table 5.4.

	Cholesky	Transform	Eigenvalues	Backtransform
LAPACK	21.6 s	58.5 s	63.7 s	4.1 s
New	3.5 s	9.4 s	35.1 s	4.0 s

Table 5.4: Third improvement of `lapw1`. This is the first version which is faster than the highly tuned original version.

Total computation times of the newly developed routine compared to the original, highly tuned program are given in Chapter 7.

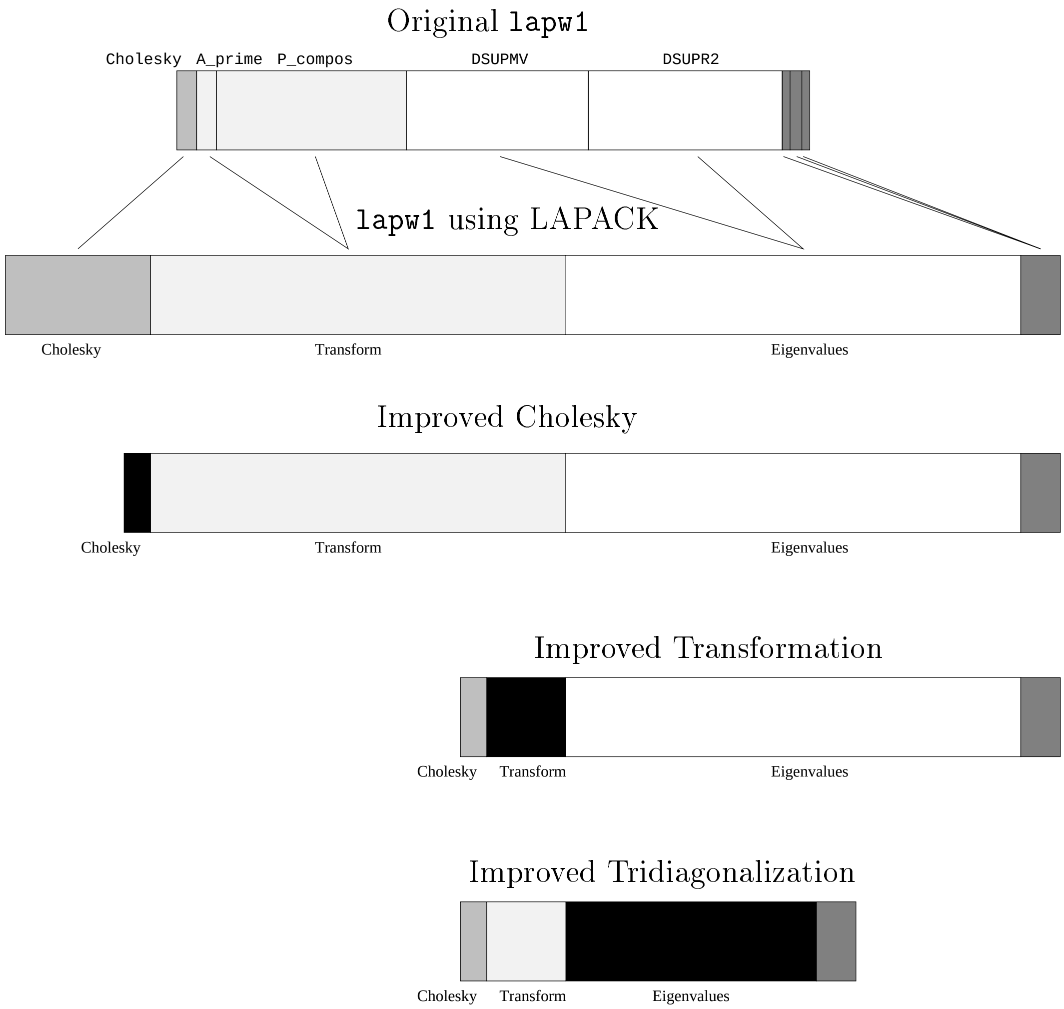

Figure 5.19: Third improvement of `lapw1`. This is the first version which is faster than the highly tuned original version.

Since the blocking algorithm slows down if block sizes become too small, the routine `dshtrd` switches automatically to the packed LAPACK algorithm for a block size up to ten (hard coded parameter MINHB in the routine `dshtrd`). This is a value which should not be reached in actual program runs, however.

A limiting factor for the blocking which should not be forgotten is that a block of the matrix may not fit into the cache if it becomes very large. On the SGI Power Challenge XL the cache size is 2 MB. This means that for large matrices a whole block of columns does not fit into the cache. For a matrix size of 2000 and a block size of 128 and DOUBLE PRECISION data format (8 byte), the cache requirement is $2000 \times 128 \times 8 = 2.048\,\text{MB}$, which slightly exceeds the cache size. Another blocking level below this routine is required. This is done in Level 3 BLAS.

A matrix size $n = 10\,000$ for DOUBLE PRECISION data would only perform well up to a block size of 24. Thus a problem of this size with COMPLEX*16 data should be run with a block size of 12. Using such small block sizes limits the performance improvement of the newly developed routines by some percent. For very big problems the cache size should be adequately sized, or else a third blocking loop has to be inserted consequently. This is already done due to the blocked storage scheme for some routines, others, however, do not require this third loop and are thus only running using two blocking levels.

5.3.5 Improved Functionality

Changes in the functionality are made, too, during the transformation of the routine `LAPACK/dspevx` to `dshevx`:

- The number of eigenvectors must not exceed the size of its allocated field. So after the eigenvalue routine `LAPACK/dstebz` the number of eigenvalues between the lower and upper limits is examined and, if necessary, the eigenvector routine `LAPACK/dstein` is called with modified parameters.
- The number of eigenvalues below the lower limit is computed and returned to the calling routine.

5.4 Back-transformation Routine DTPSV

The eigenvectors of the standard eigenproblem are transformed back to the eigenvectors of the original generalized eigenproblem using a loop over `BLAS-2/dtpsv`. This loop could be rewritten using a blocked algorithm. However, this loop was neglected because the time needed for executing this loop is much smaller than the time required by the other routines whose improvements were described in this chapter.

Chapter 6

Improved Complex Routines

Complex versions have been developed for all of the tuned routines. This was of great importance, since the original `lapw1` program uses slow LAPACK routines in the complex case.

Nevertheless, the speedup is not as high as in the real case, because the complex version is not as memory bound as the double precision (real) version. A multiply-add ($a = c \times d + e$) operation on complex data involves four multiplications and four additions on real data ($a_r = c_r \times d_r - c_i \times d_i + e_r$ and $a_c = c_r \times d_i + c_i \times d_r + e_i$), whereas the data requirement is only doubled. So, the complex version needs only half as many memory references per double precision floating-point operation as the double precision real version.

6.1 Cholesky Factorization Routine ZPPTRF

A hierarchically blocked version of `LAPACK/zpptrf` (which computes a Cholesky factorization) has been developed and tested. The differences of the resulting routines `zphtrf` and `dphtrf` are found in Appendix B.

For matrices of size $n \approx 1000$ a speedup of 500 % for the Cholesky factorization and a speedup of 14 % for `lapw1c` is achieved (Figure 6.4) by the first improvement, a total of 45 % is won by improving both the Cholesky factorization and the transformation steps.

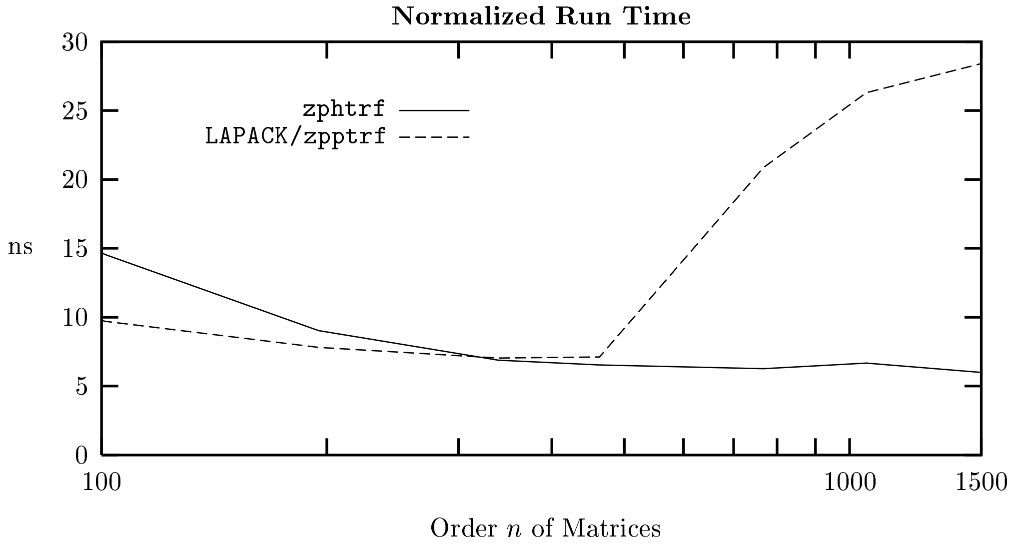

Figure 6.1: Execution times of the packed LAPACK routine `LAPACK/zpptrf` and the hierarchically blocked routine `zphtrf`. (Run time/n^3 in nanoseconds)

6.2 Transformation Routine ZHPGST

The routine `zhhgst` has been developed in order to replace the calls to the LAPACK packed mode routine `LAPACK/zhpgst`. Figure 6.2 shows the achieved speedup, which is not as high as for the REAL version of the routine (Figure 5.10), but still very good and still better than the blocked LAPACK routine `LAPACK/zhegst`. An interesting fact is that the packed LAPACK routine performs for small matrices (less than 400) better than the unpacked as well as the newly developed routine. The newly developed routine may be called with a blocking parameter of one to use the packed LAPACK routine.

6.3 Eigenproblem Solver ZHPEVX

Run time measurements for the new and the packed storage LAPACK routines have been made (Figure 6.3). They show a certain speedup, but, as mentioned in chapter 5, the remaining matrix-vector product is so limiting that the gained speedup is less than for the Cholesky factorization or the transformation routine.

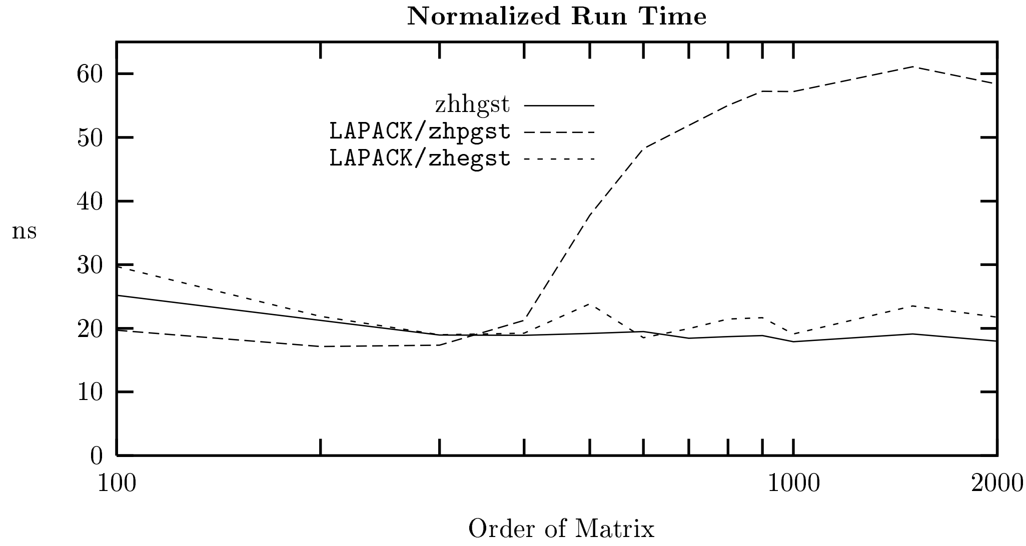

Figure 6.2: Execution times of the packed routine `LAPACK/zhpgst`, the unpacked (blocked) routine `LAPACK/zhegst`, and the block packed routine `zhhgst` (block size $h_b = 64$). (Run time/n^3 in nanoseconds)

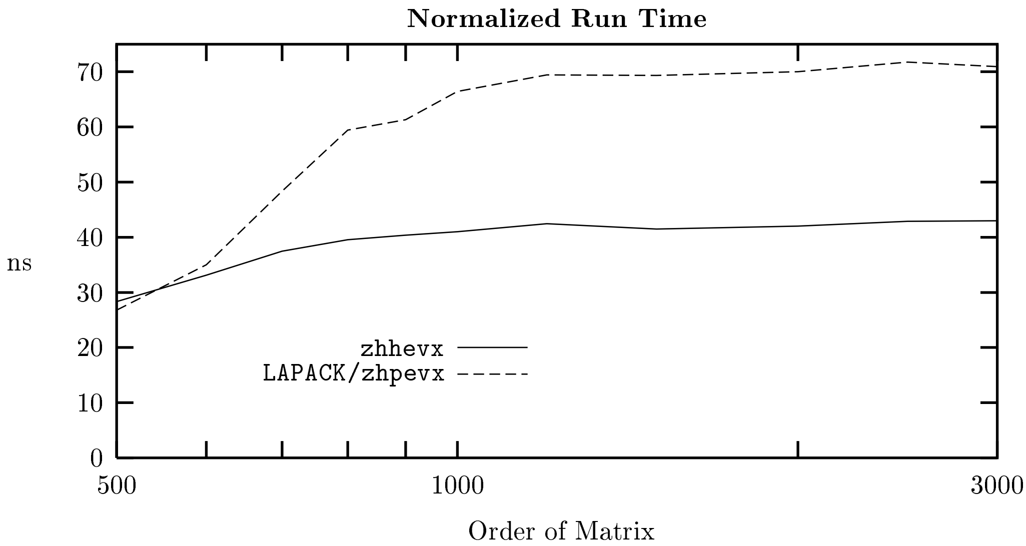

Figure 6.3: Execution times for the packed storage routine `LAPACK/zhpevx` and the block packed routine `zhhevx`. Block size as well as the number of computed eigenvectors is 32. (Run time/n^3 in nanoseconds)

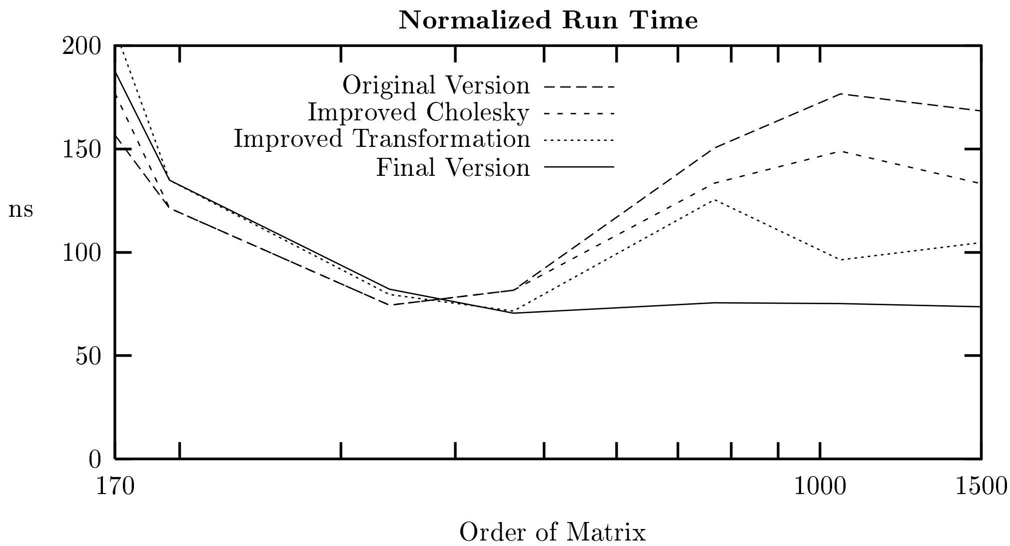

Figure 6.4: Execution times for different versions of the program `lapw1c`. (Run time/n^3 in nanoseconds)

Chapter 7

Performance Results for Real Data

The columns of the tables in this section are the following.

Size: the order n of the matrix;

Cholesky: the run time for the Cholesky factorization;

Transformation: the run time to transform the generalized eigenvalue problem to a standard eigenvalue problem;

Eigenvalue: the run time to compute the eigenvalues and selected eigenvectors (using matrix-matrix operations plus one matrix-vector product which requires most of the run time);

Back-transformation: the run time to transform the eigenvectors of the standard eigenvalue problem back to a solution of the general eigenvalue problem (utilizing matrix-vector operations);

LAPW1: total run time for `lapw1` using the new routines;

Original: total run time for `lapw1` using the original EISPACK based routines;

LAPACK: total run time for `lapw1` using LAPACK routines.

The figures and tables of this chapter show the overall execution time of the program `lapw1` using DOUBLE PRECISION data. The computer systems which have been used in the run time experiments are listed in Table 7.1. In this table the following information is given.

System: the name of the vendor and of the computer system;

Processor: the processor and its clock frequency (in MHz);

Peak Perf.: peak performance of the processor (in Mflop/s);

Memory Levels: the size of the memory levels (in KB, MB or GB), beginning from one or several levels of cache and ending with the size of the main memory;

OS, BLAS: the operating system, and the BLAS versions used;

Compiler, Arguments: the name of the compiler, and the arguments of the compiler run.

The peak performance values of Table 7.1 have beed used to scale the performance tables and figures of the Chapters 7 and 8.

System	Processor	Peak Perf.	Memory Levels	OS BLAS	Compiler Arguments
IBM RS/6000 Model 410	Power PC 601 80 MHz	80	32 KB 64 MB	AIX 3.2 ESSL	xlf -O2
IBM RS/6000 Model 550	Power 44 MHz	88	32 KB 128 MB	AIX 3.2 ESSL	xlf -O2
(Local Vendor) Pentium PC	Intel Pentium 166 MHz	166	8 KB 512 KB 64 MB	Linux PHiPAC	g77 -O3
HP 9000 K-200	PA-7200 100 MHz	200	256 KB 512 MB	HP-UX 10.01 libblas.a	f77 B.10.01 -O
IBM SP2 Thin Node	Power2 66.7 MHz	266	64 KB 256 MB	AIX 3.2 ESSL P2	xlf -O2
IBM SP2 Thin2 Node	Power2 66.7 MHz	266	128 KB 256 MB	AIX 4.1 ESSL P2	xlf -O2
IBM SP2 Wide Node	Power2 66.7 MHz	266	256 KB 512 MB	AIX 4.1 ESSL P2	xlf -O2
(Local Vendor) Pentium II PC	Intel Pentium II 266 MHz	266	16 KB 512 KB 128 MB	Linux Fortran	g77 -O3
IBM RS/6000 Model 390	Power2 67 MHz	268	64 KB 1 MB 256 MB	AIX 4.1.4 ESSL 2.2.2	xlf 3.2.4 -O
DEC Alpha-Server 2100	Alpha 21064a 275 MHz	275	16 KB 4 MB 512 MB	OSF1 V3.2 dxml	Digital DFAO 4.0 +O3 -pipeline -transform_loops
SGI Power Challenge	MIPS R8000 90 MHz	360	64 KB 4 MB 768 MB	IRIX 64 6.2 complib. sgimath	Ragnarok 6.2 -mips4 -O3 -n32
SGI Power Challenge XL	MIPS R10000 195 MHz	390	32 KB 2 MB 4 GB	IRIX 64 6.2 complib. sgimath	f77 MIPS -mips4 -O3 -n32
IBM SP2 P2SC Node	Power2 120 MHz	480	128 KB 256 MB	AIX 4.1 ESSL P2	xlf -O2
HP 9000 K-460	PA-8000 180 MHz	720	1 MB 384 MB	HP-UX 10.20 libblas.1205.a	f77 B.10.20.02 +O2 +U77
DEC Alpha-Server 2100	Alpha 21164 433 MHz	866	8 KB 96 KB 4 MB $\geq$512 MB	OSF1 V4.0 dxml	Digital DFAO 4.1 +O3 -pipeline -transform_loops
Quant-X Alpha	Alpha 21164a 500 MHz	1000	8 KB 96 KB 2 MB 512 MB	Linux PHiPAC	Digital DFAO 4.0 +O3 -pipeline -transform_loops
NEC SX-4B/2	112,5 MHz	1800	1 GB	Super-UX 6.2 libblas.a	f77

Table 7.1: Hardware platforms.

7.1 SGI Power Challenge XL

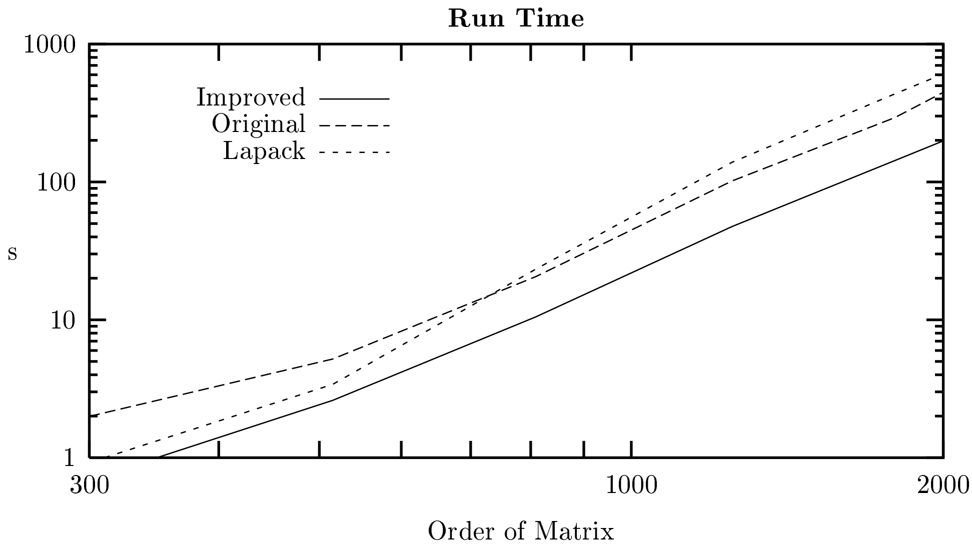

Figure 7.1: Total run time of `lapw1` on an SGI Power Challenge XL.

n	Cholesky	Transf.	Eigenv.	`lapw1`	Original	LAPACK
283	0.0 s	0.1 s	0.4 s	0.6 s	1.8 s	0.8 s
515	0.2 s	0.6 s	1.6 s	2.6 s	5.2 s	3.4 s
809	0.8 s	2.4 s	6.0 s	10.5 s	20.6 s	23.3 s
1251	3.4 s	9.0 s	31.5 s	47.6 s	101.6 s	138.9 s
1795	10.1 s	26.7 s	98.8 s	143.1 s	292.7 s	435.6 s
2457	25.8 s	71.2 s	259.4 s	370.7 s	997.3 s	1146.5 s
3385	67.5 s	191.2 s	684.4 s	970.9 s	1842.8 s	3041.5 s
4399	152.6 s	443.6 s	1574.3 s	2222.2 s	4122.6 s	
5615	323.5 s	947.6 s	3077.2 s	4443.7 s	8487.7 s	

Table 7.2: Total run time of `lapw1` on an SGI Power Challenge XL.

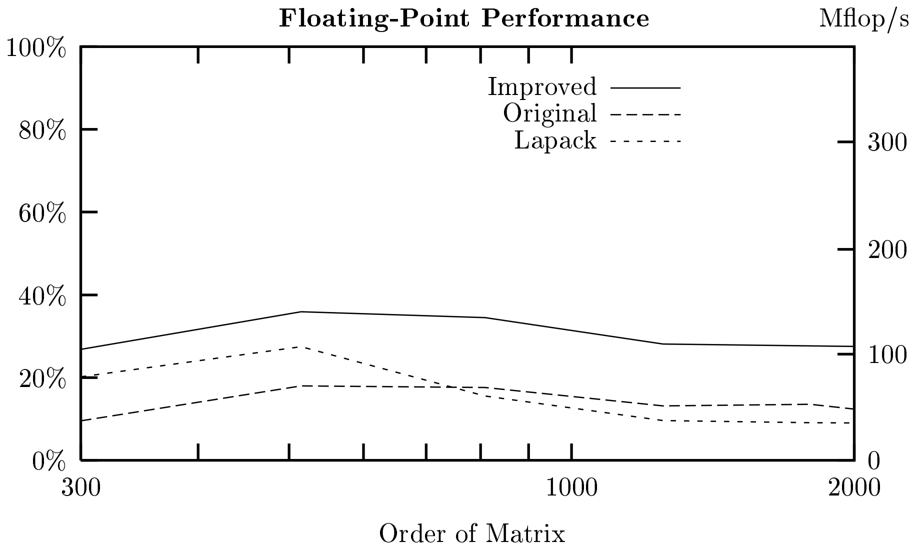

Figure 7.2: Performance of `lapw1` on an SGI Power Challenge XL.

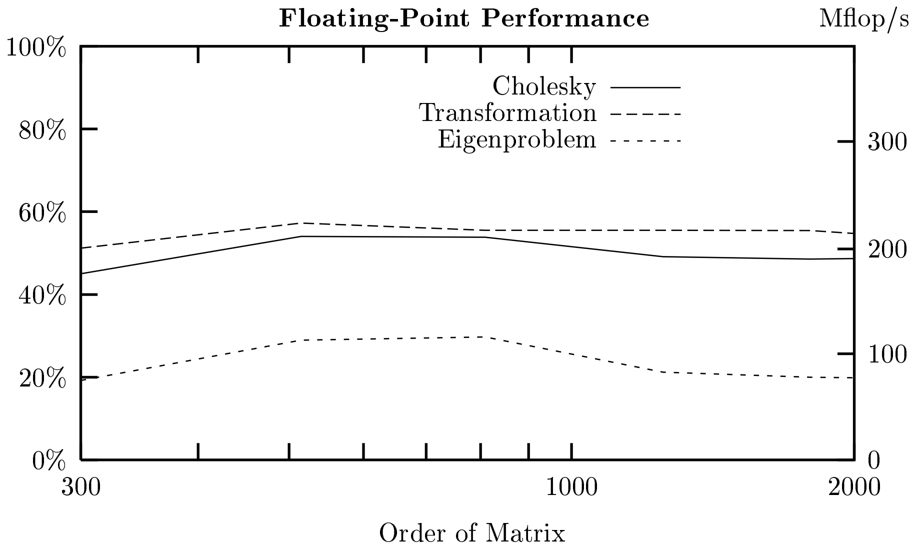

Figure 7.3: Performance of `lapw1` on an SGI Power Challenge XL.

7.2 SGI Power Challenge

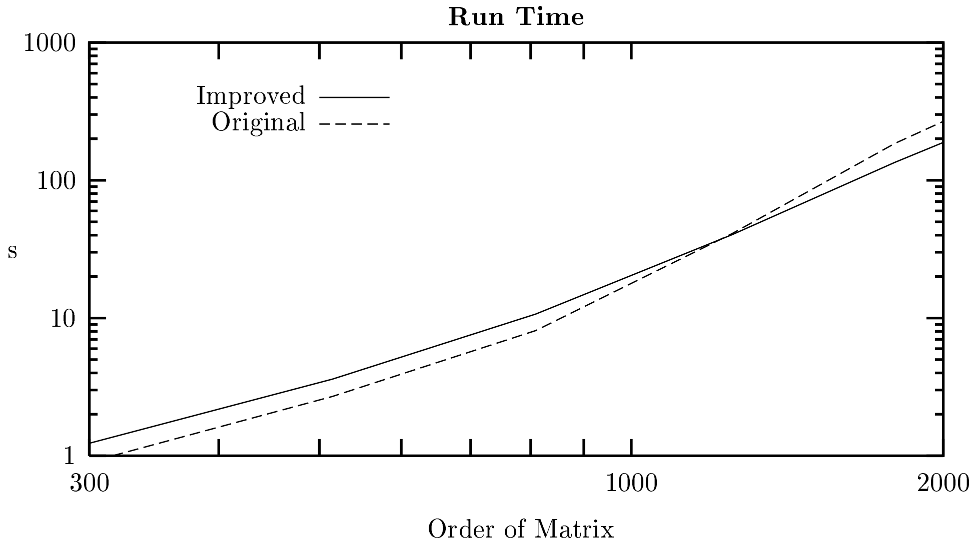

Figure 7.4: Total run time of `lapw1` on an SGI Power Challenge.

n	Cholesky	Transf.	Eigenv.	Backtr.	`lapw1`	Original
283	0.0 s	0.1 s	0.8 s	0.0 s	1.1 s	0.8 s
515	0.2 s	0.7 s	2.3 s	0.1 s	3.6 s	2.7 s
809	0.9 s	2.8 s	6.3 s	0.4 s	10.7 s	8.1 s
1251	3.6 s	10.6 s	23.2 s	2.6 s	40.3 s	41.1 s
1795	11.3 s	31.7 s	84.5 s	7.0 s	134.7 s	185.0 s
2457	29.1 s	82.1 s	232.2 s	12.9 s	356.5 s	540.5 s

Table 7.3: Total run time of `lapw1` on an SGI Power Challenge.

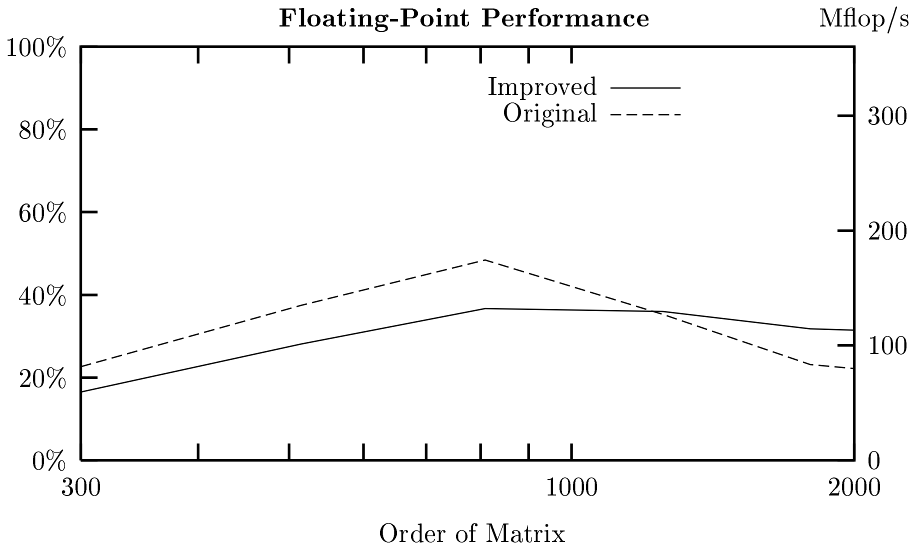

Figure 7.5: Performance of `lapw1` on an SGI Power Challenge.

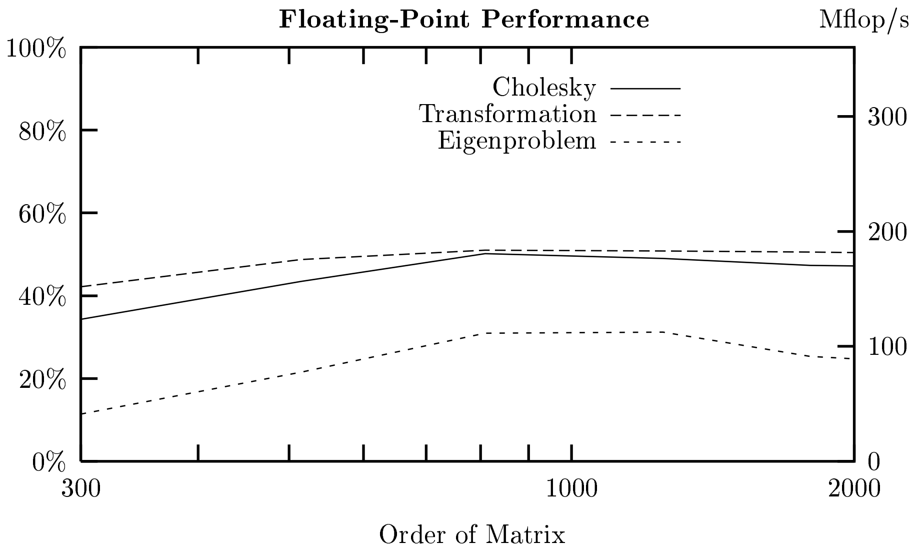

Figure 7.6: Performance of `lapw1` on an SGI Power Challenge.

7.3 Alpha Workstation Clone

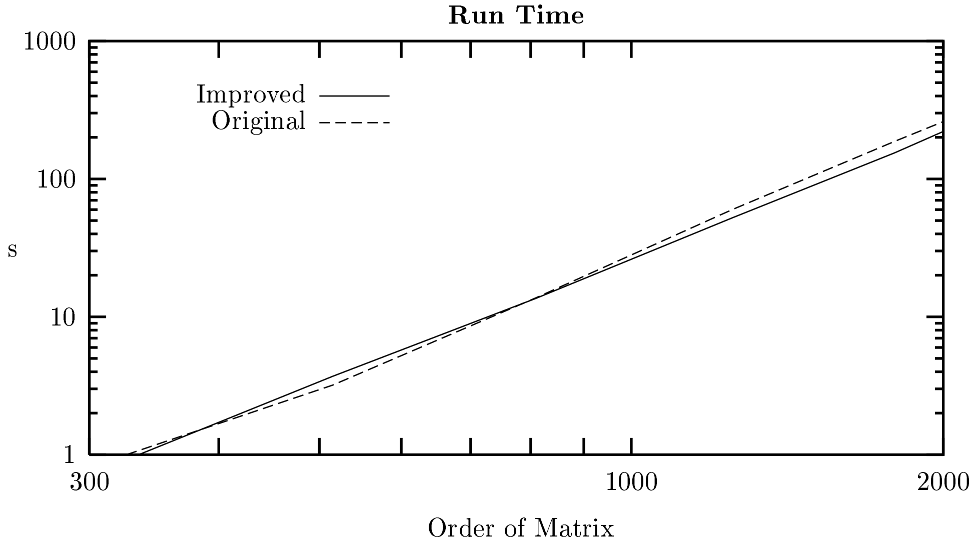

Figure 7.7: Total run time of `lapw1` on an Alpha workstation (clone) using the Fortran optimizing compiler kapf77.

n	Cholesky	Transf.	Eigenv.	Backtr.	`lapw1`	Original
283	0.0 s	0.1 s	0.4 s	0.0 s	0.6 s	0.7 s
515	0.2 s	0.8 s	2.3 s	0.3 s	3.7 s	3.2 s
809	0.8 s	2.9 s	9.0 s	0.8 s	13.6 s	13.7 s
1251	3.2 s	10.8 s	35.4 s	2.4 s	52.1 s	59.8 s
1795	9.6 s	32.4 s	107.3 s	5.2 s	154.7 s	187.4 s
2457	28.5 s	95.6 s	300.1 s	9.9 s	434.2 s	486.9 s
3385	70.8 s	233.8 s	812.0 s	18.9 s	1135.8 s	1290.0 s

Table 7.4: Total run time of `lapw1` on an Alpha workstation (clone) using the Fortran optimizing compiler kapf77.

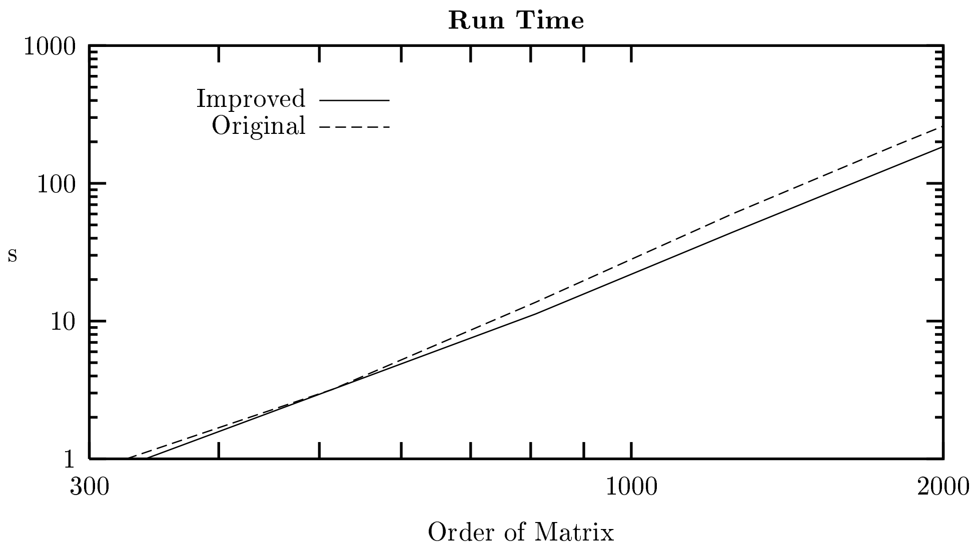

Figure 7.8: Total run time of `lapw1` on an Alpha workstation (clone) using the `BLAS-3/dgemm` optimized with PhiPac.

n	Cholesky	Transf.	Eigenv.	Backtr.	`lapw1`	Original
283	0.0 s	0.1 s	0.4 s	0.0 s	0.6 s	0.7 s
515	0.1 s	0.5 s	2.1 s	0.2 s	3.2 s	3.2 s
809	0.6 s	2.0 s	7.6 s	0.8 s	11.3 s	13.7 s
1251	2.6 s	7.7 s	31.2 s	2.4 s	44.1 s	59.8 s
1795	7.7 s	22.4 s	97.6 s	5.2 s	133.2 s	187.4 s
2457	20.4 s	57.1 s	257.7 s	9.8 s	345.3 s	486.9 s
3385	54.8 s	154.8 s	691.6 s	18.7 s	920.2 s	1290.0 s
4399	127.3 s	357.2 s	1544.1 s	33.7 s	2062.6 s	

Table 7.5: Total run time of `lapw1` on an Alpha workstation (clone) using the `BLAS-3/dgemm` optimized with PhiPac.

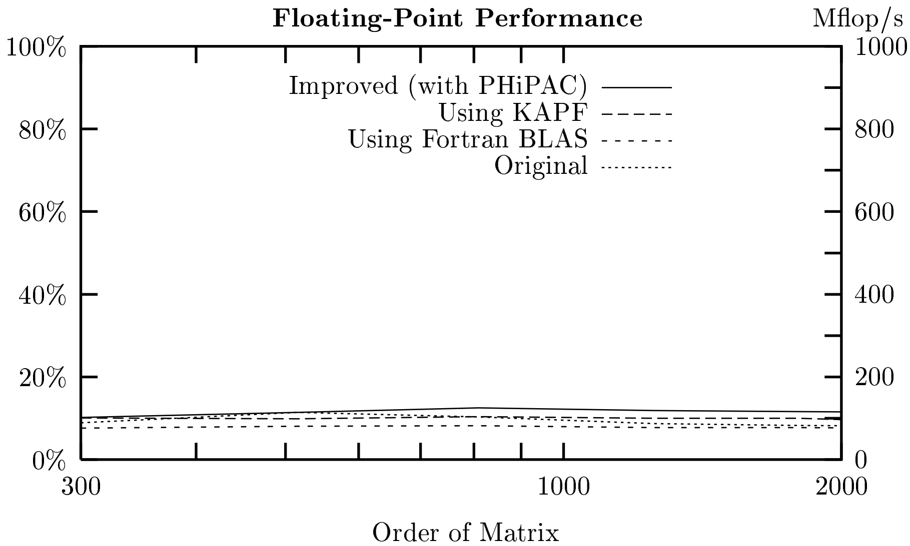

Figure 7.9: Performance of `lapw1` on an Alpha workstation (clone).

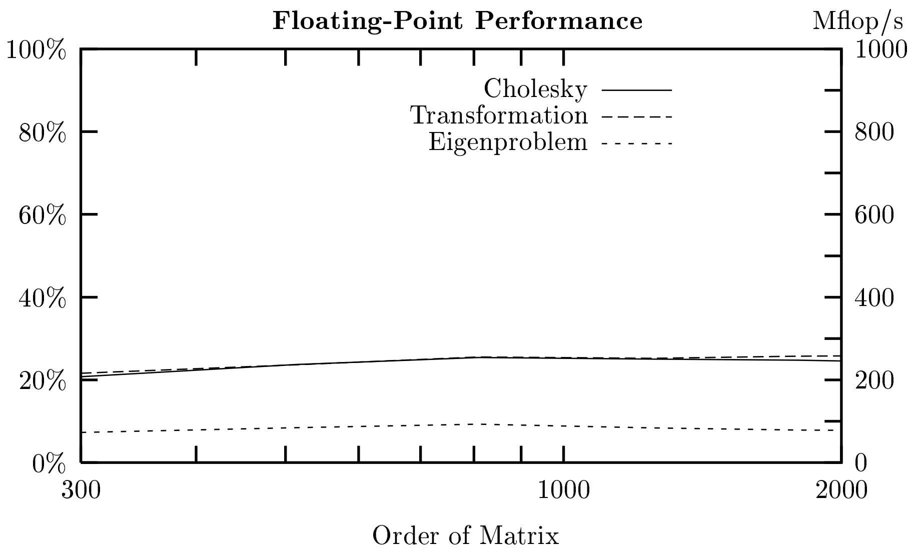

Figure 7.10: Performance of `lapw1` on an Alpha workstation (clone).

7.4 HP Workstation K460

n	Cholesky	Transf.	Eigenv.	Backtr.	`lapw1`	Original
283	0.1 s	0.3 s	1.0 s	0.0 s	1.6 s	1.0 s
515	0.6 s	2.2 s	3.8 s	0.2 s	7.1 s	4.5 s
809	2.8 s	8.6 s	14.0 s	2.0 s	27.5 s	20.6 s
1251	10.6 s	31.7 s	49.5 s	4.8 s	96.7 s	78.9 s
1795	31.8 s	93.6 s	142.2 s	9.8 s	277.6 s	232.8 s
2457	82.0 s	239.5 s	363.0 s	18.4 s	703.0 s	594.3 s
3385	218.0 s	629.0 s	952.1 s	35.1 s	1834.3 s	1548.7 s

Table 7.6: Total run time of `lapw1` on an HP K 460 workstation using a program optimized for the HP 700 workstation line.

n	Cholesky	Transf.	Eigenv.	Backtr.	`lapw1`	Original
283	0.1 s	0.2 s	0.6 s	0.0 s	1.0 s	0.5 s
515	0.5 s	1.5 s	2.2 s	0.1 s	4.4 s	2.1 s
809	2.1 s	6.0 s	8.8 s	0.9 s	18.0 s	11.6 s
1251	7.9 s	22.5 s	33.7 s	2.1 s	66.4 s	47.5 s
1791	23.6 s	66.8 s	98.8 s	4.4 s	193.9 s	141.4 s
2457	60.8 s	172.0 s	254.5 s	8.3 s	495.8 s	364.4 s
3385	159.9 s	452.1 s	663.6 s	15.9 s	1291.6 s	946.0 s

Table 7.7: Total run time of `lapw1` on an HP workstation using a self-compiled Blas.

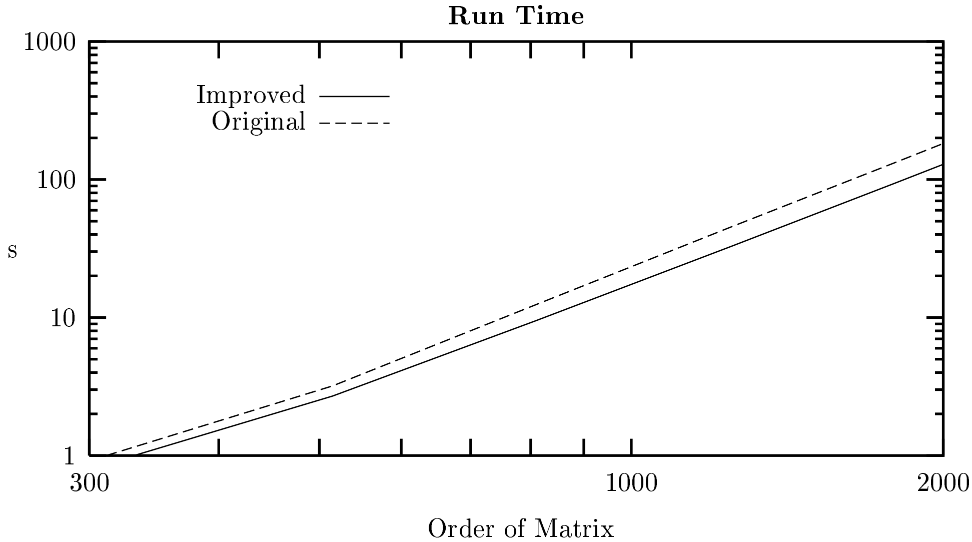

Figure 7.11: Total run time of `lapw1` on an HP workstation using a beta version of the BLAS, which will be available with a future version of the HP-UX operating system.

n	Cholesky	Transf.	Eigenv.	Backtr.	`lapw1`	LAPACK
283	0.0 s	0.1 s	0.5 s	0.0 s	0.7 s	0.8 s
515	0.2 s	0.5 s	1.8 s	0.1 s	2.7 s	3.2 s
809	0.7 s	2.1 s	6.2 s	0.4 s	9.5 s	12.4 s
1251	2.8 s	8.1 s	21.0 s	0.9 s	33.0 s	45.6 s
1795	8.5 s	24.4 s	58.9 s	2.0 s	93.9 s	132.5 s
2457	22.1 s	63.3 s	146.6 s	3.7 s	235.8 s	335.7 s
3385	59.6 s	173.3 s	380.2 s	7.2 s	620.5 s	873.6 s

Table 7.8: Total run time of `lapw1` on an HP workstation using a beta version of the BLAS, which will be available with a future version of the HP-UX operating system.

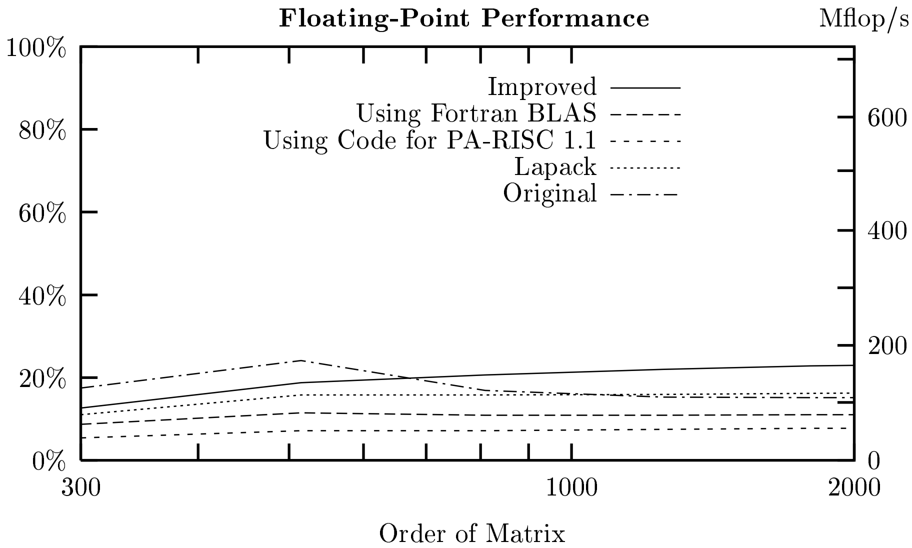

Figure 7.12: Performance of `lapw1` on an HP workstation.

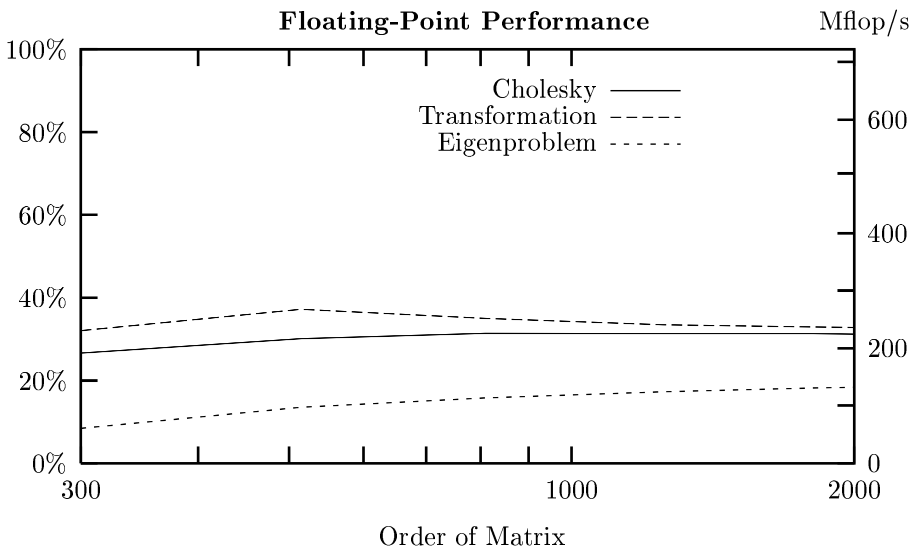

Figure 7.13: Performance of `lapw1` on an HP workstation.

7.5 IBM Workstation RS/6000

n	Cholesky	Transf.	Eigenv.	`lapw1`	Original	LAPACK
283	0.1 s	0.4 s	2.0 s	2.8 s	2.5 s	3.5 s
515	0.8 s	2.6 s	7.1 s	11.0 s	10.7 s	15.1 s
809	3.1 s	9.7 s	21.6 s	35.6 s	35.1 s	51.7 s
1251	11.5 s	35.2 s	69.0 s	118.1 s	117.9 s	178.8 s
1795	34.3 s	103.5 s	190.3 s	333.0 s	336.2 s	508.6 s
2457	90.1 s	262.2 s	460.2 s	830.2 s	858.0 s	1280.1 s
3385	235.4 s	695.1 s	1209.0 s	2162.0 s	2299.7 s	

Table 7.9: Total run time of `lapw1` on an IBM workstation.

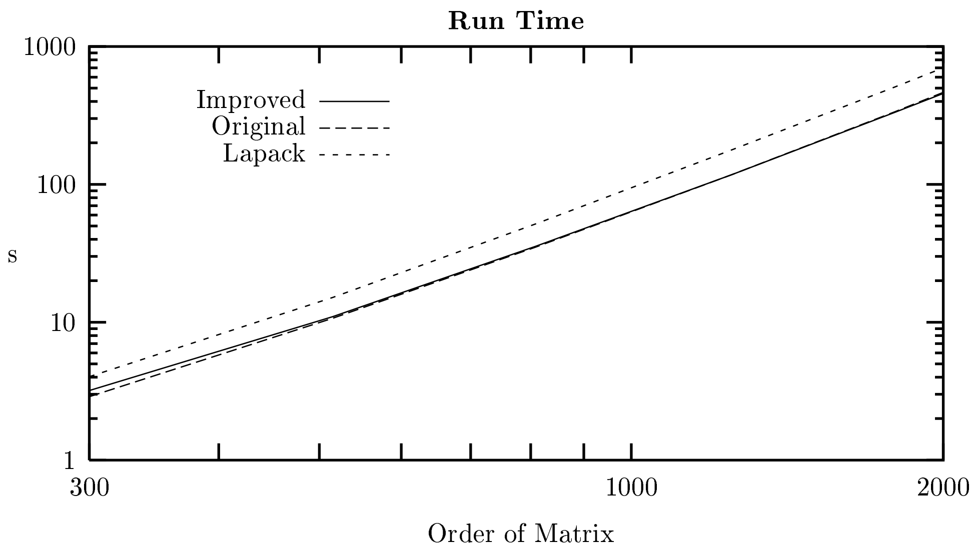

Figure 7.14: Total run time of `lapw1` on an IBM workstation.

n	Cholesky	Transf.	Eigenv.	`lapw1`	Original
283	0.1 s	0.5 s	2.3 s	3.3 s	2.8 s
515	0.9 s	2.9 s	9.7 s	14.5 s	14.0 s
809	3.5 s	10.9 s	31.5 s	48.3 s	50.7 s
1251	13.0 s	39.3 s	104.8 s	162.6 s	187.3 s

Table 7.10: Total run time of `lapw1` on an IBM workstation using a PowerPC processor.

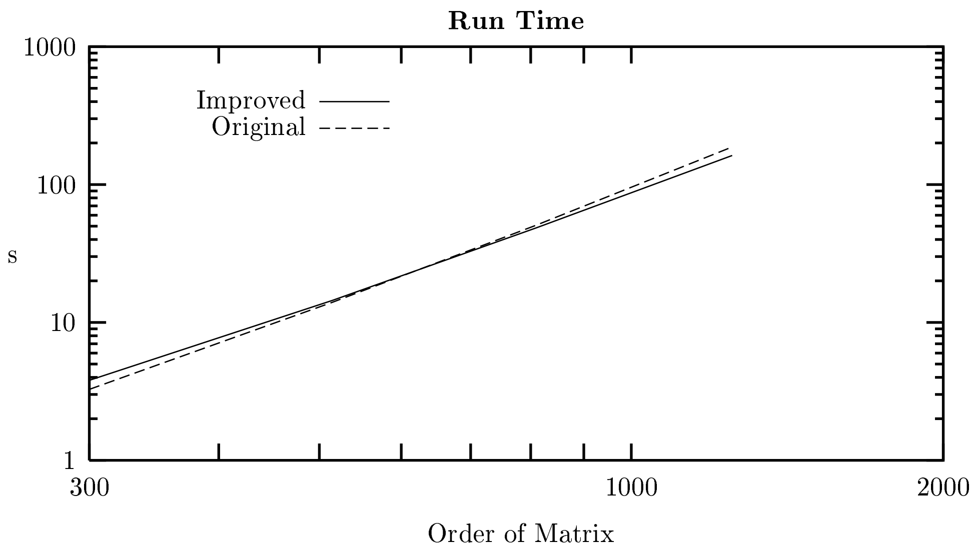

Figure 7.15: Total run time of `lapw1` on an IBM workstation using a PowerPC processor.

7.6 IBM SP2

n	Cholesky	Transf.	Eigenv.	Backtr.	`lapw1`	Original
283	0.0 s	0.1 s	0.8 s	0.0 s	1.1 s	1.0 s
515	0.2 s	0.7 s	2.7 s	0.1 s	4.0 s	4.4 s
809	0.9 s	2.9 s	8.0 s	0.4 s	12.4 s	14.4 s
1251	3.7 s	10.8 s	25.8 s	1.0 s	41.5 s	48.5 s
1795	11.2 s	31.6 s	72.1 s	2.1 s	117.0 s	136.7 s
2475	28.6 s	80.1 s	180.4 s	4.0 s	293.3 s	342.3 s

Table 7.11: Total run time of `lapw1` on an IBM SP2 thin2 node using the SP2 version of the ESSL library.

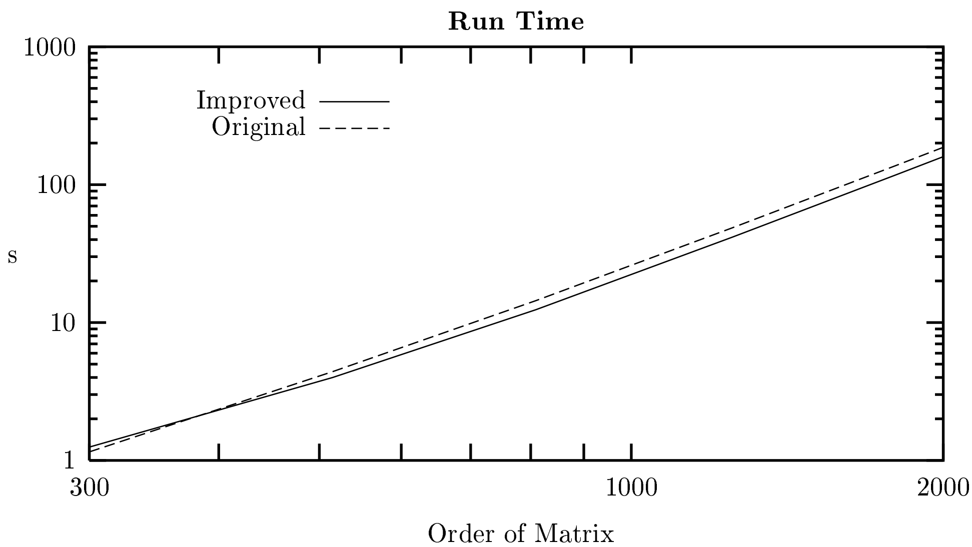

Figure 7.16: Total run time `lapw1` on an IBM SP2 thin node using the SP2 version of the ESSL library.

n	Cholesky	Transf.	Eigenv.	Backtr.	`lapw1`	Original
283	0.0 s	0.1 s	0.5 s	0.0 s	0.8 s	0.8 s
515	0.1 s	0.5 s	2.2 s	0.1 s	3.1 s	3.3 s
809	0.6 s	2.0 s	7.3 s	0.3 s	10.4 s	10.8 s
1251	2.4 s	7.2 s	24.8 s	0.9 s	35.5 s	37.0 s
1795	7.2 s	21.2 s	69.9 s	1.8 s	100.3 s	108.0 s
2457	18.6 s	53.7 s	177.6 s	3.6 s	253.7 s	285.2 s
3385	48.7 s	139.1 s	456.1 s	6.5 s	650.6 s	799.8 s

Table 7.12: Total run time of `lapw1` on an IBM SP2 with the P2SC processor running with 120 MHz using the SP2 version of the ESSL library.

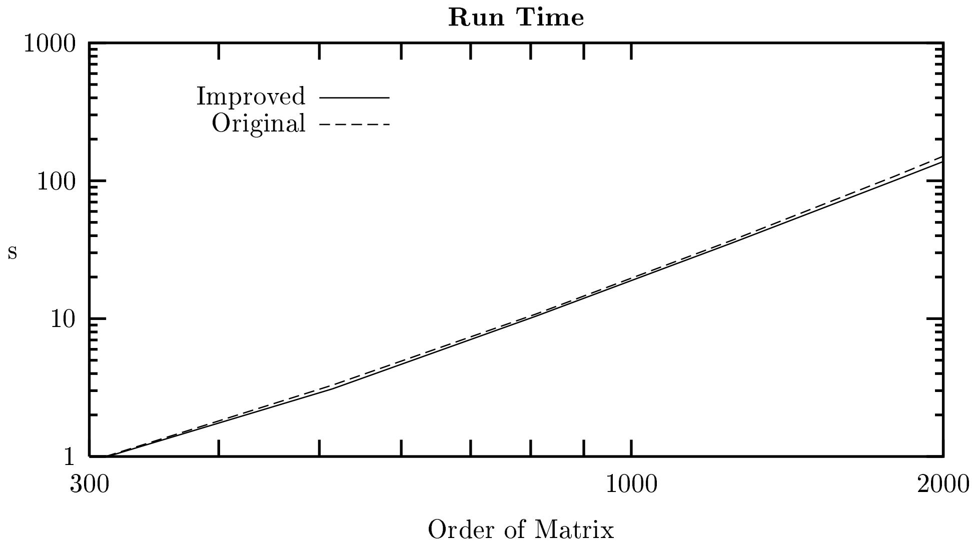

Figure 7.17: Total run time of `lapw1` on an IBM SP2 with the P2SC processor running with 120 MHz using the SP2 version of the ESSL library.

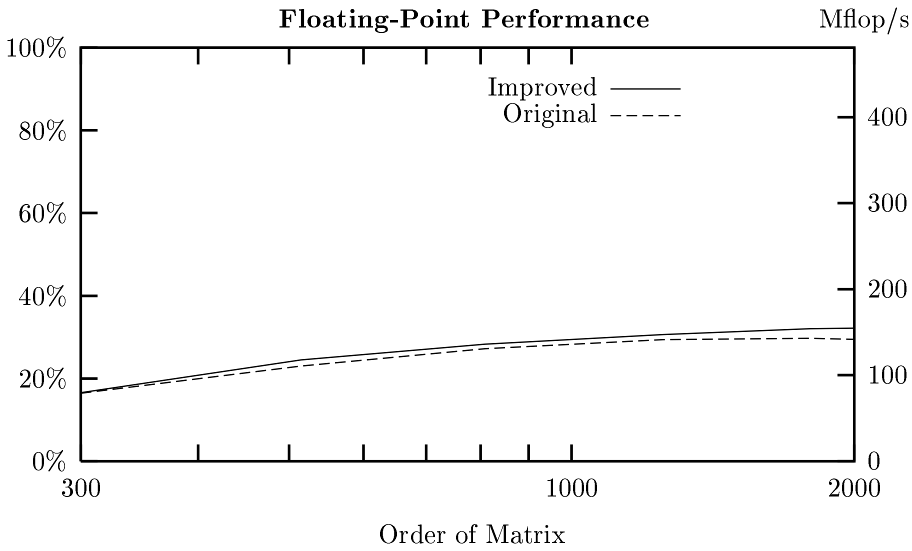

Figure 7.18: Performance of `lapw1` on an IBM SP2 with the P2SC processor running with 120 MHz using the SP2 version of the ESSL library.

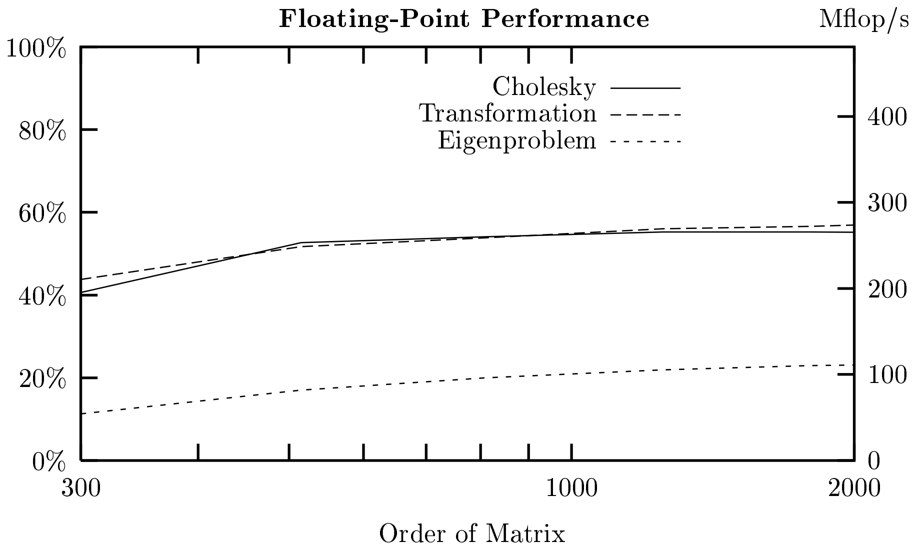

Figure 7.19: Performance of `lapw1` on an IBM SP2 with the P2SC processor running with 120 MHz using the SP2 version of the ESSL library.

n	Cholesky	Transf.	Eigenv.	Backtr.	`lapw1`	Original
283	0.0 s	0.1 s	0.8 s	0.0 s	1.0 s	0.9 s
515	0.2 s	0.7 s	2.3 s	0.1 s	3.5 s	3.8 s
809	0.9 s	2.7 s	6.7 s	0.3 s	10.8 s	12.6 s
1251	3.4 s	10.1 s	20.4 s	0.7 s	34.9 s	42.1 s
1795	10.6 s	30.0 s	56.1 s	1.5 s	98.4 s	118.2 s
2475	27.6 s	77.3 s	139.7 s	2.8 s	247.6 s	295.2 s
3385	72.7 s	210.7 s	360.3 s	5.4 s	640.4 s	757.1 s

Table 7.13: Total run time of `lapw1` on an IBM SP2 wide node using the SP2 version of the ESSL library.

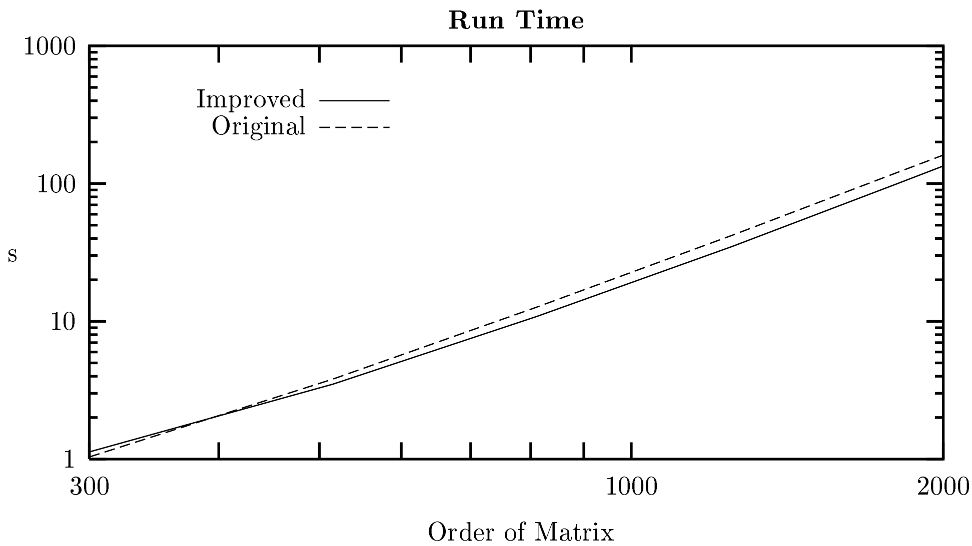

Figure 7.20: Total run time of `lapw1` on an IBM SP2 wide node using the SP2 version of the ESSL library.

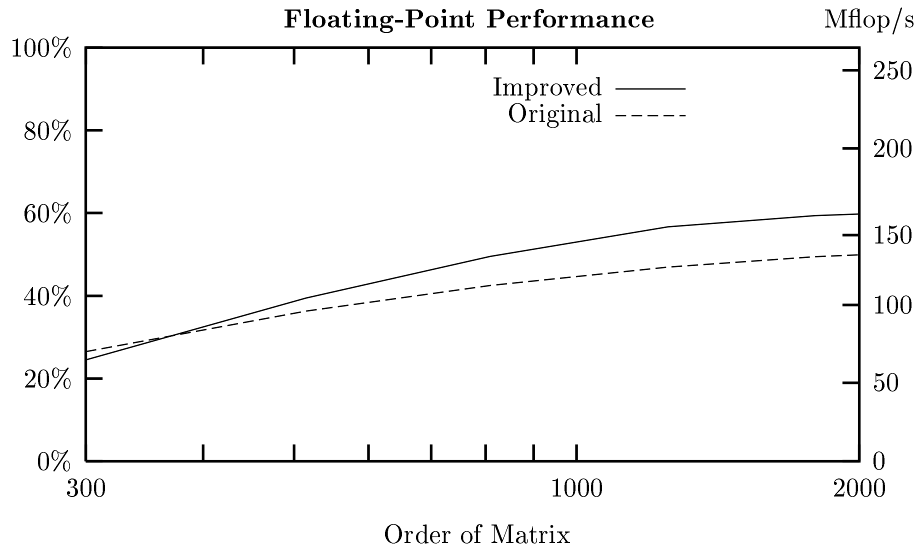

Figure 7.21: Performance of `lapw1` on an IBM SP2 wide node using the SP2 version of the ESSL library.

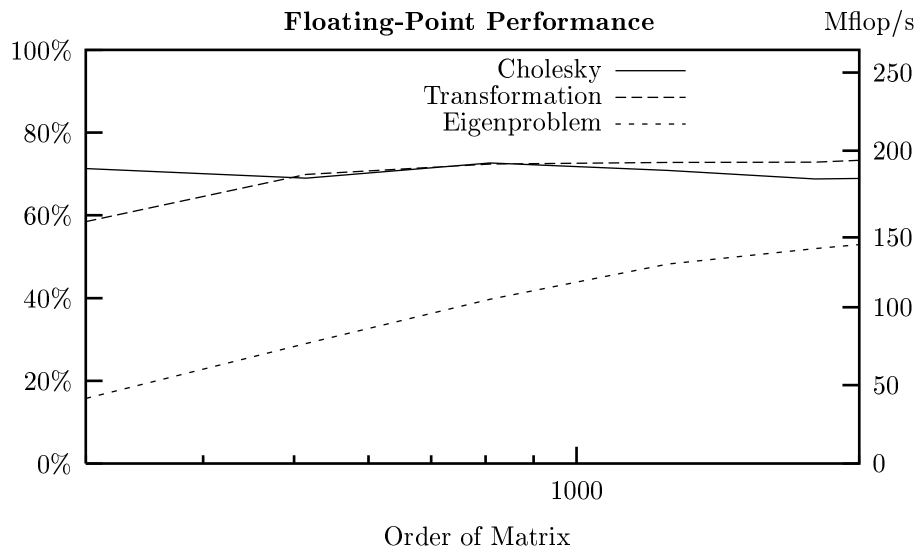

Figure 7.22: Performance of `lapw1` on an IBM SP2 wide node using the SP2 version of the ESSL library.

7.7 DEC Alpha Workstation

n	Cholesky	Transf.	Eigenv.	`lapw1`	Original
283	0.1 s	0.2 s	1.0 s	1.4 s	1.1 s
515	0.5 s	1.2 s	3.9 s	5.9 s	5.4 s
809	1.5 s	4.6 s	12.5 s	19.1 s	20.5 s
1251	6.0 s	17.3 s	56.0 s	87.3 s	118.1 s
1795	17.8 s	50.4 s	186.0 s	264.9 s	425.9 s
2457	46.9 s	127.8 s	489.6 s	680.8 s	1174.3 s

Table 7.14: Total run time of `lapw1` on a DEC Alpha workstation with a 21064 processor using the Digital DXML library for the BLAS subroutines.

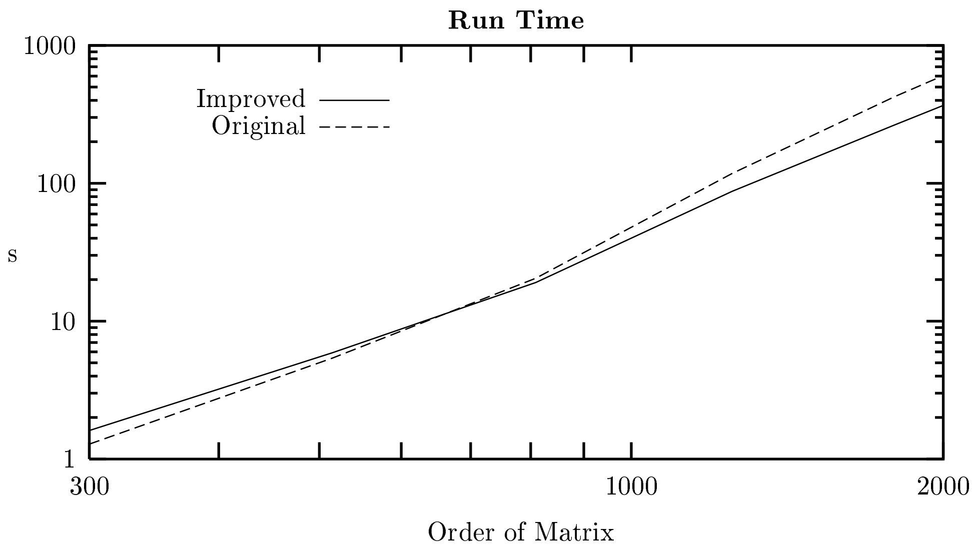

Figure 7.23: Total run time of `lapw1` on a DEC Alpha workstation with a 21064 processor using the Digital DXML library for the BLAS subroutines.

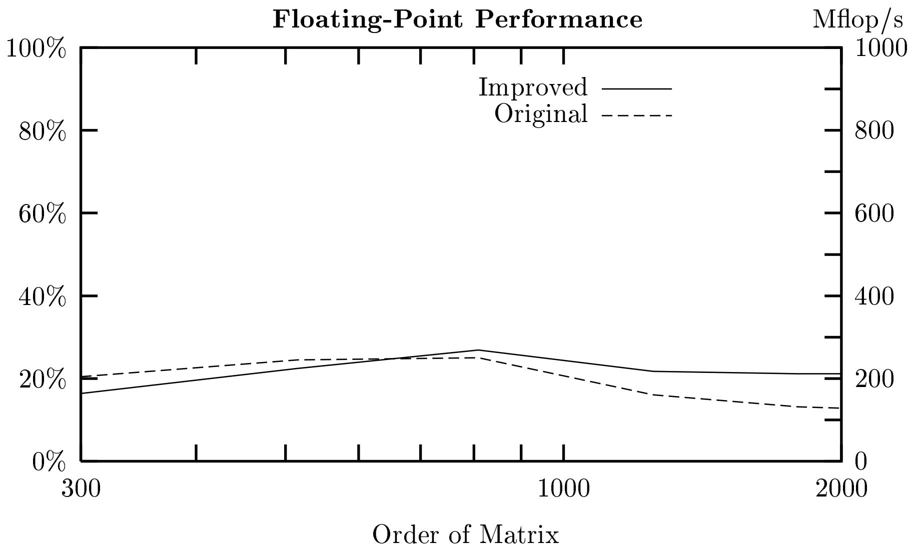

Figure 7.24: Performance of `lapw1` on a DEC Alpha workstation with a 21064 processor using the Digital DXML library for the BLAS subroutines.

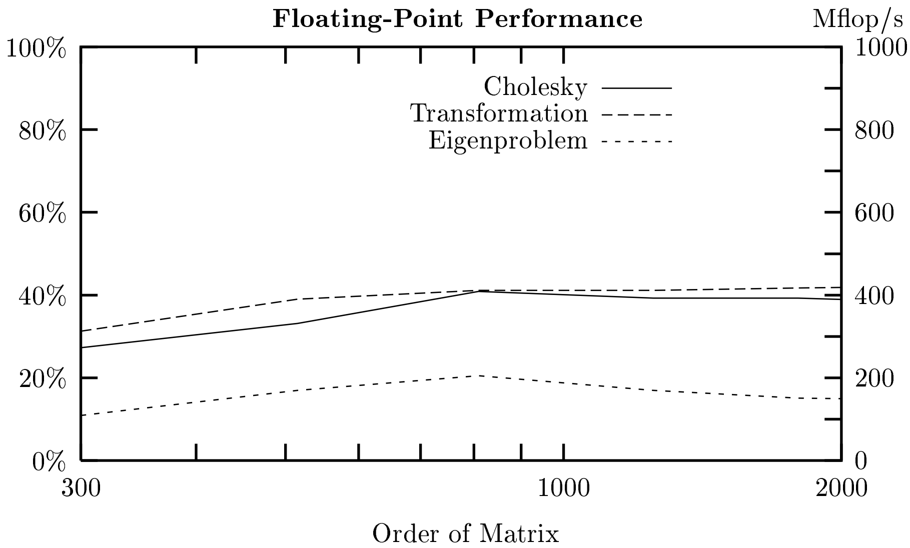

Figure 7.25: Performance of `lapw1` on a DEC Alpha workstation with a 21064 processor using the Digital DXML library for the BLAS subroutines.

n	Cholesky	Transf.	Eigenv.	`lapw1`	Original
283	0.1 s	0.1 s	0.4 s	0.7 s	0.5 s
515	0.2 s	0.7 s	1.9 s	3.1 s	2.6 s
809	0.8 s	2.4 s	6.7 s	10.5 s	9.1 s
1251	3.3 s	9.4 s	25.2 s	40.8 s	39.1 s
1795	10.7 s	29.8 s	93.4 s	143.9 s	176.2 s
2457	28.7 s	77.6 s	274.0 s	401.1 s	593.9 s

Table 7.15: Total run time of `lapw1` on a DEC Alpha workstation with a 21164 processor using the Digital DXML library for the BLAS subroutines.

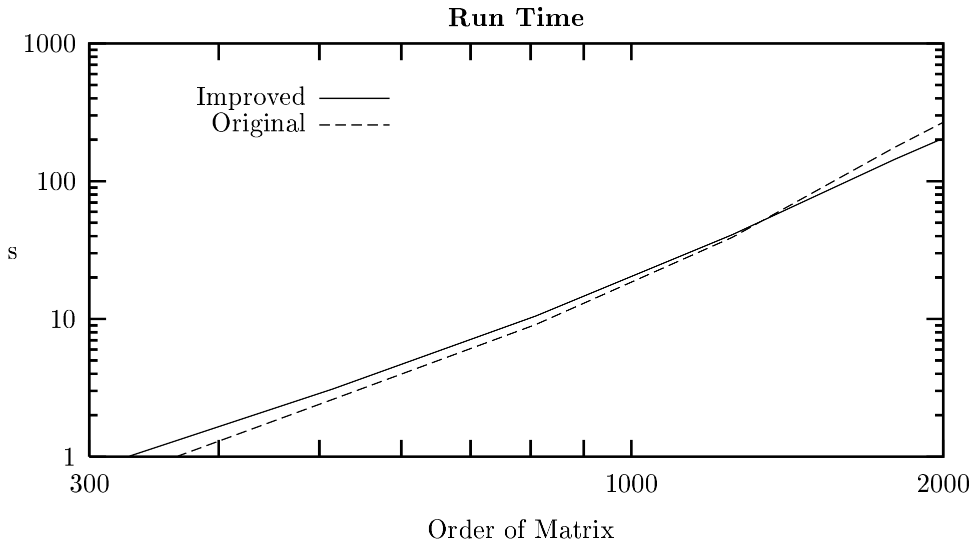

Figure 7.26: Total run time of `lapw1` on a DEC Alpha workstation with a 21164 processor using the Digital DXML library for the BLAS subroutines.

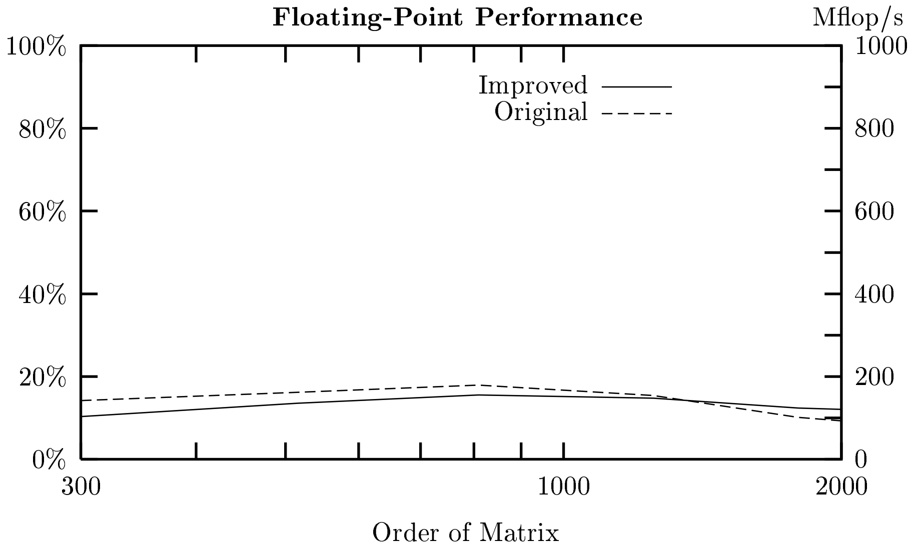

Figure 7.27: Performance of `lapw1` on a DEC Alpha workstation with a 21164 processor using the Digital DXML library for the BLAS subroutines.

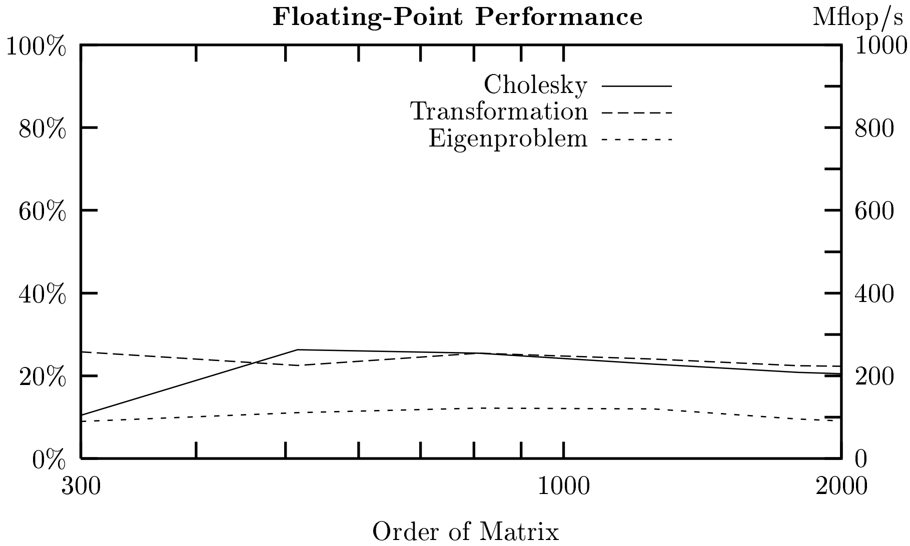

Figure 7.28: Performance of `lapw1` on a DEC Alpha workstation with a 21164 processor using the Digital DXML library for the BLAS subroutines.

7.8 Pentium PC

n	Cholesky	Transf.	Eigenv.	Backtr.	`lapw1`	Original
283	0.3 s	1.1 s	2.6 s	0.3 s	4.5 s	3.5 s
515	1.8 s	6.7 s	12.9 s	1.1 s	22.7 s	19.5 s
809	7.0 s	25.3 s	49.7 s	3.2 s	85.4 s	74.5 s
1251	25.7 s	91.8 s	186.9 s	8.3 s	312.9 s	270.0 s
1795	77.8 s	275.0 s	567.7 s	17.7 s	938.3 s	785.1 s
2475	197.5 s	694.5 s	1507.7 s	33.8 s	2433.6 s	1998.8 s

Table 7.16: Total run time of `lapw1` on a Pentium based PC using Linux as operating system.

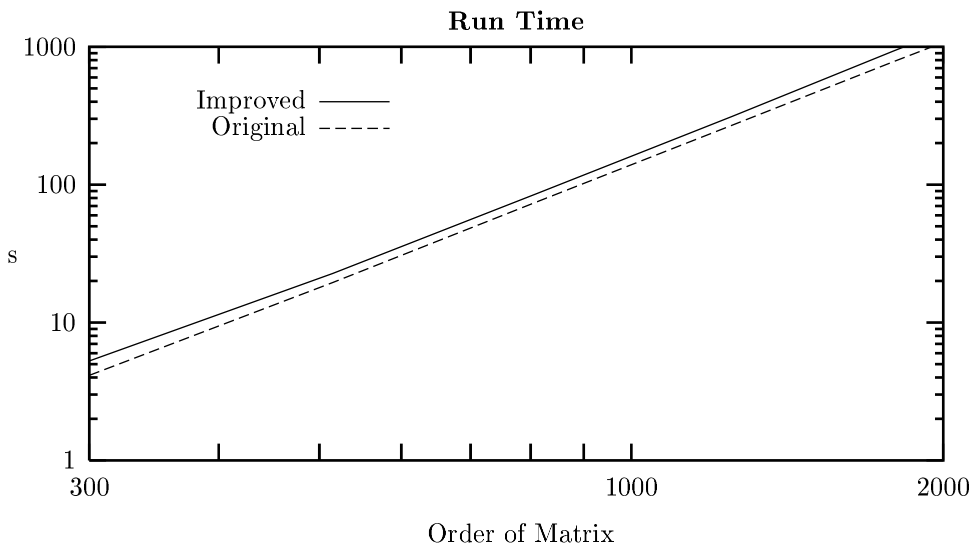

Figure 7.29: Total run time of `lapw1` on a Pentium based PC using Linux as operating system.

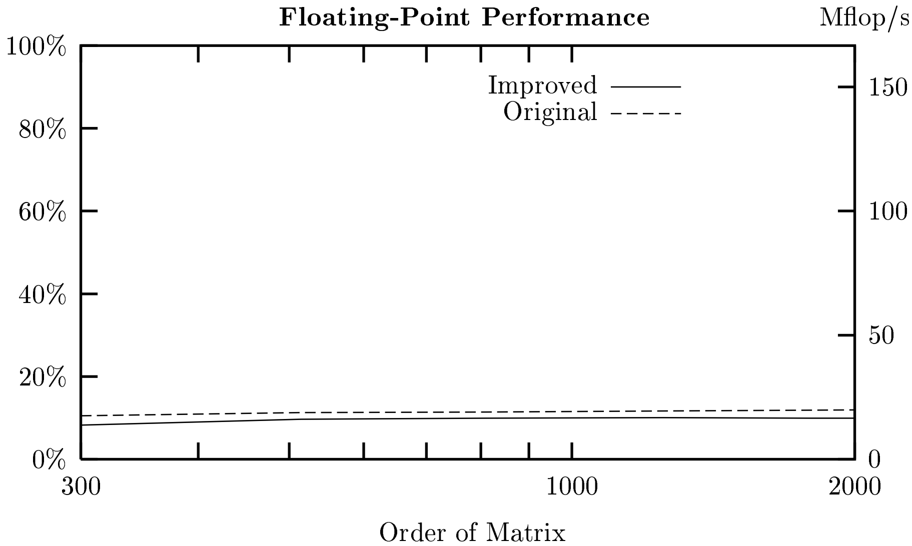

Figure 7.30: Performance of `lapw1` on a Pentium based PC using Linux as operating system.

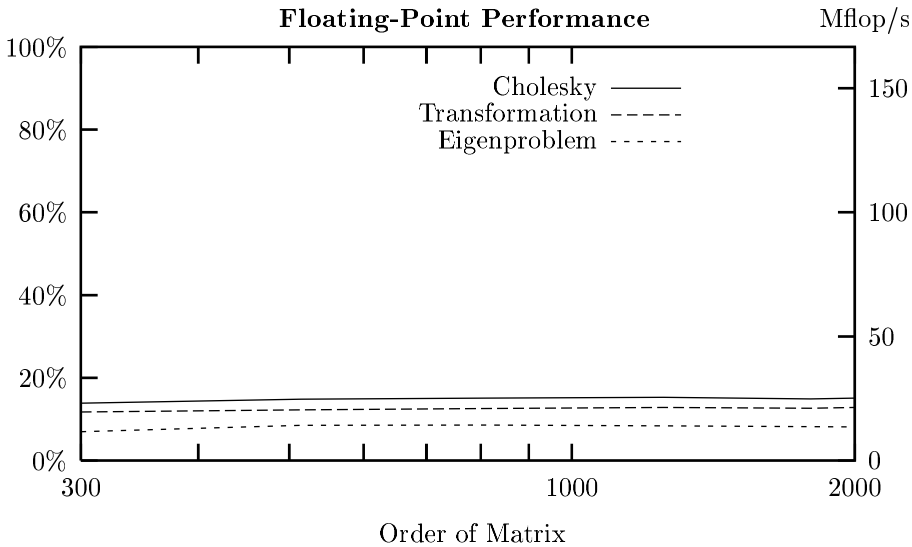

Figure 7.31: Performance of `lapw1` on a Pentium based PC using Linux as operating system.

n	Cholesky	Transf.	Eigenv.	Backtr.	`lapw1`	Original
283	0.1 s	0.3 s	0.9 s	0.1 s	1.6 s	1.3 s
515	0.6 s	2.2 s	4.8 s	0.7 s	8.4 s	8.4 s
809	2.4 s	8.3 s	18.1 s	1.8 s	30.8 s	35.1 s
1251	8.9 s	30.7 s	67.1 s	4.4 s	111.2 s	132.7 s
1795	26.1 s	89.0 s	214.0 s	9.2 s	338.5 s	392.5 s
2475	68.4 s	240.0 s	608.5 s	17.5 s	934.6 s	1002.7 s

Table 7.17: Total run time of `lapw1` on a Pentium II based PC using Linux as operating system.

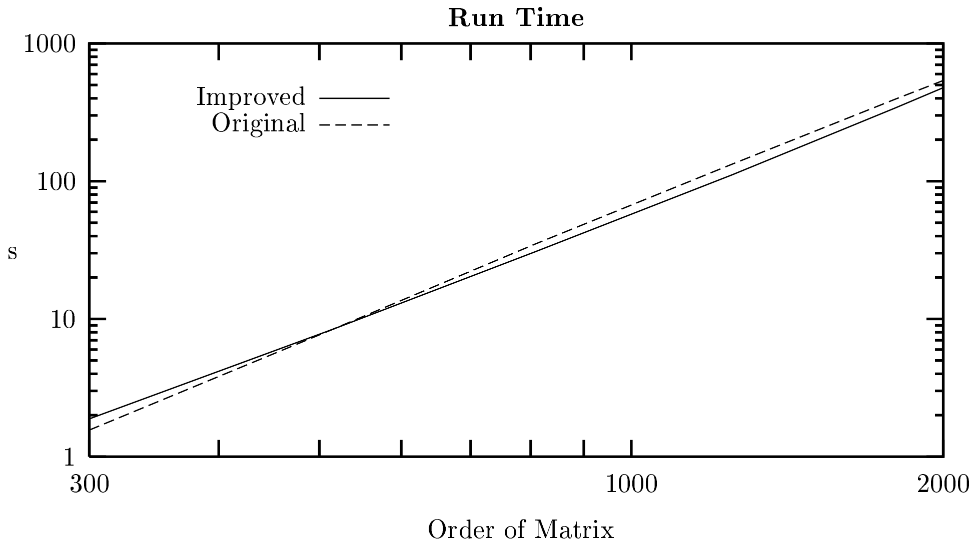

Figure 7.32: Total run time of `lapw1` on a Pentium II based PC using Linux as operating system.

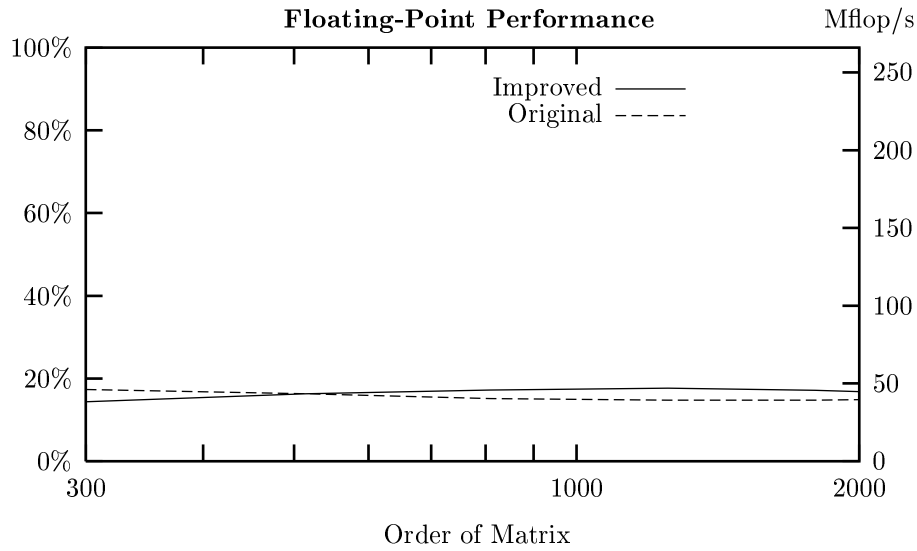

Figure 7.33: Performance of `lapw1` on a Pentium II based PC using Linux as operating system.

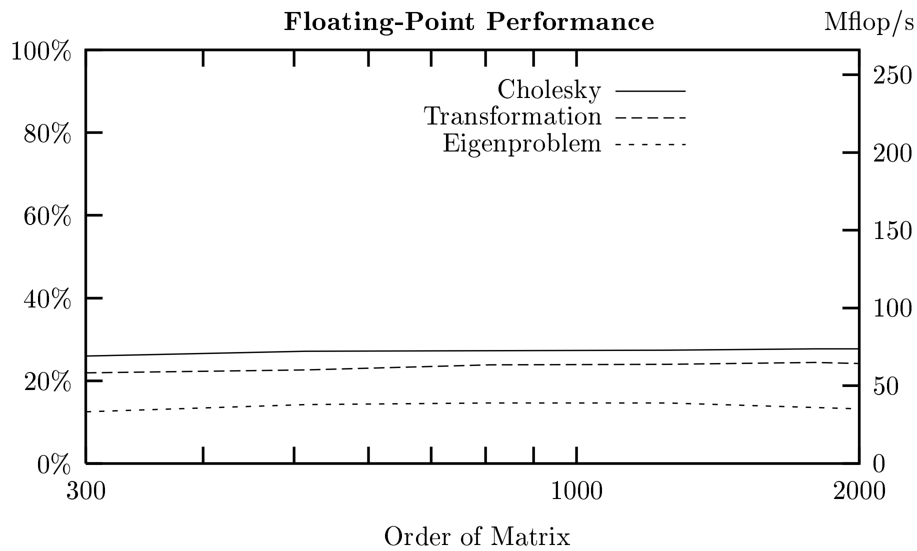

Figure 7.34: Performance of `lapw1` on a Pentium II based PC using Linux as operating system.

7.9 NEC Vectorcomputer SX-4

n	Cholesky	Transf.	Eigenv.	Backtr.	`lapw1`	Original
283	0.0 s	0.0 s	0.3 s	0.0 s	0.4 s	0.5 s
515	0.0 s	0.1 s	0.8 s	0.0 s	1.1 s	1.5 s
809	0.1 s	0.4 s	1.6 s	0.0 s	2.3 s	3.6 s
1251	0.4 s	1.4 s	3.8 s	0.1 s	5.8 s	9.2 s
1795	1.2 s	3.9 s	8.3 s	0.2 s	13.8 s	21.1 s
2457	3.0 s	9.6 s	17.7 s	0.4 s	30.7 s	44.9 s
3385	7.7 s	24.3 s	40.2 s	0.7 s	73.0 s	100.5 s
4399	16.7 s	52.3 s	81.7 s	1.2 s	152.2 s	199.3 s
5615	34.6 s	107.3 s	161.3 s	2.1 s	305.6 s	

Table 7.18: Total run time of `lapw1` on a NEC SX4.

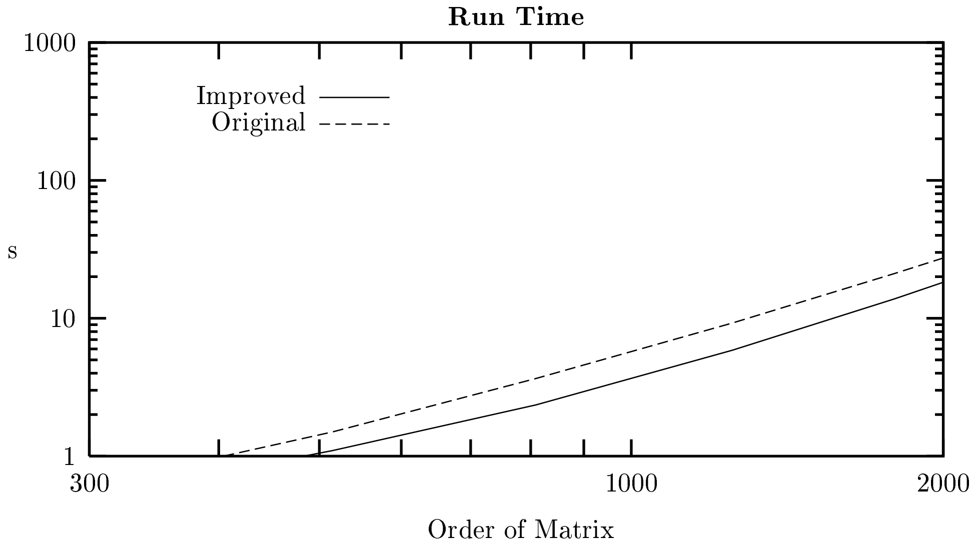

Figure 7.35: Total run time of `lapw1` on a NEC SX4.

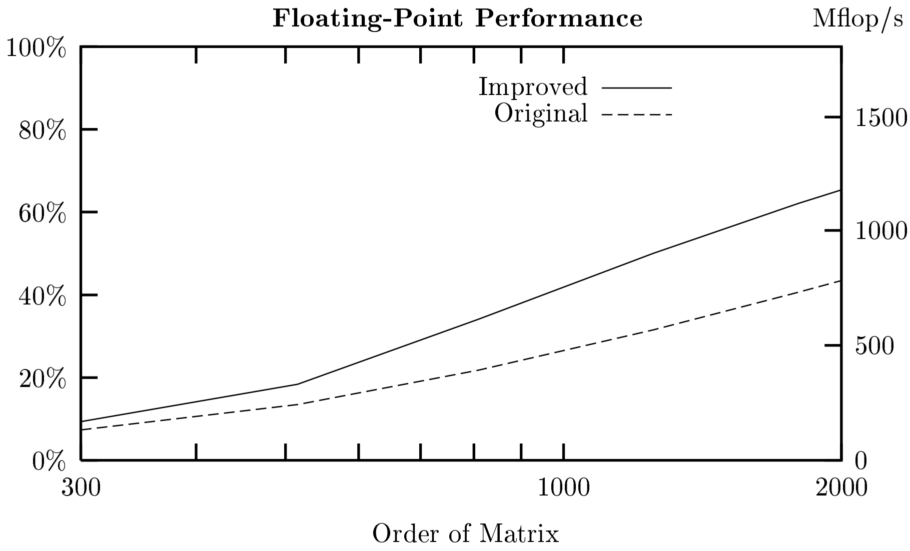

Figure 7.36: Performance of `lapw1` on a NEC SX4.

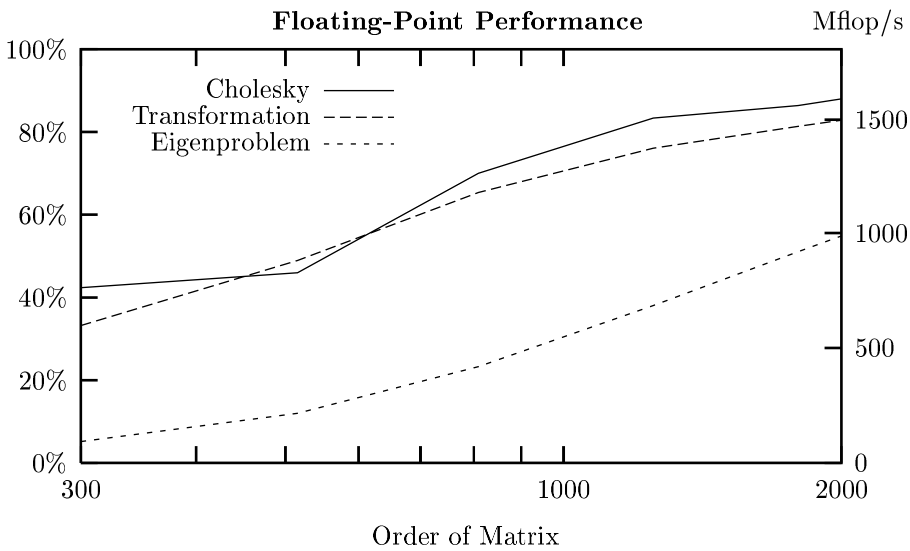

Figure 7.37: Performance of `lapw1` on a NEC SX4.

7.10 Summary

The figures in this section show a comparison of the computers which have been used in this report. Figure 7.38 shows that the floating-point performance of most current workstations lies within a small band. Some computers in this band show higher performance for small matrix sizes and lower performance for larger matrix sizes, whereas others behave the opposite way. The reason for this behavior is that computers with high clock rate (for example, workstations using SGI or DEC processors) show high performance with small matrix sizes and computers with high memory throughput (for example, workstations using IBM or HP processors) show high performance with larger matrix sizes.

Computers with Intel processors (PCs) show lower performance than most of the current workstations, whereas the vector computer NEC SX-4 is significantly faster. It can be seen that the vector computer does not scale with n^3 with matrices up to 2000×2000.

Table 7.19 shows the floating-point performance values achieved with the old and the new `lapw1` program. Speedup numbers are given in the last column.

The performance degradation of some computers when increasing the matrix size can be seen in Figures 7.39 and 7.40. This is due to cache effects, which could not be removed completely due to the bad cache behavior of parts of the tridiagonalization routine. Other computers show increasing floating-point performance with increasing matrix size. The Alpha 21164 clone's floating-point performance is relatively constant for all matrix sizes.

The new routines show improved performance for larger matrix sizes on all computers tested. On some computers (for example, on the SGI Power Challenge using an R8000 processor) the new routines were slightly slower than the old ones, but only for smaller matrix sizes. This performance degradation for small problems is of no practical relevance.

Table 7.20 shows that high clock rates or high peak performances do not always lead to high floating-point performance values. An IBM workstation running with 66 MHz (with a peak performance of 264 Mflop/s) and an efficiency of 70.5 % can easily outperform a DEC Alpha workstation running with 500 MHz (with a peak performance of 1 Gflop/s).

The tridiagonalization routine which consumes up to 50 % of the total time of the new `lapw1` program is the only major routine which was not fully blocked. The performance of computers with high clock rate and relatively low memory throughput could be improved by more than 70 % when the tridiagonalization routine would use only Level 3 operations (see Table 7.21).

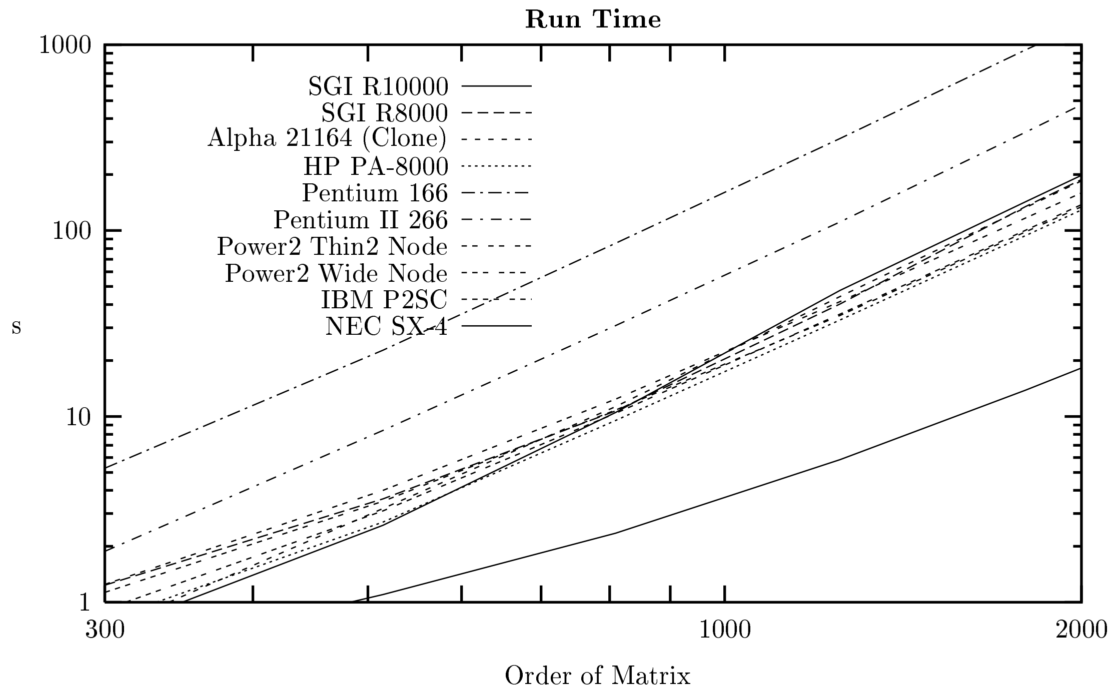

Figure 7.38: Total run time of the improved `lapw1` on various computer systems.

Host	Cholesky	Transf.	Eigenv.	`lapw1`	Original	Speedup
SGI R10000	195.3	212.7	77.9	124.2	46.1	169 %
SGI R8000	169.4	180.7	85.1	126.3	83.3	52 %
Alpha 21064	107.7	118.6	41.3	67.6	39.1	72 %
Alpha 21164	172.2	191.1	72.2	112.3	75.8	48 %
Alpha Clone	242.3	259.8	76.7	130.4	92.5	41 %
HP PA 8000	228.2	239.4	137.9	195.3	137.1	42 %
IBM SP2 Thin2	176.3	189.1	112.1	156.9	134.5	17 %
IBM SP2 Wide	182.7	196.0	144.7	185.9	156.0	19 %
IBM SP2 P2SC	270.5	282.1	113.8	181.5	161.5	12 %
IBM Power	56.1	57.8	43.9	55.4	53.6	3 %
Pentium 166	25.0	21.4	13.1	18.6	22.5	-18 %
Pentium II	73.8	63.2	33.2	49.3	45.9	7 %
NEC SX-4	1676.2	1566.8	1140.2	1497.0	1024.1	46 %

Table 7.19: Floating-point performance in Mflop/s for various processors. Computed from time measurements with matrix size $n = 2457$. Estimated number of operations: Cholesky: $n^3/3$, Transformation: n^3, Eigenvalue routines: $4n^3/3$, Sum: $8n^3/3$.

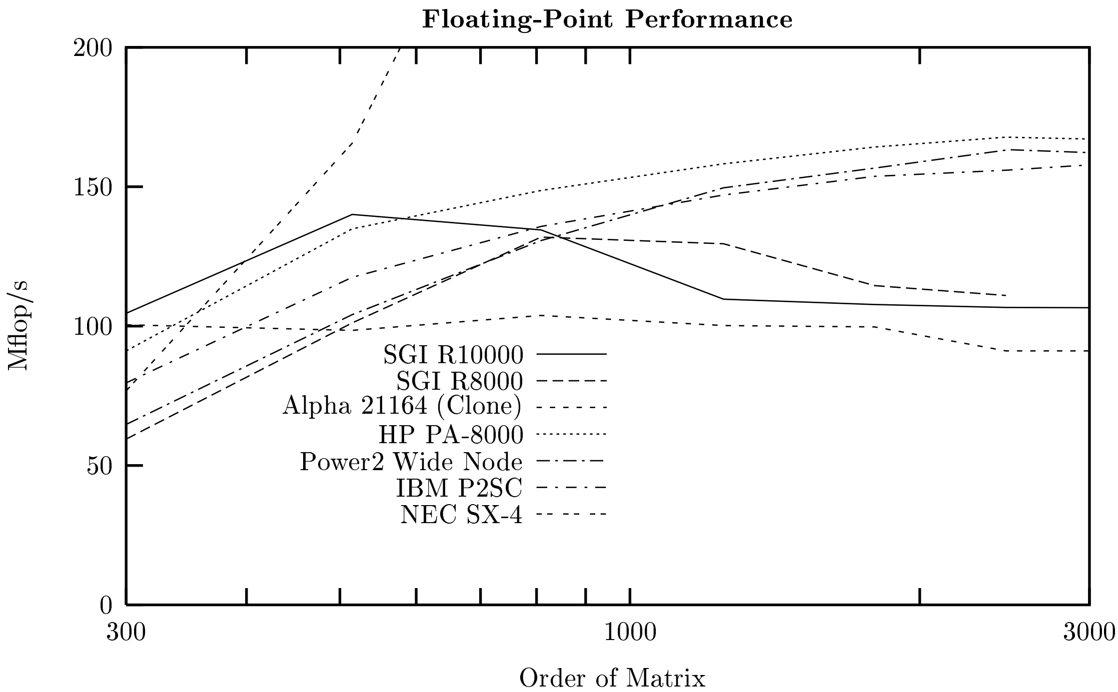

Figure 7.39: Floating-point performance of the new version of `lapw1` on various computer systems.

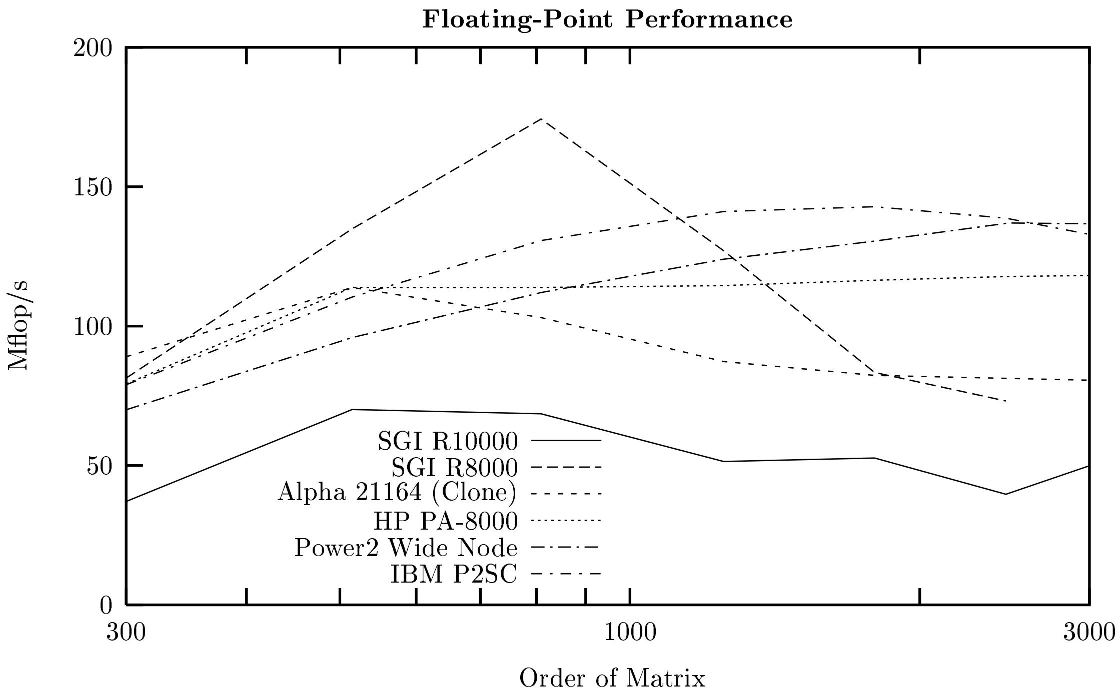

Figure 7.40: Floating-point performance of the original program `lapw1` on various computer systems.

Host	Clock	Peak	Eff.	Cholesky	Transf.	Eigenv.
SGI R10000	195	390	31.8 %	7.0 %	19.2 %	70.0 %
SGI R8000	90	360	35.0 %	5.7 %	23.0 %	65.2 %
Alpha 21064	275	275	24.6 %	6.9 %	18.8 %	71.9 %
Alpha 21146	433	866	13.0 %	7.2 %	19.3 %	68.3 %
Alpha Clone	500	1000	13.1 %	5.9 %	16.5 %	74.6 %
HP PA 8000	180	720	27.1 %	9.4 %	26.9 %	62.2 %
IBM SP2 Thin2	66	264	59.4 %	9.8 %	27.3 %	61.5 %
IBM SP2 Wide	66	264	70.5 %	11.1 %	31.2 %	56.4 %
IBM SP2 P2SC	120	480	37.8 %	7.4 %	21.2 %	70.0 %
IBM Power	44	88	63.0 %	10.9 %	31.6 %	55.4 %
NEC SX-4	112.5	1800	83.1 %	9.8 %	31.3 %	57.6 %

Table 7.20: Weight of the main subroutines of `lapw1` in percent of execution time. Clock rates are in MHz, Peak Performance in Mflop/s, Efficiency is the sustained performance (of `lapw1`) as percentage of the peak performance. The three last columns give the relative amount of time in the routines Cholesky, Transformation, and Eigenvalues. (Percentage values taken from measurements with matrix size 2475.)

Host	`lapw1`	Cholesky	Potential
SGI R10000	124.2	195.3	57 %
SGI R8000	126.3	169.4	34 %
Alpha 21064	67.6	107.7	59 %
Alpha 21164	112.3	172.2	53 %
Alpha Clone	130.4	242.3	86 %
HP PA 8000	195.3	228.2	17 %
IBM SP2 Thin2	156.9	176.3	12 %
IBM SP2 Wide	185.9	182.7	12 %
IBM SP2 P2SC	181.5	270.5	70 %
IBM Power	55.4	56.1	15 %
Pentium 166	18.6	25.0	53 %
Pentium II 266	49.3	73.8	70 %
NEC SX-4	1497.0	1676.2	27 %

Table 7.21: Potential speedup for the case that the remaining matrix-vector product in `lapw1` is converted to a matrix-matrix product. The column `lapw1` is the performance of the whole program, including the matrix-vector call, the column Cholesky is the performance of the `cholesky` routine, which is a (matrix-matrix oriented) subroutine of `lapw1`. (Both columns in Mflop/s and copied from Table 7.19.) The last column is the speedup which can be gained by using only Level 3 routines.

Chapter 8

Performance Results for Complex Data

Different processors have very different performance figures for varying subroutines. The IBM computers did already work well with the original code versions. Others (HP, SGI, Alpha) were improved by varying factors. The processors with higher clock rates (all except the IBM Power, Power2 and the R8000 of the SGI Power Challenge) which rely on caching suffer on significant performance degradation for the eigenvalue routines. This is due to the matrix-vector product which could not be removed in this routine.

The sustained performance is for all computers higher than in the real case (compare Tables 7.19 and 8.13). This is due to lower memory bandwidth requirements. Every complex multiply-add statement is equal to four real multiply-adds. The amount of data required, however, is only twice as much. The result is half the memory bandwidth requirements in the complex case.

These lower bandwidth requirements result in a better (relative) performance of Level 2 Blas. This can be seen in the percentages of the eigenvalue routine in Tables 7.20 and 8.14. The potential speedup for the full blocking of the tridiagonalization routine is thus also lower in the complex case (compare Tables 7.21 and 8.15).

8.1 SGI Power Challenge XL

n	Cholesky	Transf.	Eigenv.	Backtr.	`lapw1c`	Original
195	0.0 s	0.1 s	0.3 s	0.0 s	0.6 s	0.5 s
339	0.2 s	0.7 s	1.2 s	0.1 s	2.4 s	2.2 s
463	0.6 s	1.9 s	2.7 s	0.2 s	5.6 s	6.9 s
767	2.9 s	8.7 s	15.0 s	1.9 s	28.7 s	59.7 s
1053	7.7 s	22.5 s	40.7 s	3.7 s	74.7 s	176.5 s
1507	23.1 s	66.5 s	118.7 s	7.6 s	215.1 s	540.1 s
1973	52.5 s	149.3 s	262.5 s	13.6 s	478.1 s	1234.3 s

Table 8.1: Total run time of `lapw1c` on an SGI Power Challenge XL.

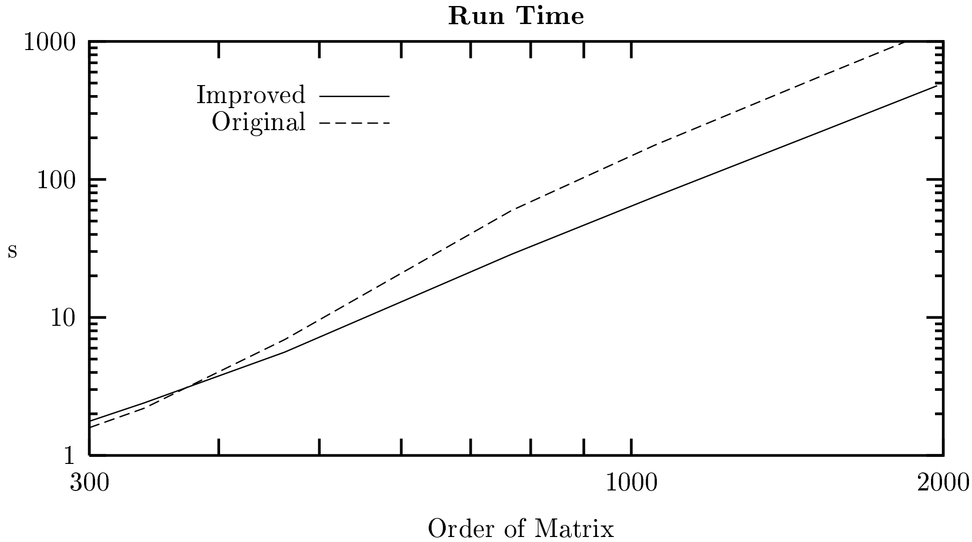

Figure 8.1: Total run time of `lapw1c` on an SGI Power Challenge XL.

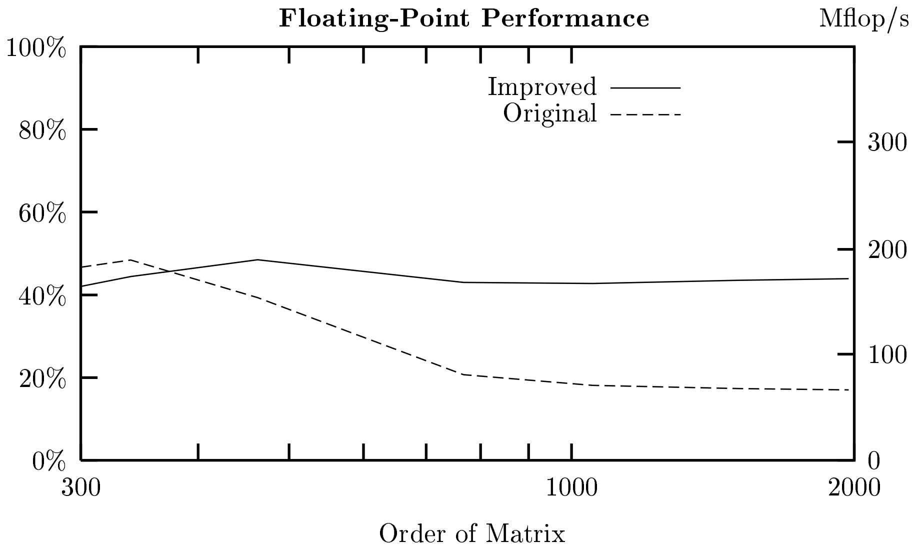

Figure 8.2: Performance of `lapw1c` on an SGI Power Challenge XL.

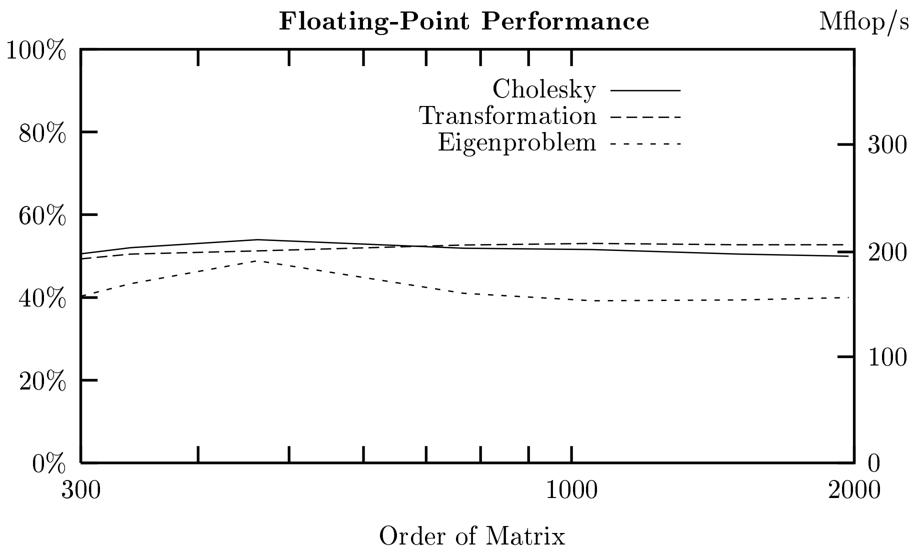

Figure 8.3: Performance of `lapw1c` on an SGI Power Challenge XL.

8.2 SGI Power Challenge

n	Cholesky	Transf.	Eigenv.	Backtr.	`lapw1c`	Original
195	0.0 s	0.2 s	0.5 s	0.0 s	0.8 s	1.0 s
339	0.3 s	0.9 s	1.4 s	0.1 s	2.8 s	4.0 s
463	0.7 s	2.2 s	2.9 s	0.2 s	6.2 s	9.3 s
767	3.1 s	9.8 s	11.4 s	0.9 s	25.4 s	46.0 s
1053	8.4 s	24.9 s	36.7 s	2.9 s	73.1 s	145.5 s
1507	24.7 s	72.8 s	121.6 s	6.7 s	226.0 s	483.9 s
1973	55.9 s	162.7 s	282.6 s	11.6 s	513.0 s	1124.4 s

Table 8.2: Total run time of `lapw1c` on an SGI Power Challenge.

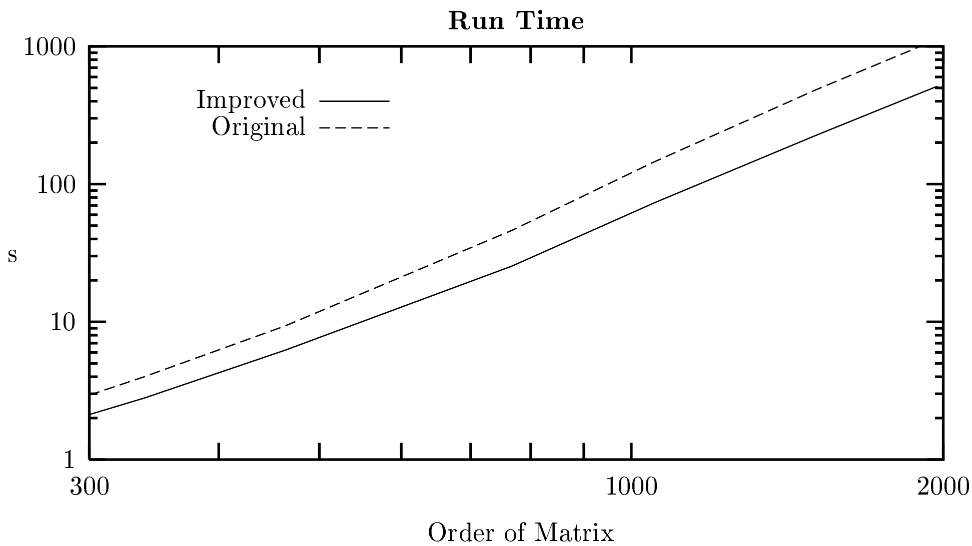

Figure 8.4: Total run time of `lapw1c` on an SGI Power Challenge XL.

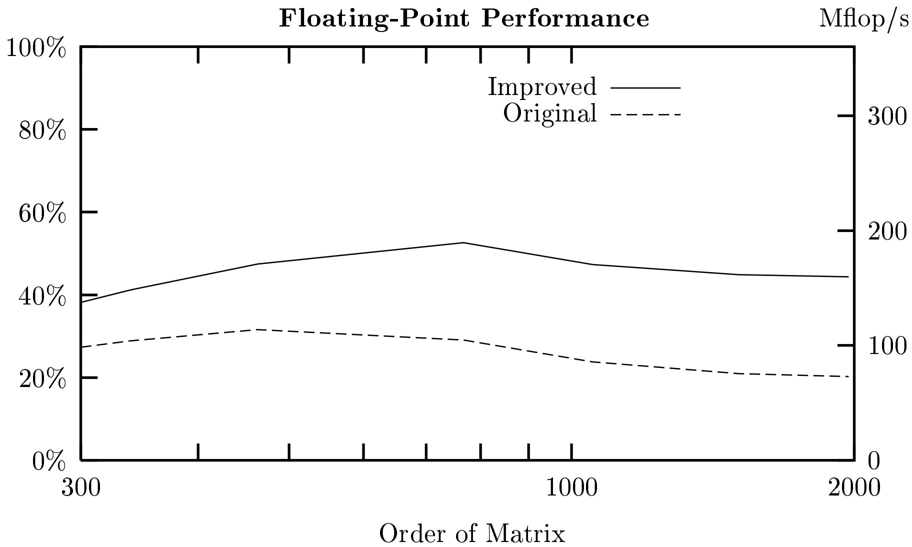

Figure 8.5: Performance of `lapw1c` on an SGI Power Challenge XL.

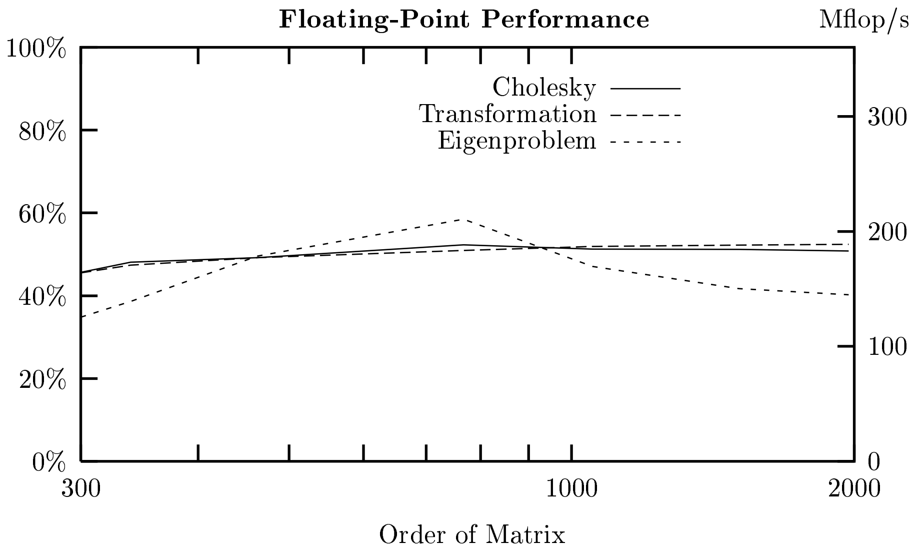

Figure 8.6: Performance of `lapw1c` on an SGI Power Challenge XL.

8.3 Alpha Workstation (Clone)

n	Cholesky	Transf.	Eigenv.	Backtr.	`lapw1c`	Original
195	0.0 s	0.1 s	0.3 s	0.0 s	0.5 s	0.5 s
339	0.1 s	0.6 s	1.5 s	0.1 s	2.5 s	2.9 s
463	0.4 s	1.5 s	3.8 s	0.3 s	6.2 s	7.8 s
767	2.1 s	7.3 s	19.3 s	1.2 s	30.1 s	42.8 s
1053	5.7 s	18.6 s	52.0 s	2.5 s	78.9 s	118.0 s
1507	16.6 s	54.2 s	158.3 s	5.2 s	234.5 s	358.4 s
1973	38.3 s	120.7 s	376.1 s	8.9 s	543.2 s	819.9 s

Table 8.3: Total run time of `lapw1c` on an Alpha workstation.

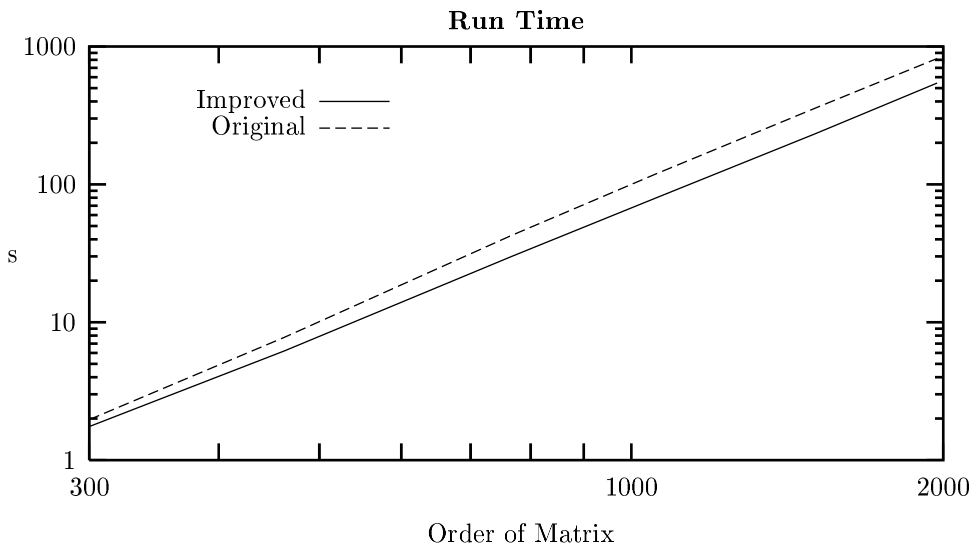

Figure 8.7: Total run time of `lapw1c` on an Alpha workstation.

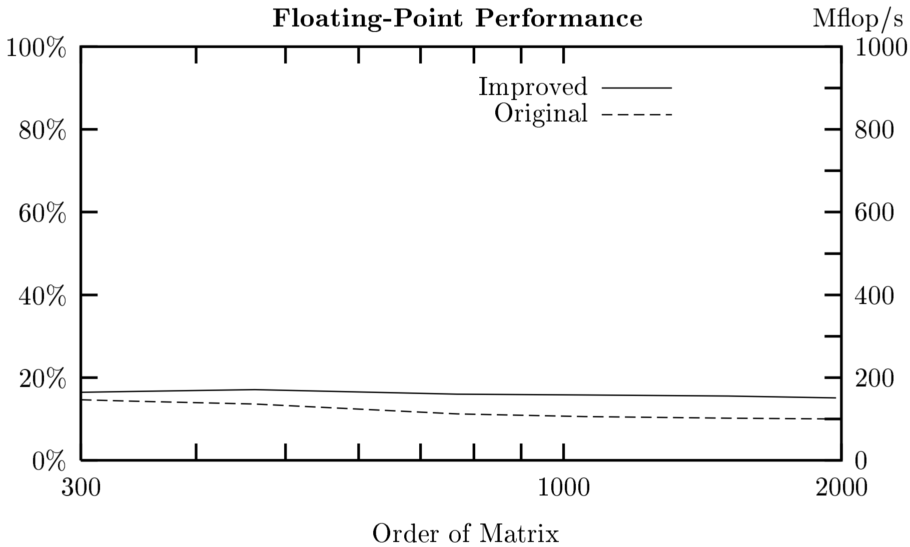

Figure 8.8: Performance of `lapw1c` on an Alpha workstation.

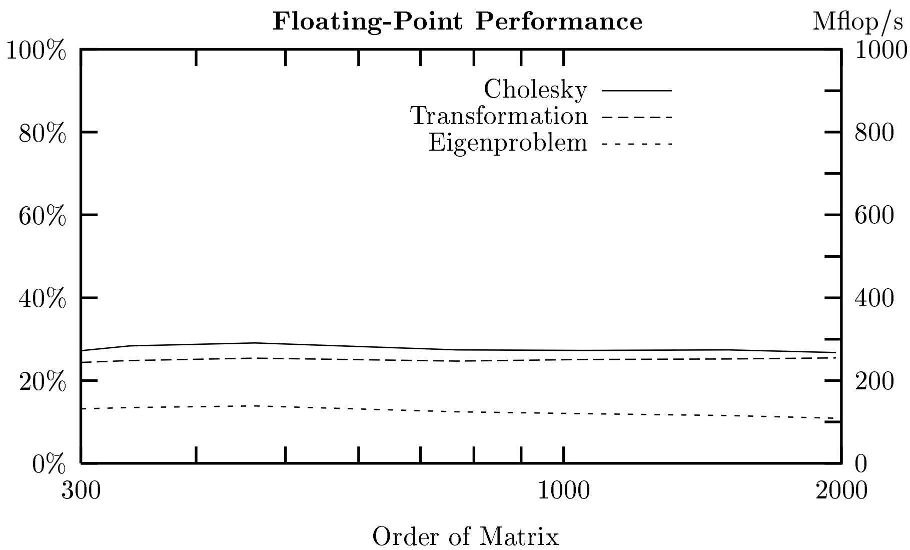

Figure 8.9: Performance of `lapw1c` on an Alpha workstation.

8.4 DEC Alpha Workstation

n	Cholesky	Transf.	Eigenv.	Backtr.	`lapw1c`	Original
195	0.0 s	0.3 s	0.8 s	0.0 s	1.3 s	1.1 s
339	0.4 s	1.5 s	3.4 s	0.2 s	5.6 s	5.3 s
463	0.9 s	3.8 s	8.4 s	0.4 s	13.8 s	13.0 s
767	4.8 s	18.3 s	40.6 s	2.8 s	66.6 s	85.7 s
1053	13.7 s	49.1 s	113.1 s	6.5 s	182.6 s	269.4 s
1507	38.8 s	138.5 s	348.8 s	14.6 s	540.9 s	892.0 s
1973	88.4 s	312.6 s	807.8 s	25.4 s	1234.4 s	2080.6 s

Table 8.4: Total run time of `lapw1c` on a Digital Alpha workstation with a 21064 processor.

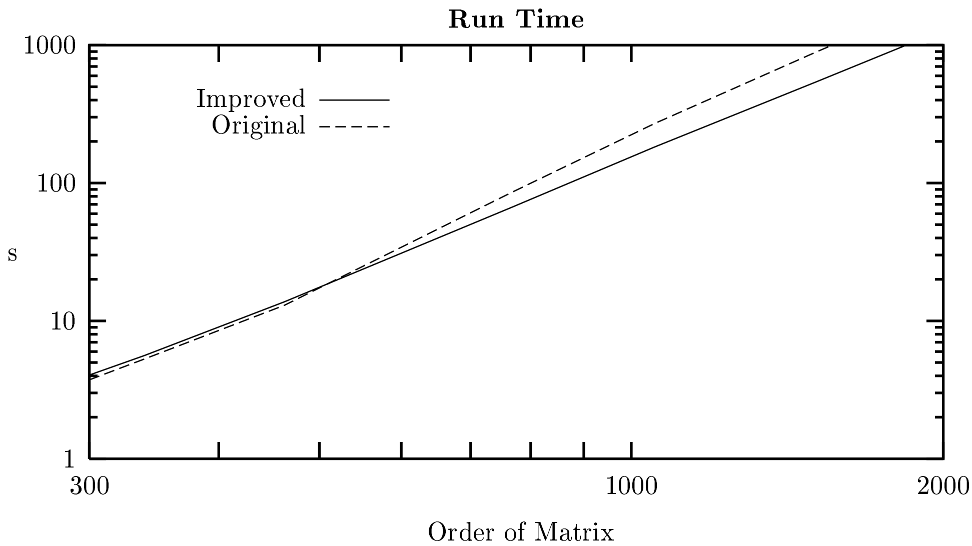

Figure 8.10: Total run time of `lapw1c` on a Digital Alpha workstation with a 21064 processor.

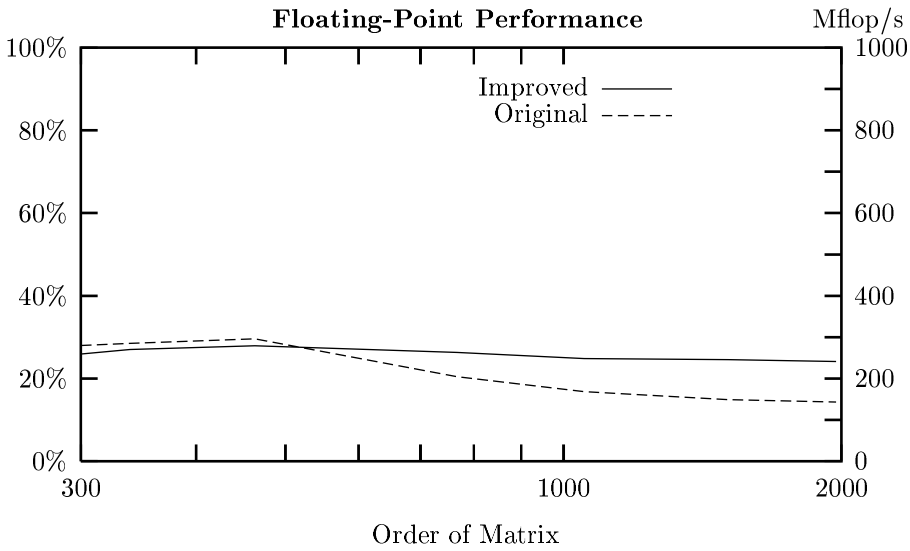

Figure 8.11: Performance of `lapw1c` on a Digital Alpha workstation with a 21064 processor.

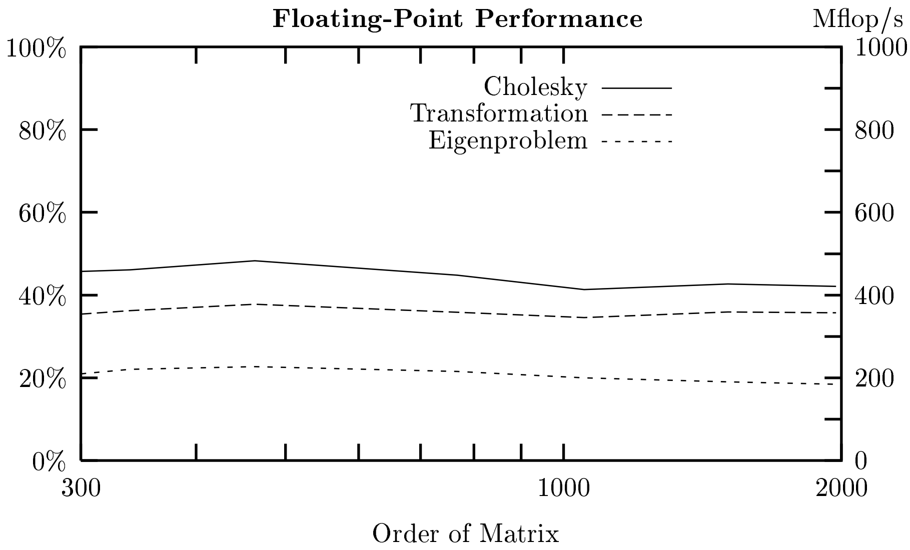

Figure 8.12: Performance of `lapw1c` on a Digital Alpha workstation with a 21064 processor.

n	Cholesky	Transf.	Eigenv.	Backtr.	`lapw1c`	Original
195	0.1 s	0.1 s	0.3 s	0.0 s	0.6 s	0.5 s
339	0.2 s	0.6 s	1.5 s	0.1 s	2.6 s	2.4 s
463	0.5 s	1.6 s	3.5 s	0.1 s	6.0 s	5.8 s
767	2.3 s	7.1 s	15.2 s	0.7 s	25.5 s	25.9 s
1053	6.4 s	19.3 s	44.1 s	3.1 s	73.1 s	111.7 s
1507	20.1 s	57.9 s	156.2 s	10.2 s	244.6 s	490.3 s
1973	45.9 s	129.2 s	371.7 s	17.7 s	564.7 s	1270.0 s

Table 8.5: Total run time of `lapw1c` on a Digital Alpha workstation with a 21164 processor.

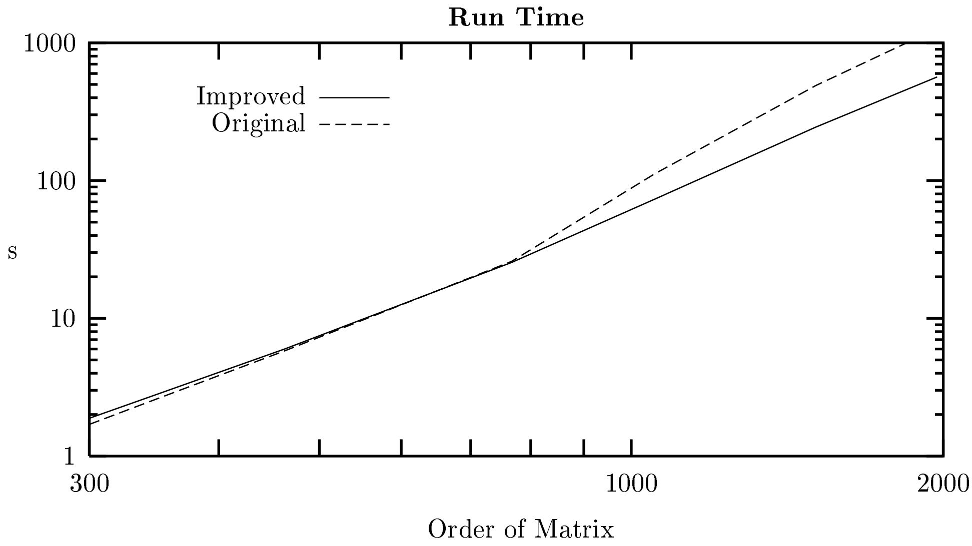

Figure 8.13: Total run time of `lapw1c` on a Digital Alpha workstation with a 21164 processor.

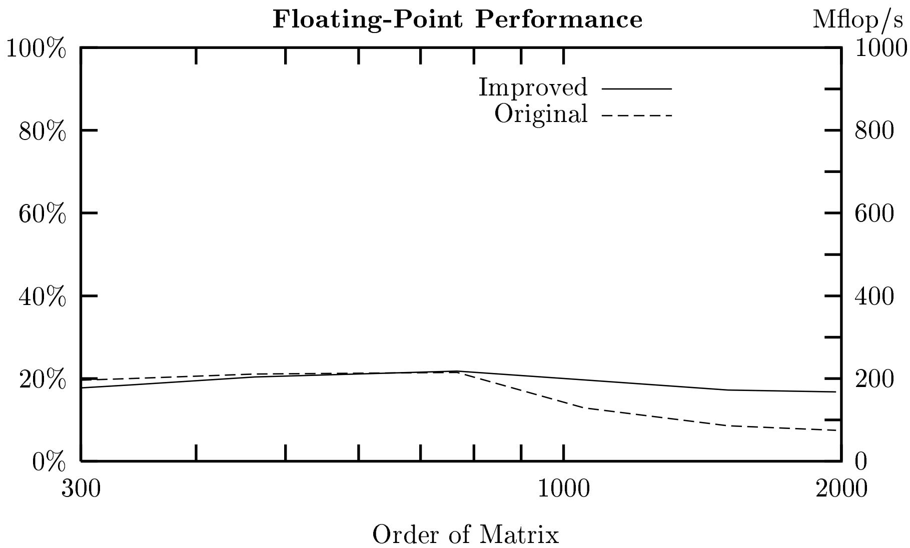

Figure 8.14: Performance of `lapw1c` on a Digital Alpha workstation with a 21164 processor.

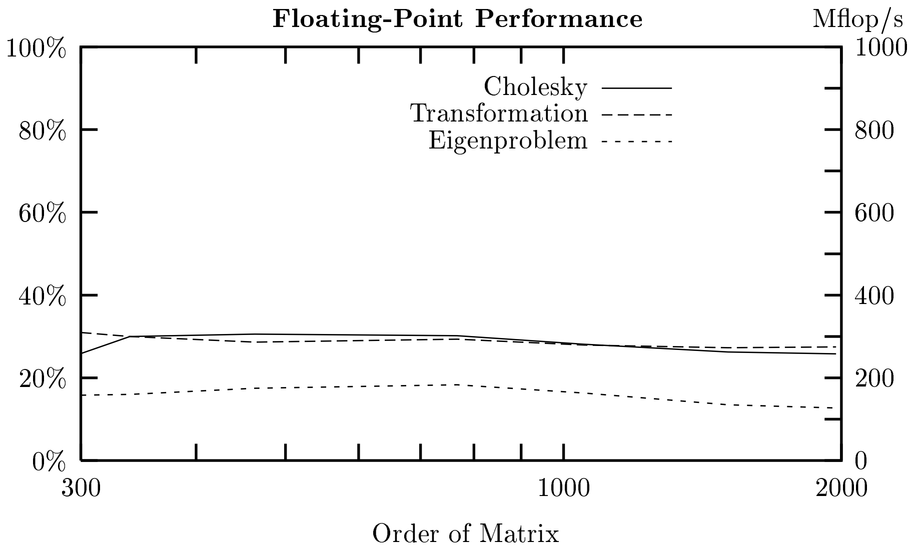

Figure 8.15: Performance of `lapw1c` on a Digital Alpha workstation with a 21164 processor.

8.5 HP Workstation K460

n	Cholesky	Transf.	Eigenv.	Backtr.	`lapw1c`	Original
195	0.0 s	0.1 s	0.4 s	0.0 s	0.7 s	0.7 s
339	0.1 s	0.4 s	1.6 s	0.1 s	2.4 s	3.1 s
463	0.4 s	1.1 s	4.0 s	0.4 s	6.1 s	7.7 s
767	1.7 s	4.6 s	18.6 s	1.4 s	26.5 s	33.7 s
1053	4.2 s	11.6 s	46.9 s	2.7 s	65.6 s	85.1 s
1507	12.1 s	33.4 s	134.8 s	5.6 s	186.0 s	243.7 s
1973	26.5 s	76.1 s	303.1 s	9.6 s	415.5 s	541.2 s

Table 8.6: Total run time of `lapw1c` on an HP workstation .

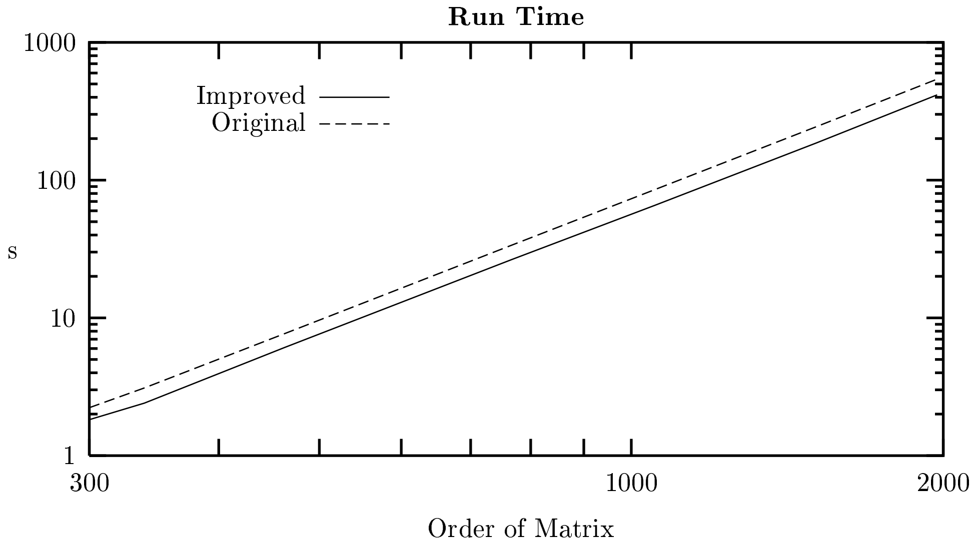

Figure 8.16: Total run time of `lapw1c` on an HP workstation.

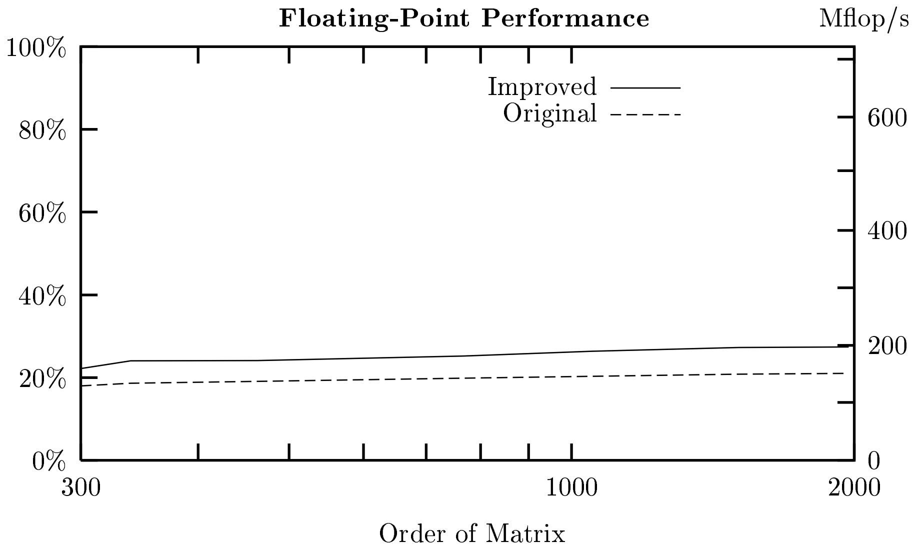

Figure 8.17: Performance of `lapw1c` on an HP workstation.

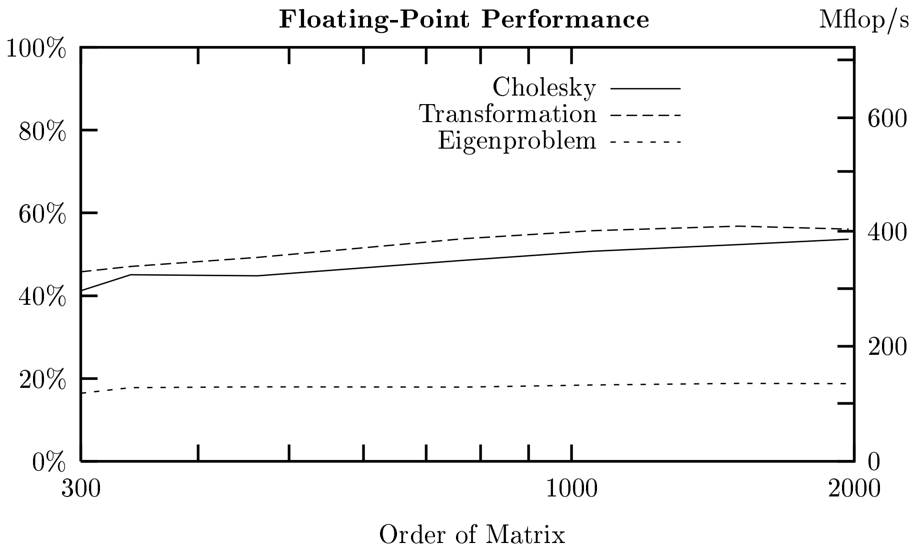

Figure 8.18: Performance of `lapw1c` on an HP workstation.

8.6 IBM SP2

n	Cholesky	Transf.	Eigenv.	Backtr.	`lapw1c`	Original
195	0.0 s	0.1 s	0.6 s	0.0 s	1.0 s	0.8 s
339	0.3 s	0.9 s	2.3 s	0.1 s	3.7 s	3.3 s
463	0.7 s	2.2 s	4.8 s	0.2 s	8.0 s	7.5 s
767	2.9 s	9.0 s	18.0 s	0.6 s	30.7 s	31.2 s
1053	7.7 s	22.9 s	43.9 s	1.2 s	75.9 s	75.6 s
1507	22.6 s	65.2 s	122.0 s	2.4 s	212.3 s	219.3 s
1973	49.4 s	144.2 s	269.7 s	4.1 s	467.5 s	479.2 s

Table 8.7: Total run time of `lapw1c` on an IBM SP2 using a P2SC processor with 120 MHz.

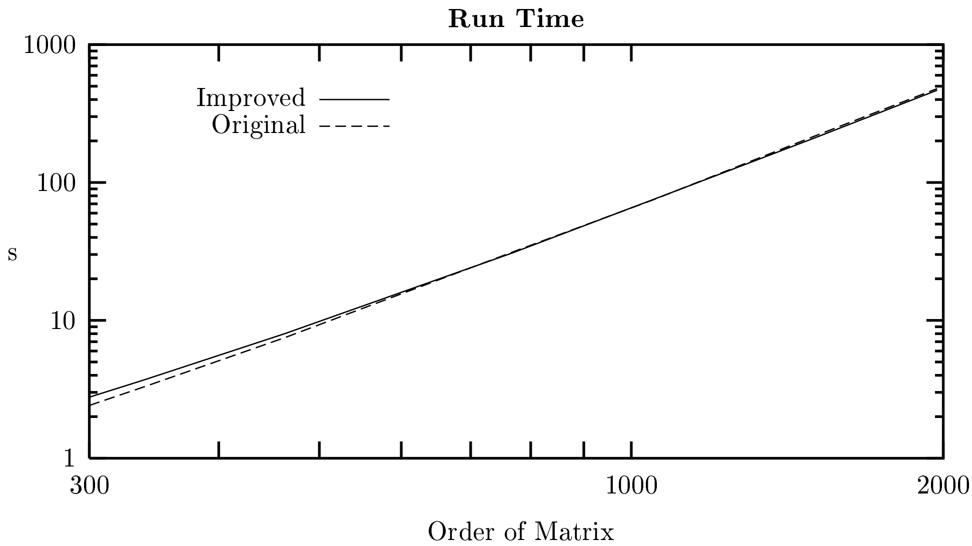

Figure 8.19: Total time of `lapw1c` on an IBM SP2 using a P2SC processor with 120 MHz.

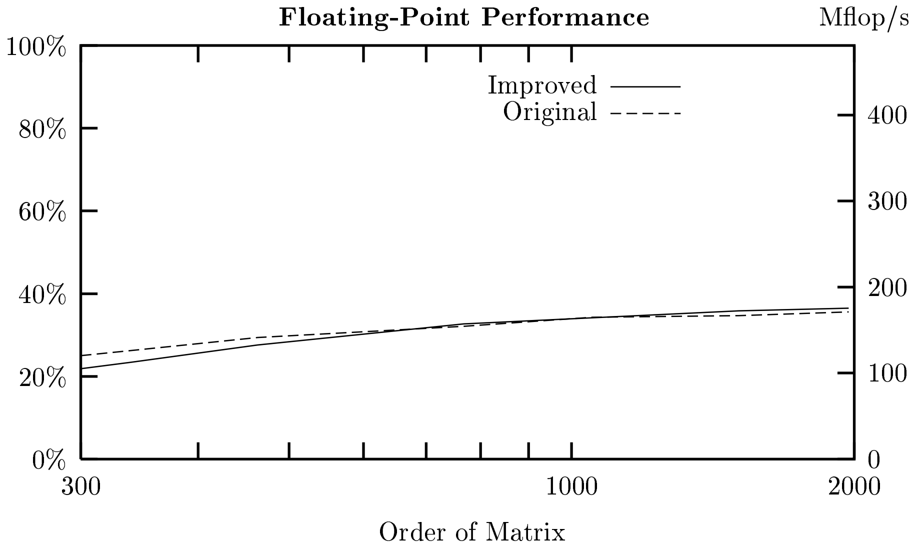

Figure 8.20: Performance of `lapw1c` on an IBM SP2 using a P2SC processor with 120 MHz.

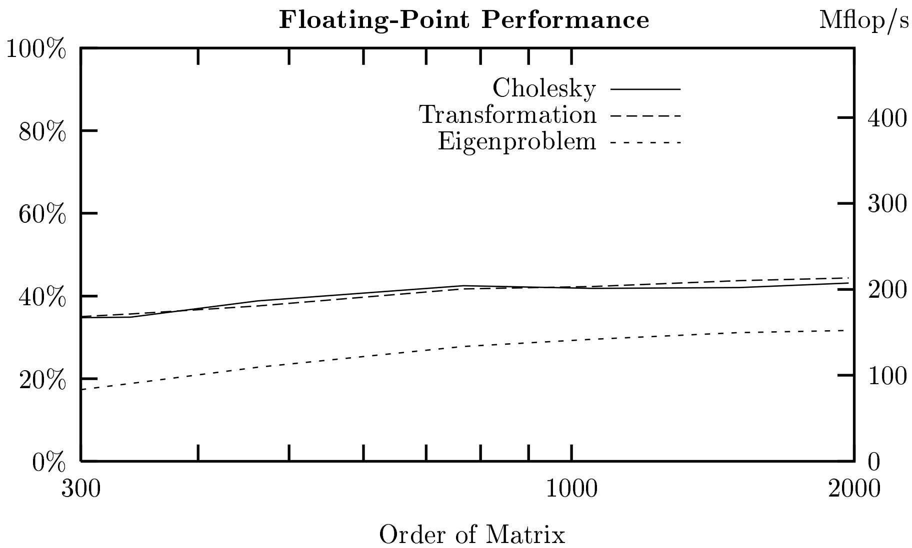

Figure 8.21: Performance of `lapw1c` on an IBM SP2 using a P2SC processor with 120 MHz.

n	Cholesky	Transf.	Eigenv.	Backtr.	`lapw1c`	Original
195	0.0 s	0.1 s	0.6 s	0.0 s	0.9 s	0.8 s
339	0.2 s	0.8 s	2.0 s	0.1 s	3.3 s	3.0 s
463	0.7 s	2.2 s	4.2 s	0.1 s	7.3 s	6.8 s
767	3.0 s	9.1 s	16.1 s	0.5 s	28.9 s	27.6 s
1053	7.6 s	22.7 s	38.9 s	0.9 s	70.2 s	68.4 s
1507	21.9 s	64.7 s	109.1 s	1.9 s	197.8 s	193.0 s
1973	48.4 s	142.8 s	238.4 s	3.3 s	433.0 s	425.5 s

Table 8.8: Total run time of `lapw1c` on an IBM SP2 using a Power2 processor (wide node).

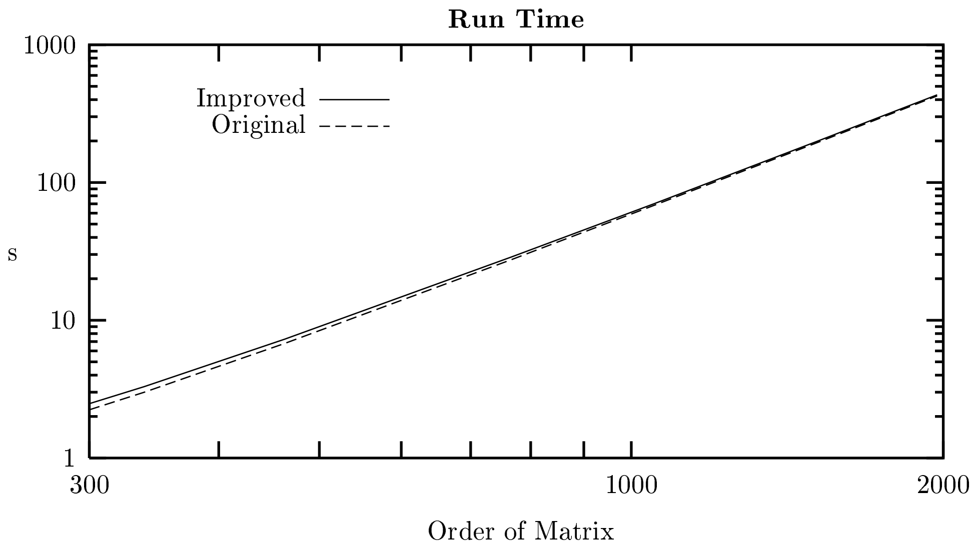

Figure 8.22: Total run time of `lapw1c` on an IBM SP2 using a Power2 processor (wide node).

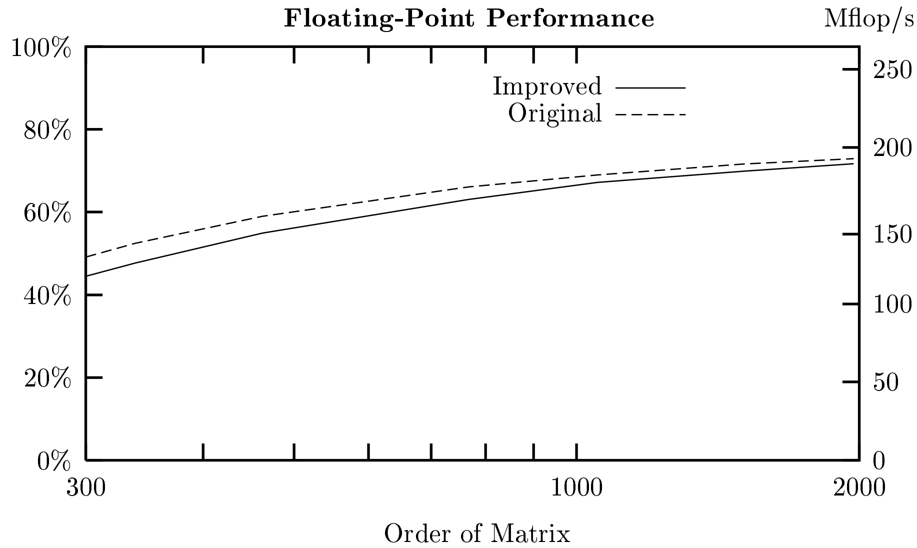

Figure 8.23: Performance of `lapw1c` on an IBM SP2 using a Power2 processor (wide node).

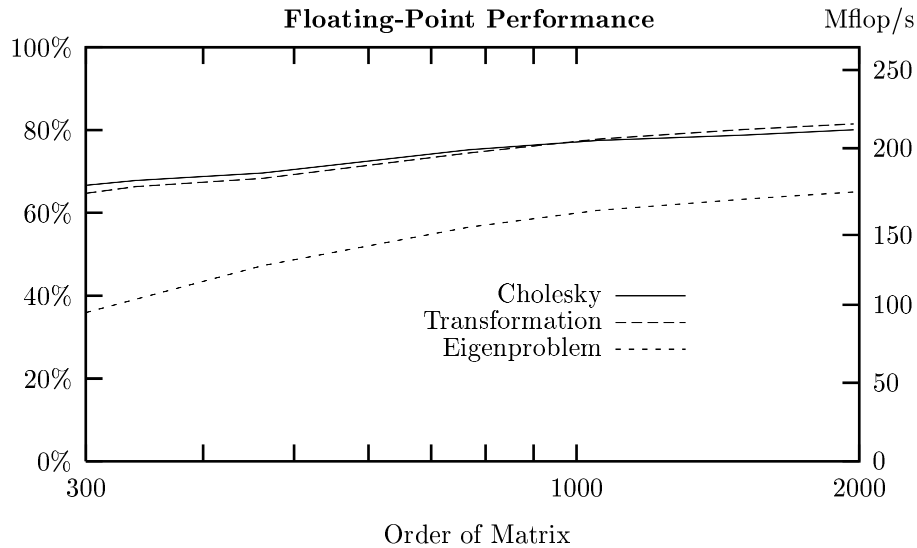

Figure 8.24: Performance of `lapw1c` on an IBM SP2 using a Power2 processor (wide node).

8.7 IBM Workstation RS/6000

n	Cholesky	Transf.	Eigenv.	`lapw1c`	riginal
195	0.1 s	0.5 s	1.5 s	2.4 s	2.3 s
339	0.8 s	2.5 s	5.4 s	9.1 s	9.1 s
463	1.9 s	6.3 s	11.8 s	20.7 s	21.0 s
767	8.8 s	27.2 s	45.9 s	83.5 s	88.1 s
1053	22.6 s	70.1 s	110.7 s	206.4 s	221.1 s
1507	66.5 s	203.3 s	304.8 s	580.7 s	641.9 s
1973	146.4 s	452.3 s	676.1 s	1285.4 s	1440.7 s

Table 8.9: Total run time of `lapw1c` on an IBM workstation.

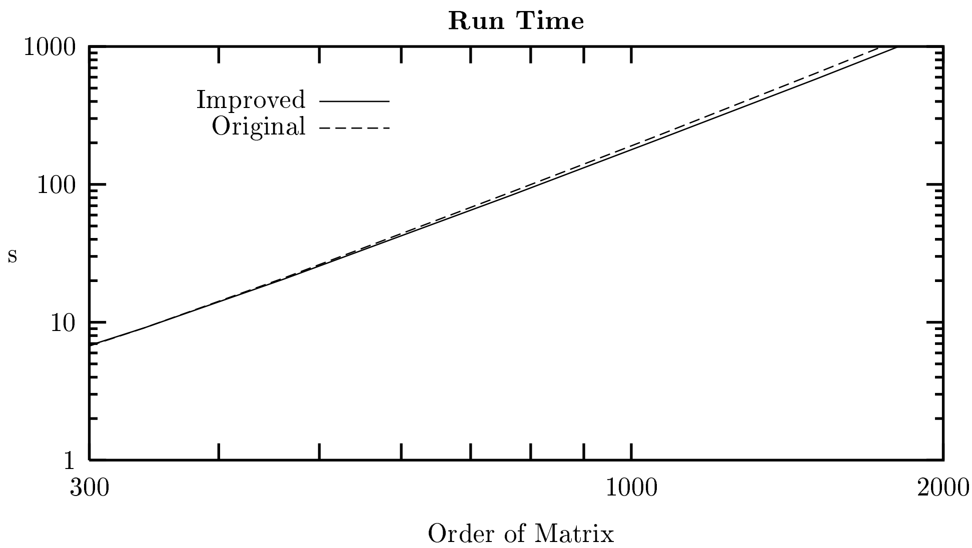

Figure 8.25: Total run time of `lapw1c` on an IBM workstation.

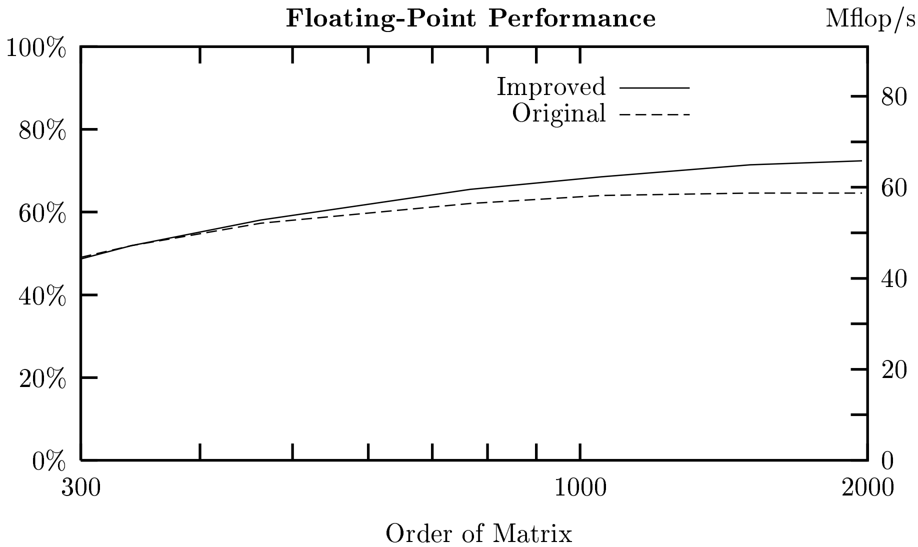

Figure 8.26: Performance of `lapw1c` on an IBM workstation.

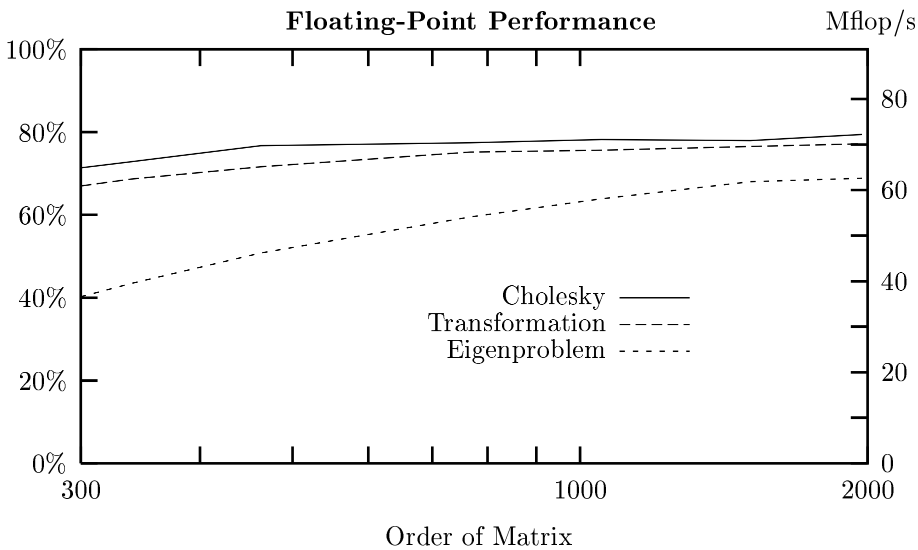

Figure 8.27: Performance of `lapw1c` on an IBM workstation.

8.8 Pentium PC

n	Cholesky	Transf.	Eigenv.	Backtr.	`lapw1c`	Original
195	0.6 s	1.9 s	2.6 s	0.2 s	5.5 s	4.8 s
339	3.4 s	9.4 s	11.6 s	0.7 s	25.2 s	24.0 s
463	8.7 s	23.2 s	28.6 s	1.6 s	62.2 s	59.8 s
767	38.9 s	101.8 s	127.0 s	4.7 s	272.5 s	269.9 s
1053	101.0 s	262.0 s	329.2 s	8.9 s	701.3 s	695.0 s
1507	295.0 s	760.3 s	967.2 s	18.5 s	2041.1 s	2026.3 s
1973	661.9 s	1700.0 s	2200.0 s	33.4 s	4591.4 s	4674.9 s

Table 8.10: Total run time of `lapw1c` on a Pentium PC.

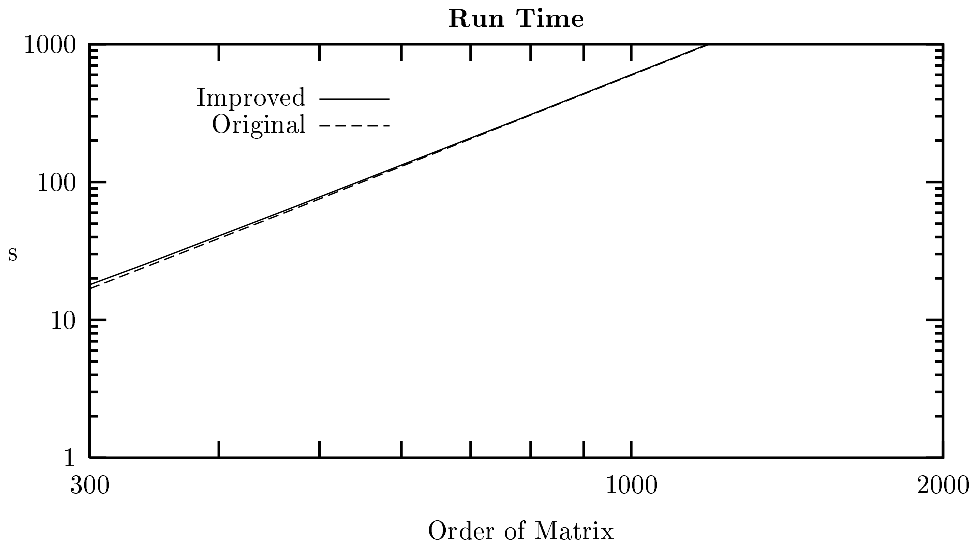

Figure 8.28: Total run time of `lapw1c` on a Pentium PC.

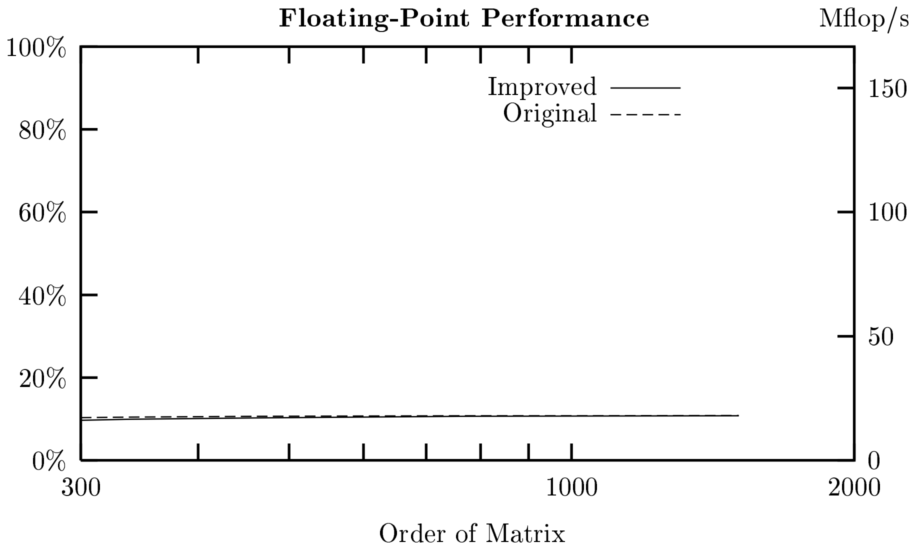

Figure 8.29: Performance of `lapw1c` on a Pentium PC.

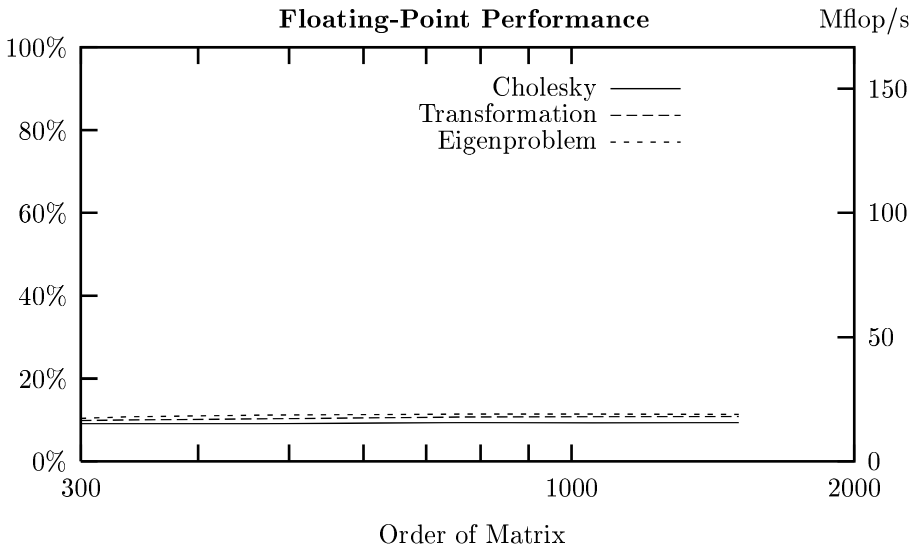

Figure 8.30: Performance of `lapw1c` on a Pentium PC.

n	Cholesky	Transf.	Eigenv.	Backtr.	`lapw1c`	Original
195	0.3 s	0.7 s	1.0 s	0.0 s	2.2 s	2.8 s
339	1.6 s	3.8 s	4.9 s	0.4 s	10.9 s	15.5 s
463	4.3 s	9.5 s	11.9 s	0.8 s	26.7 s	40.3 s
767	19.6 s	42.5 s	53.9 s	2.3 s	118.5 s	184.9 s
1053	51.7 s	110.5 s	139.9 s	4.4 s	306.8 s	361.2 s
1507	150.4 s	320.8 s	451.1 s	9.7 s	932.3 s	1081.3 s
1973	339.6 s	736.8 s	1056.5 s	15.9 s	2149.0 s	2414.1 s

Table 8.11: Total run time of `lapw1c` on a Pentium II PC.

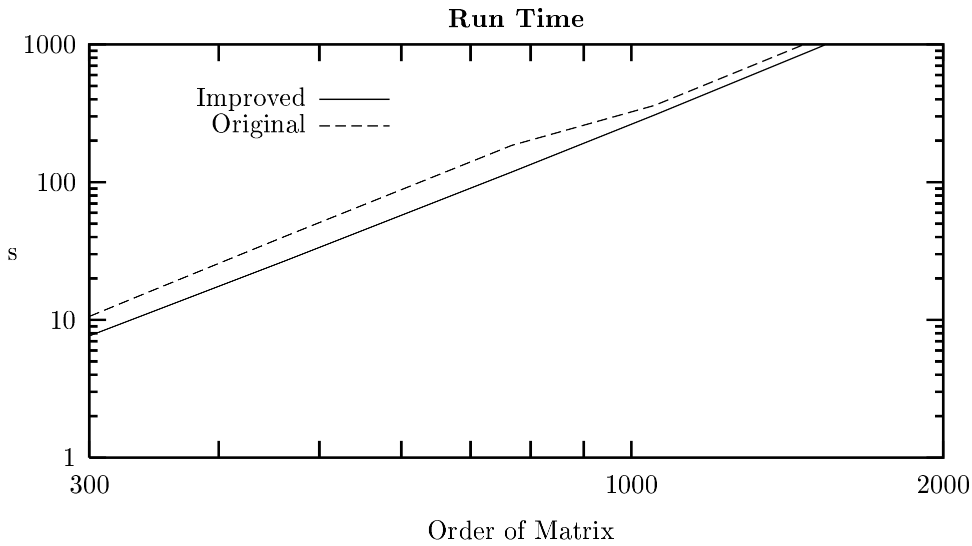

Figure 8.31: Total run time of `lapw1c` on a Pentium II PC.

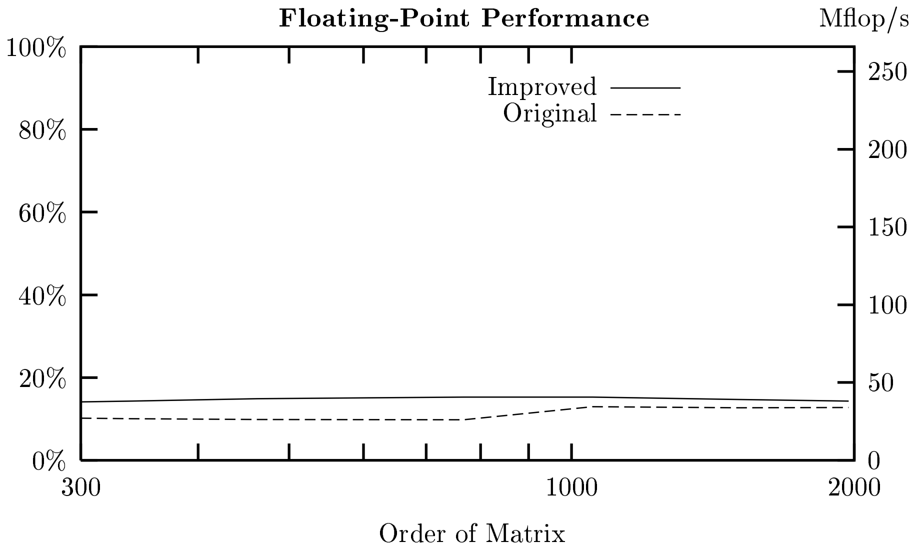

Figure 8.32: Performance of `lapw1c` on a Pentium II PC.

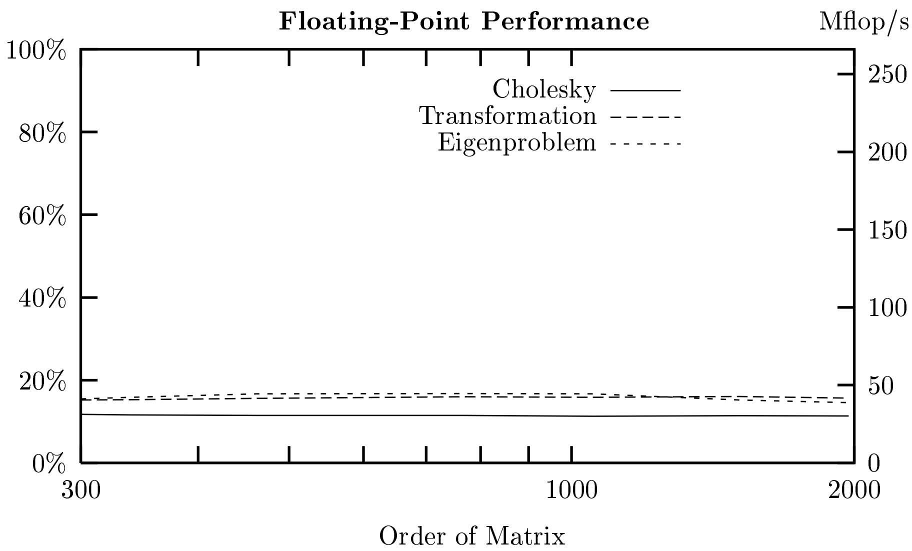

Figure 8.33: Performance of `lapw1c` on a Pentium II PC.

8.9 NEC Vectorcomputer SX-4

n	Cholesky	Transf.	Eigenv.	Backtr.	`lapw1c`	Original
195	0.0 s	0.1 s	0.3 s	0.0 s	0.4 s	0.4 s
339	0.0 s	0.3 s	0.7 s	0.0 s	1.2 s	1.3 s
463	0.1 s	0.6 s	1.3 s	0.0 s	2.1 s	2.5 s
767	0.5 s	2.0 s	4.2 s	0.1 s	6.8 s	8.8 s
1053	1.2 s	4.3 s	8.6 s	0.1 s	14.3 s	16.9 s
1507	3.1 s	10.8 s	20.5 s	0.3 s	34.9 s	41.8 s
1973	6.9 s	22.6 s	40.7 s	0.5 s	70.9 s	84.4 s

Table 8.12: Total run time of `lapw1c` on an NEC SX-4 vectorcomputer.

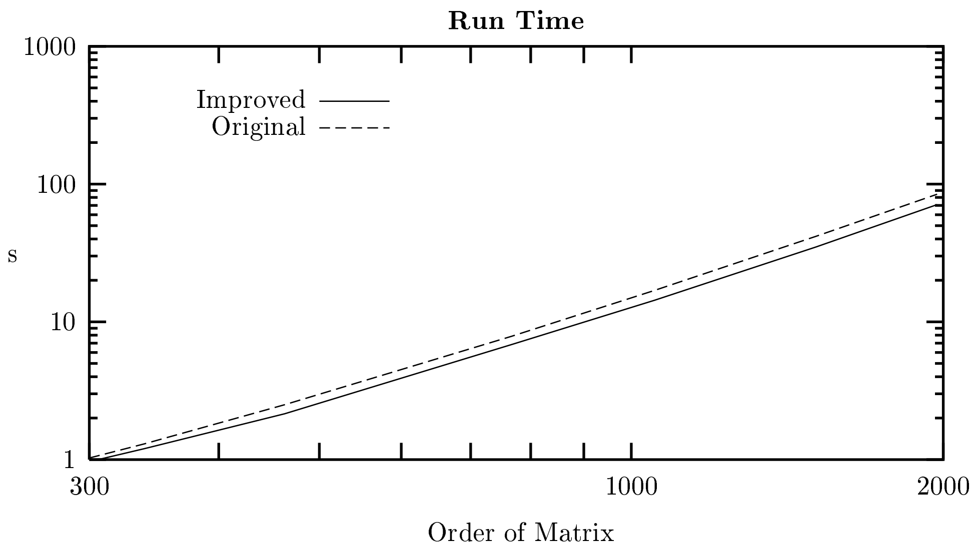

Figure 8.34: Total run time of `lapw1c` on an NEC SX-4 vectorcomputer.

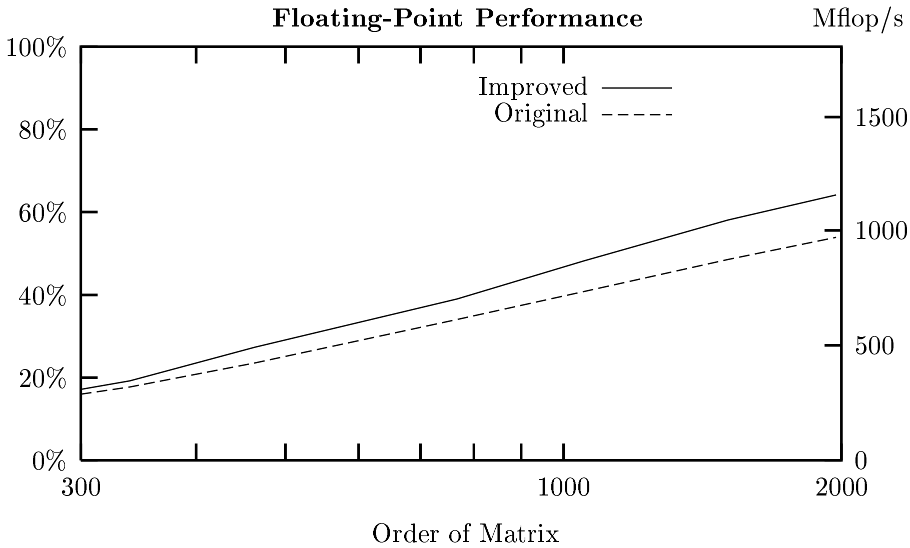

Figure 8.35: Performance of `lapw1c` on an NEC SX-4 vectorcomputer.

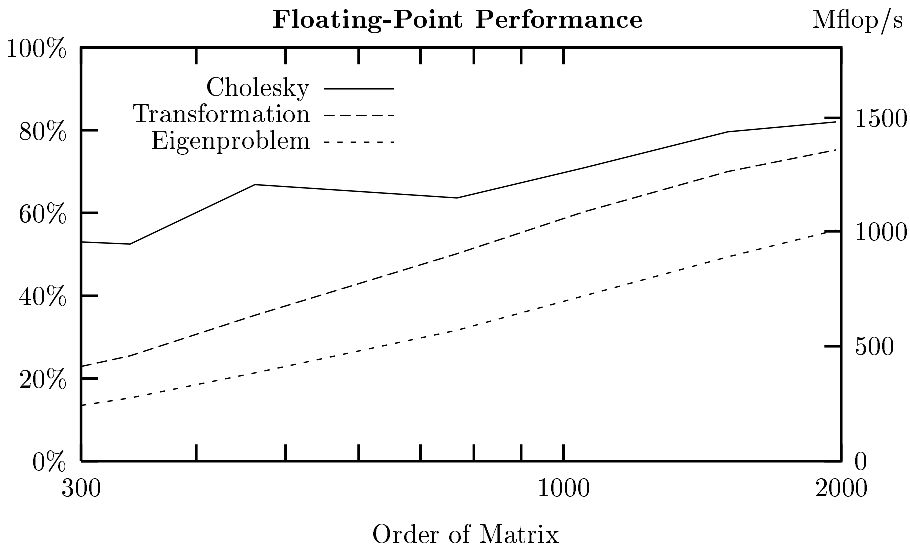

Figure 8.36: Performance of `lapw1c` on an NEC SX-4 vectorcomputer.

8.10 Summary

The results obtained with complex data are similar to the results obtained with real data (see Chapter 7). The main difference is that usually the efficiency for complex data is higher than for real data, because the amount of data which has to be transferred from and to memory is relatively lower for complex data types. Complex floating-point numbers require twice the storage of real floating-point numbers, but every complex multiply-add operation, which is the central operation in complex linear algebra routines, requires four real multiply-add operations. So only half the memory bandwidth is required for algorithms operating on complex data.

Figure 8.37 shows again that the floating-point performance of most current workstations lies within a small band.

Table 8.13 gives the floating-point performance values achieved with the old and the new `lapw1` program. The speedup achieved is given in the last column.

The improved `lapw1` program (whose floating-point performance is shown in Figure 8.38) is more efficient on those computers where the performance values were lower before (see Figure 8.39).

Table 8.14 shows again that processors with high clock rate or high peak performance do not always produce high floating-point performance values. An IBM workstation running with 66 MHz (with a peak performance of 264 Mflop/s) and an efficiency of more than 70 % can easily outperform a DEC Alpha workstation running with 500 MHz (with a peak performance of 1 Gflop/s) for large matrices. The amount of time spent in various subroutines is different for each computer. The ratio of the number of operations in the routines Cholesky, Transformation, and Eigensolver is roughly 11 %, 33 %, and 56 %, respectively. Some computers (for example, HP workstations) have significantly varying efficiency in various routines.

Table 8.15 shows the potential speedup (in terms of the new `lapw1` program) of a fully blocked tridiagonalization routine. The performance of computers with high clock rate and relatively low memory throughput could be improved by more than 70 % when using only matrix-matrix operations also in the tridiagonalization routine.

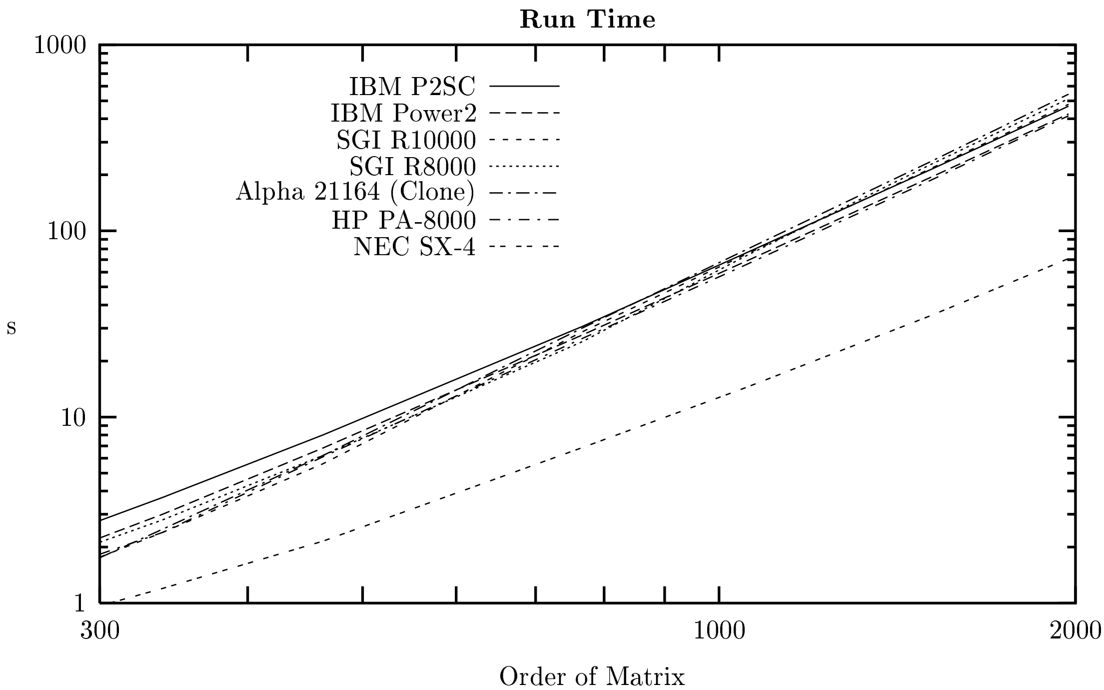

Figure 8.37: Total run time of `lapw1c` on various computer systems.

Host	Cholesky	Transf.	Eigenv.	`lapw1c`	Original	Speedup
SGI R10000	194.7	205.7	156.0	185.3	71.8	158 %
SGI R8000	182.9	188.8	144.9	172.6	78.8	119 %
Alpha 21164	223.1	317.0	110.2	156.8	69.7	125 %
Alpha Clone	267.4	254.4	108.9	165.8	108.0	54 %
HP PA 8000	386.1	403.6	135.1	213.2	163.6	30 %
IBM Power2	211.4	215.1	171.8	204.5	208.1	-2 %
IBM P2SC	207.3	213.0	151.9	189.4	184.8	2 %
IBM Power	69.9	67.9	60.6	68.9	61.5	12 %
Pentium	15.5	18.1	18.6	19.2	18.9	2 %
Pentium II	30.2	41.2	38.8	41.2	36.6	12 %
NEC SX-4	1476.6	1354.0	1004.4	1248.0	1049.1	19 %

Table 8.13: Floating-point performance in Mflop/s for various processors. Computed from time measurements with matrix size $n = 1973$. Estimated number of operations: Cholesky: $n^3/3$, Transformation: n^3, Eigenvalue routines: $4n^3/3$, Sum: $8n^3/3$.

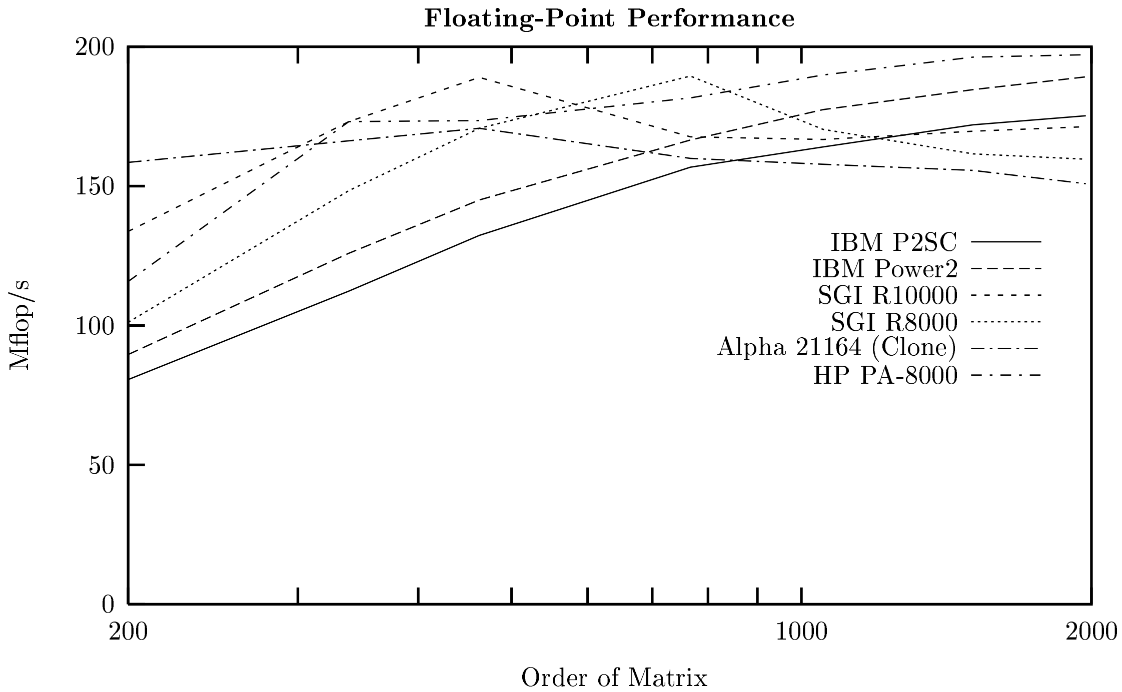

Figure 8.38: Floating-point performance `lapw1c` on various computer systems.

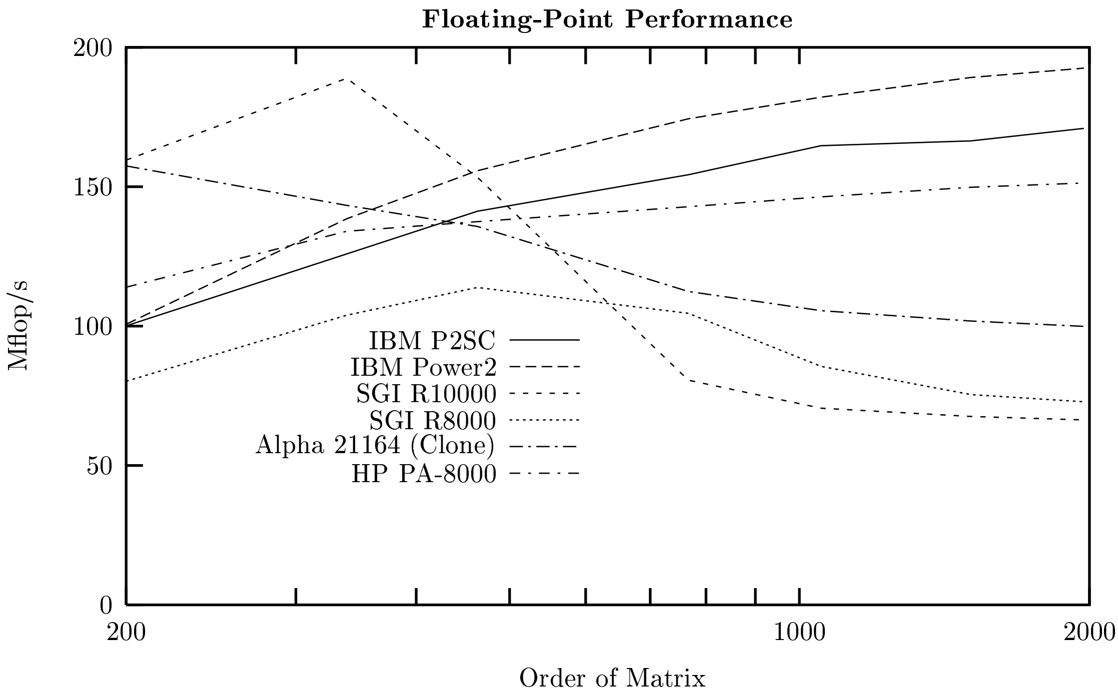

Figure 8.39: Floating-point performance the original program `lapw1c` on various computer systems.

Host	Clock	Peak	Eff.	Cholesky	Transf.	Eigenv.
SGI R10000	195	390	47.4 %	11.0 %	31.2 %	54.9 %
SGI R8000	90	360	48.0 %	10.9 %	31.7 %	55.1 %
Alpha 21164	433	866	18.2 %	8.1 %	22.9 %	65.8 %
Alpha Clone	500	1000	16.5 %	7.1 %	22.2 %	69.3 %
HP PA 8000	180	720	29.6 %	6.4 %	18.3 %	73.0 %
IBM SP2 Wide	66	264	77.5 %	11.2 %	33.0 %	55.1 %
IBM SP2 P2SC	120	480	39.5 %	10.6 %	30.9 %	57.7 %
IBM Power	44	88	78.3 %	11.4 %	35.2 %	52.6 %
NEC SX-4	112.5	1800	69.4 %	9.8 %	32.0 %	57.5 %

Table 8.14: Weight of the main subroutines of `lapw1c` in percent of execution time. Clock rates are in MHz, Peak Performance in Mflop/s, Efficiency is the sustained performance as percentage of the peak performance. (Percentage values taken from measurements with matrix size 2475.)

Host	`lapw1c`	Cholesky	Potential
SGI R10000	185.3	194.7	5 %
SGI R8000	172.6	182.9	6 %
Alpha 21164	156.8	223.1	42 %
Alpha Clone	165.8	267.4	61 %
HP PA 8000	213.2	386.1	81 %
IBM SP2 Wide	204.5	211.4	3 %
IBM SP2 P2SC	189.4	207.3	9 %
IBM Power	68.9	69.9	1 %
NEC SX-4	1248.0	1476.6	18 %

Table 8.15: Potential speedup for the case that the remaining matrix-vector product in `lapw1` is converted to a matrix-matrix product. The column `lapw1c` is the performance of the whole program, including the matrix-vector call, the column Cholesky is the performance of the `cholesky` routine, which is a (matrix-matrix oriented) subroutine of `lapw1c`. (Both columns are in Mflop/s.) The last column is the speedup which can be gained by using only matrix-matrix routines.

Chapter 9

Using the New Procedures

All the new procedures are collected in only one file called `lap_bp.f`. This file is compiled and linked exactly as the other files of `lapw1`, and it has the same dependencies: it requires a parameter from the file `param.inc`: the block size `HB`. It does not depend very much on the optimization level because nearly all the time is spent in routines of the BLAS library.

The parameter `HB` (in the file `param.inc`) should be chosen according to machine characteristics:

1. If the machine has a fast cache memory of about 1 MB, the parameter `HB` should be set to a value between 20 and 50 (32 is a useful default value)[1]. The Cholesky factorization and the subsequent transformation work well with values from 10 to 100, but the eigensolver requires stricter bounds.

2. If the cache memory is not very fast, one has to distinguish between the real and the complex case[2]:

    - For the real case tuned programs exist, which do not use LAPACK or BLAS routines. These programs should be used, because the benefits of the new storage scheme and the new blocking level are not high enough

[1]Machines of type 1 which have been tested are the SGI Power Challenge XL, HP and Alpha workstations using optimized BLAS routines (see Appendix C). The original code was optimized for IBM processors. Thus, on IBM processors there is no spectacular speedup. With the exception of the multichip POWER2 architecture, the new code turned out to be slightly superior on IBM processors, nevertheless.

[2]Machines of type 2 which have been tested do not have highly optimized BLAS: an Alpha workstation using BLAS routines compiled with the Fortran compiler f77 (using -O3 optimization), HP workstations using the BLAS routines of the operating system HP-UX Version 10.20, and a Pentium based Linux PC.

to favor the new routines. The old routines are chosen automatically by setting the parameter `HB` to 1.

- For complex data the original programs use LAPACK and BLAS routines. These are replaced by the new routines with `HB`=32.

3. If the machines are running with less than 100 MHz, it may happen that the new routines produce too much integer overhead also in the complex case. If this situation occurs, a blocking parameter `HB` of 1 should be chosen. In this way the EISPACK oriented original routines are used in the real case and the original LAPACK routines in the complex case.

4. On vector machines the blocking parameter should be chosen according to the vector length of the machine. On an NEC SX4 with vector length 256 optimum performance was obtained by setting `HB` to 255. This choice avoids that the matrix size (after copying to blocked format) is a multiple of the number of memory banks (which is always a power of two) and thus references to matrix elements are distributed over different memory banks. Compared to a value of 256 only a few percent are won. Choosing `HB` too small, e. g., 64, leads to a significant performance degradation. Even on vector machines performance gains due to Level 3 BLAS usage can be obtained. Level 2 BLAS routines often achieve only 60 % of the peak performance, whereas the use of Level 3 BLAS routines results usually in an efficiency of more than 90 %.

Chapter 10

Conclusion

This report demonstrates that it was possible to double the efficiency of the already massively tuned `lapw1` module of WIEN 97 on computer systems with hierarchical memory.

The most time-consuming routines were recognized, some new algorithms (similar to the well known LAPACK routines, but using a new storage scheme) were developed and their performance was tested on many computer systems.

On most computer systems, especially those with fast, large cache memories (like the SGI Power Challenge XL, or the Alpha and HP workstations) significant performance improvement over existing codes has been achieved.

The activity described in this report was a first step in speeding up the WIEN 97 program package. After improving the tridiagonalization by better blocking strategies, the next step will be the parallelization of the most important parts of the code. The achievements described in this report are preparatory steps towards a fine-grained parallel version of WIEN 97.

Acknowledgement

We would like to thank Christoph Ueberhuber for his interest, for helpful discussions, and for his comments on this report, Karlheinz Schwarz and Peter Blaha for providing us with information on the program package WIEN 97, and Wilfried Gansterer for his contributions to Chapter 2.

In addition, we would like to acknowledge the financial support of the Austrian Science Fund FWF.

Bibliography

[1] E. Anderson, J. Dongarra, S. Ostrouchov, *Implementation Guide for* LAPACK, Technical report, LAPACK Working Note 35, 1991.

[2] J. Bilmes, K. Asanovic, C.-W. Chin, J. Demmel, *Optimizing Matrix Multiply using* PHIPAC*: a Portable, High-Performance, ANSI C Coding Methodology*, Proceedings of the International Conference on Supercomputing, ACM, Vienna, Austria, 1997, pp. 340–347.

[3] J. Bilmes, K. Asanovic, J. Demmel, D. Lam, C.-W. Chin, *Optimizing Matrix Multiply using* PHIPAC*: a Portable, High-Performance, ANSI C Coding Methodology*, Technical report, LAPACK Working Note 111, 1996.

[4] P. Blaha, K. Schwarz, P. Sorantin, S. B. Trickey, *Full-Potential, Linearized Augmented Plane Wave Programs for Crystalline Systems*, Comp. Phys. Commun. 59 (1990), pp. 399–415.

[5] J. J. Dongarra, I. S. Duff, D. C. Sorensen, H. A. van der Vorst, *Linear Algebra and Matrix Theory*, SIAM Press, Philadelphia, 1998.

[6] W. N. Gansterer, D. F. Kvasnicka, *High Performance Computing in Material Sciences. The Standard Eigenproblem – Concepts*, Technical Report AURORA TR1998-18, Vienna University of Technology, 1998.

[7] W. N. Gansterer, D. F. Kvasnicka, *High Performance Computing in Material Sciences. The Standard Eigenproblem – Experiments*, Technical Report AURORA TR1998-19, Vienna University of Technology, 1998.

[8] G. Golub, C. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, Maryland, 1996.

[9] P. Hohenberg, W. Kohn, *Inhomogeneous Electron Gas*, Phys. Rev. B 136 (1964), pp. 864–871.

[10] B. Kagstrom, P. Ling, C. Van Loan, *Portable High Performance GEMM-Based Level 3* BLAS, Parallel Processing for Scientific Computing, (R. F. Sincovec et al., Ed.), SIAM Publications, 1993, pp. 339–346.

[11] W. Kohn, L. J. Sham, *Self-Consistent Equations Including Exchange and Correlation Effects*, Phys. Rev. A 140 (1965), pp. 1133–1138.

[12] D. F. Kvasnicka, W. N. Gansterer, C. W. Ueberhuber, *A Level 3 Algorithm for the Symmetric Eigenproblem*, Proceedings of the Third International Meeting on Vector and Parallel Processing (VECPAR '98), Vol. 1, 1998, pp. 267–275.

[13] D. J. Singh, *Simultaneous Solution of Diagonalization and Self-Consistency Problems for Transition-Metal Systems*, Phys. Rev. B 40 , pp. 5428–5431.

[14] D. J. Singh, *Planewaves, Pseudopotentials and the LAPW Method*, Kluwer Academic Publishers, Boston Dordrecht London, 1994.

[15] C. W. Ueberhuber, *Numerical Computation*, Springer-Verlag, Heidelberg, 1997.

Appendix A

How to Create Block Packed Algorithms

The block packed algorithms described in this paper are developed from blocked LAPACK algorithms. These routines were analyzed and modified in several steps:

1. Data dependencies have to be recognized.
2. An additional blocking level is introduced.
3. Blocked loops running in reverse order have to be reorganized.

   Complex subroutine calls must not be misinterpreted (e. g., routines for symmetric matrices have to be changed to routines computing the rows and others for computing the columns).
4. If the routine is working after this modification, the new packing scheme is applied.

   Block size and storage size, which are often equal, must not be confused, especially in subroutine calls.

Data dependencies are not always easy to recognize. To make this task easier several figures helped to recognize which section of a matrix is modified using which other sections (Figure 5.7 and Figure 5.17).

If loops over blocks are running from high to low indices, they have to be rewritten, because otherwise the compute blocks would not match the storage blocks (if the order of the matrix is not a whole multiple of the size of the blocks). The first (highest) block has to be made smaller, and a new block size variable has to be introduced. The loop variable has to run so that the last iteration is done with column one (and the block of columns which is following).

Appendix B

How to Create COMPLEX*16 Algorithms

After DOUBLE PRECISION algorithms were created and tested, only a few changes had to be made to get their COMPLEX*16 counterparts:

1. Parameters and variables are changed to COMPLEX*16 format instead of REAL*8. Attention has to be paid in this step since but some complex BLAS routines may have also real arguments.

2. Constant parameters have to be changed to complex notation (with a zero value in the imaginary part).

3. Conjugate transposed matrices are used instead of transposed ones.

4. Z* BLAS and LAPACK routines (using COMPLEX*16 instead of REAL*8) have to be called instead of the D* ones. Care has to be taken of parameter lists, which are not always compatible to 100 %. Routines for symmetric matrices are changed to routines for Hermitian matrices.

5. For some routines (e. g., `zlatd4`) conversion routines for the conjugation of complex vectors (`LAPACK/zlacgv`) have to be called.

Appendix C

How to Create Optimized BLAS Libraries

To be able to distribute a complete set of routines, the BLAS and LAPACK routines which are necessary in `lapw1` have been collected. The overall performance depends mostly on the performance of the BLAS library, which is often supplied by compiler or computer vendors. The most important BLAS routine is the matrix-matrix multiplication routine `BLAS-3/dgemm` (`BLAS-3/zgemm` in the COMPLEX case), because the modified LAPACK routines introduce several blocking levels which call the `BLAS-3/dgemm` routine for the largest parts of the computation. Additionally, all of the remaining Level 3 BLAS routines can be redesigned in terms of matrix-matrix multiplications (see Kagstrom et al. [10]).

The BLAS code distributed by NETLIB (`http://netlib.no`) was used as a basis for further optimization. To get satisfying performance also on computers where a vendor tuned BLAS is not available (e. g., on personal computers or alpha clones), several attempts to improve performance have been made, the following techniques have been used:

- Standard compilers,
- Code-to-code optimizers,
- Optimizing compilers,
- Code generation tools.

To distinguish various attempts, a label has been assigned to them. The labels and the corresponding compiler options and/or details are given in Table C.1.

Optimization	
SGI-BLAS	`f77 -lcomplib.sgimath`: vendor tuned BLAS on SGI Power Challenge XL
-O3	`f77 -O3`
fopt1	`fopt -cachesize=64 -inline_looplevel=3 -roundoff=3 -scalaropt=3`
fopt2-8	`fopt -cachesize=8 -inline_looplevel=3 -roundoff=3`
-sopt	`f77 -sopt`
OPT	`f77 -mips4 -O3 -TARG:platform=IP25 -OPT:Olimit=0:roundoff=3:div_split=ON:alias=typed -LNO:cs1=32K:cs2=2M -n32`
DXML	`f77 -ldxml`: vendor tuned BLAS on Digital Alpha
fopt2-32, -O5	`fopt -cachesize=32`, then `f77 -O5`
-O3	`f77 -O3`
-tune ev4	`f77 -O3 -tune ev4`
fopt2-8, -O3	`fopt -cachesize=8`, then `f77 -O3`
-tune ev5	`f77 -O3 -tune ev5`
unroll5	`+O3 -transform_loops -unroll 5`
unroll10	`+O3 -transform_loops -unroll 10`
kapf1	`kf77`
kapf2	`kf77 -ag -chl=32,256 -cplc=1 -chs=8,96 -nat -sasc=3`
kapf3	`kf77 -nag -chl=256,256 -cplc=1 -chs=8,96 -nat -sasc=1`
kapf4	`kf77 -ag -chl=256,256 -cplc=0 -chs=8,96 -nat -sasc=1`
kapf5	`kf77 -ag -chl=256,256`
kapf6	`kf77 -ag -chl=128,256 -chs=8,96 -nat`
kapf7	`kf77 -ag -chl=64,256 -chs=8,96 -nat`
kapf8	`kf77 -ag -chl=32,256 -chs=8,96 -nat`
kapf9	`kf77 -nag -chl=32,256 -chs=8,96 -nat`
kapf10	`kf77 -nag -nat`
kapf11	`kf77 -ag -chl=32,256 -chs=8,96 -nat -cplc=1`
PHIPAC1	blocking parameters 4, 16, and 128
PHIPAC2	blocking parameters 4, 12, and 4

Table C.1: Labels for optimization techniques as used in Tables C.2 to C.6.

The tests have been performed on a Digital Alpha to produce a (portable) library which has a performance similar to Digitals DXML library.

Standard Compiler. Using a BLAS library generated with compiler option `-O3` for optimization results in doubled runtimes compared to using a vendor supplied BLAS (see Table C.1).

n	Optimization	`LAPACK/dpotrf`	`LAPACK/dpptrf`	`dphtrf`
500	SGI-BLAS	0.17 s	0.36 s	0.18 s
500	-O3	0.29 s	0.29 s	0.30 s
500	fopt1	4.15 s	1.95 s	4.15 s
500	-sopt	2.86 s	2.37 s	2.85 s
500	OPT	0.22 s	0.28 s	0.24 s
500	fopt2-8	0.30 s	0.29 s	0.33 s
1000	SGI-BLAS	1.40 s	8.68 s	1.59 s
1000	-O3	2.79 s	8.35 s	2.65 s
1000	OPT	1.73 s	7.53 s	1.85 s
1000	fopt2-8	2.24 s	8.01 s	2.66 s
1500	SGI-BLAS	4.55 s	39.6 s	5.54 s
1500	-O3	9.15 s	41.3 s	9.12 s
1500	OPT	5.70 s	38.4 s	6.37 s
1500	fopt2-8	7.85 s	40.8 s	9.24 s

Table C.2: Execution time needed for the Cholesky factorization using full storage (`LAPACK/dpotr`), packed storage (`LAPACK/dpptrf`), and block packed storage scheme (`dphtrf`) (blocking factor $h_b = 64$) on an SGI Power Challenge XL.

Code-to-Code Optimizer. FOPT is a Fortran-to-Fortran code optimizer. Invoked with default parameters (`fopt`), only very poor performance can be achieved. Changing the cache size from 32 kB to 64 kB has only a marginal impact on the execution time.

The advantage of the Fortran-to-Fortran code optimizer FOPT over the use of compiler parameters is that FOPT produces optimized Fortran code which can be transferred to the alpha architecture. The drawback is that only one cache size is specified, although most current machines have more than one cache level.

Tables C.3 to C.6 show various optimization levels on an alpha architecture. The DXML library contains the BLAS as well as the LAPACK routines and is optimized for this architecture.

The `fopt2-32` line is the optimized code for a cache size of 32kB from the SGI. It runs only slightly faster than the optimized code on the alpha (line `-O3`). Using different optimization levels does not make a big difference.

Optimization	LAPACK/dpotrf	LAPACK/dpptrf	dphtrf
DXML	0.40 s	0.60 s	0.43 s
fopt2-32, -O5	0.98 s	1.07 s	0.86 s
-O3	1.15 s	1.02 s	1.01 s
-tune ev4	1.10 s	1.02 s	0.87 s
fopt2-8, -O3	1.04 s	0.97 s	0.86 s
-tune ev5	1.03 s	1.03 s	0.93 s
unroll5	0.95 s	0.95 s	1.00 s
unroll10	1.10 s	0.97 s	0.98 s
kapf1	0.43 s	0.89 s	0.41 s
kapf2	0.42 s	0.91 s	0.43 s
kapf3	0.44 s	0.82 s	0.50 s
kapf4	0.45 s	0.88 s	0.49 s
kapf5	0.49 s	0.93 s	0.47 s
kapf6	0.60 s	0.94 s	0.56 s
kapf7	0.43 s	0.89 s	0.47 s
kapf8	0.41 s	0.77 s	0.41 s
kapf9	0.40 s	0.79 s	0.43 s
kapf10	0.45 s	0.85 s	0.47 s
kapf11	0.40 s	0.78 s	0.43 s
PHIPAC1	0.35 s	0.71 s	0.37 s
PHIPAC2	0.33 s	0.72 s	0.34 s

Table C.3: Execution time needed for the Cholesky factorization (for matrix size $n = 500$) using full storage (`LAPACK/dpotrf`), packed storage (`LAPACK/dpptrf`), and block packed storage scheme (`dphtrf`) (blocking factor $h_b = 32$) on an Alpha 21064 processor.

Optimizing Compiler. Calling the KAP Fortran preprocessor (section C.1) with default values resulted in the line `kapf1` (only the BLAS is optimized by KF77).
The execution time resulting from varying parameters given to KF77 are given in Tables C.3 to C.6.

Optimizing the program with `kapf8` was best. The resulting BLAS is not as good as the DXML library, but sufficient to gain a certain speedup on all alpha processors (compare Table 7.4).

Code Generation Tool. A `BLAS-3/dgemm` code generation program was presented at the ACM Supercomputing '97 conference (see Bilmes et al. [2] and [3]). This code generator performs optimization using loop unrolling, removing as-

Optimization	`LAPACK/dpotrf`	`LAPACK/dpptrf`	`dphtrf`
DXML	2.49 s	4.85 s	2.85 s
fopt2-32, -O5	8.14 s	8.38 s	6.98 s
-O3	9.53 s	8.49 s	8.53 s
-tune ev4	9.16 s	8.74 s	7.46 s
fopt2-8, -O3	8.34 s	8.27 s	6.96 s
-tune ev5	8.69 s	8.53 s	7.26 s
unroll5	7.68 s	7.67 s	7.81 s
unroll10	8.31 s	7.96 s	7.82 s
kapf1	4.62 s	6.66 s	3.43 s
kapf2	4.80 s	6.73 s	3.61 s
kapf3	5.49 s	7.86 s	3.86 s
kapf4	4.73 s	7.59 s	4.31 s
kapf5	7.39 s	9.59 s	4.09 s
kapf6	4.87 s	7.73 s	3.86 s
kapf7	4.65 s	6.63 s	3.70 s
kapf8	4.39 s	6.22 s	3.32 s
kapf9	4.77 s	6.83 s	3.30 s
kapf10	4.61 s	6.63 s	3.45 s
kapf11	4.98 s	7.10 s	3.46 s
PHIPAC1	2.78 s	6.28 s	2.79 s
PHIPAC2	2.45 s	5.95 s	2.63 s

Table C.4: Execution time needed for the Cholesky factorization (for matrix size $n = 1000$) using full storage (`LAPACK/dpotrf`), packed storage (`LAPACK/dpptrf`), and block packed storage scheme (`dphtrf`) (blocking factor $h_b = 32$) on an Alpha 21064 processor.

sumed dependencies (which are the reasons why C compilers are usually not able to produce as optimized code as Fortran compilers), and generating several levels of blocking. This is used to optimize for various cache levels as well as for register re-usage.

The PHIPAC package was released in a pre-alpha version and is available from `http://www.icsi.berkeley.edu/~bilmes/phipac`. It did not run on most machines where it was tried, including SGI Power Challenge XL, HP workstations (PA-RISC 1.1 and PA-RISC 2.0), Digital Alpha workstations, and IBM workstations. With a few changes it could be used on a Linux Alpha machine (the main target machine).

The automatic performance analyzer, which tests all possible blocking parameters

Optimization	`LAPACK/dpotrf`	`LAPACK/dpptrf`	`dphtrf`
DXML	8.44 s	40.68 s	10.07 s
fopt2-32, -O5	29.2 s	51.47 s	25.53 s
-O3	30.8 s	49.56 s	29.04 s
-tune ev4	29.3 s	52.54 s	30.00 s
fopt2-8, -O3	29.0 s	51.98 s	25.35 s
-tune ev5	31.0 s	51.36 s	26.35 s
unroll5	27.0 s	49.48 s	28.05 s
unroll10	28.6 s	50.27 s	28.30 s
kapf1	14.8 s	47.82 s	13.72 s
kapf2	14.3 s	47.99 s	13.56 s
kapf3	16.3 s	52.22 s	15.34 s
kapf4	16.6 s	50.52 s	15.87 s
kapf5	17.4 s	53.30 s	15.64 s
kapf6	17.2 s	51.28 s	14.82 s
kapf7	15.6 s	49.75 s	14.65 s
kapf8	14.4 s	45.52 s	13.02 s
kapf9	14.4 s	44.93 s	13.55 s
kapf10	14.5 s	46.98 s	13.67 s
kapf11	14.2 s	46.80 s	13.15 s
PHIPAC1	9.12 s	43.40 s	10.71 s
PHIPAC2	8.42 s	41.80 s	10.38 s

Table C.5: Execution time needed for the Cholesky factorization (for matrix size $n = 1500$) using full storage (`LAPACK/dpotrf`), packed storage (`LAPACK/dpptrf`), and block packed storage scheme (`dphtrf`) (blocking factor $h_b = 32$) on an Alpha 21064 processor.

only worked for the first level (register re-usage by loop-unrolling), resulting in the parameters 4, 4, 4 for the matrix blocks N, M, and K.

The next level of optimization did not work and resulted in compiler errors, which could only be partially resolved. However, with trying different sets of parameters by hand, good blocking parameters were found. The created routines were compiled with optimization level -O4. The optimizing compiler kcc could not be used (it created run time errors). The results are found in the tables in the lines 'PHIPAC1'.

Computing other parameters by examining cache usage leads to even better performance: The loop unrolling is performed four times, as stated above. Optimizing for the first level cache of 8 kB was not possible, no performance gain could be

Optimization	`LAPACK/dpotrf`	`LAPACK/dpptrf`	`dphtrf`
DXML	19.26 s	122.10 s	23.75 s
-O3	94.20 s	152.00 s	70.88 s
kapf2	42.05 s	151.76 s	32.97 s
kapf8	40.77 s	147.11 s	32.00 s
PHIPAC1	24.10 s	134.44 s	26.12 s
PHIPAC2	20.75 s	134.79 s	25.83 s

Table C.6: Execution time needed for the Cholesky factorization (for matrix size $n = 2000$) using full storage (`LAPACK/dpotrf`), packed storage (`LAPACK/dpptrf`), and block packed storage scheme (`dphtrf`) (blocking factor $h_b = 32$) on an Alpha 21064 processor.

achieved by choosing thus small parameters for blocking. The 96 kB cache allows three matrix blocks of size 24 kB (and leaves space for some overhead), which are 3000 double precision numbers, which gives a symmetric matrix of size 50. Thus, since the loop unrolling of 4 seems to be the optimum, the second blocking parameters should be 12 ($12 * 4 = 48$). The 2 MB cache allows 500 kB per matrix block, which leads to a matrix of size 240, so the third blocking parameter was chosen as 4 (5 would have been possible). The results of this run with parameters 4, 12, and 4 are found in the tables in the lines 'PHIPAC2'.

Compared to the vendor optimized BLAS of the dxml library, the new routines perform even better for small matrices and are within a reasonable distance for larger matrices (less than 10% overhead for the important `dphtrf` routine).

C.1 KAPF

The optimizing preprocessor KAP was used on the DEC, it can be called as `kapf` to produce Fortran code or as `kf77` to produce object code. The most important arguments are

-ag Aggressive optimization: -nag default; -ag

-chl Cache line size: -chl=32,32 default; -chl=32,256

-cplc Cache pre-fetch line count: -cplc=0 default; -cplc=1

-chs Cache size: -chs=8,0 default; -chs=8,96

-nat Natural alignment (only on double-word boundaries): -nnat default; -nat

-sasc Set-associativity: -sasc=1 default; -sasc=3

C.2 PHiPAC

The PhiPac methodology (see Bilmes et al. [3] and [2]) can automatically achieve near-peak performance on a wide range of systems. By analyzing current machines and C compilers, guidelines for writing Portable, High-Performance, ANSI C have been developed. Then automatic code generators have been produced and automatic search scripts have been developed to find the best parameters for a given system. The authors report to achieve within 90% of peak performance on the Sparcstation-20/61, IBM RS/6000-590, HP 712/80i, and 80 % of peak on the SGI Indigo R4k. The resulting routines were often faster than the vendor supplied `BLAS/gemm`.

Developing optimized libraries is a difficult and time-consuming task. Matrix-vector library routines have a large design space. Even excluding algorithmic variants such as Strassen's method for matrix multiplication, blocking sizes, loop nesting permutations, register allocation, and instruction scheduling, can all be varied.

The routines could be manually written in assembly code, but fully exploring the design space might be infeasible, and the resulting code might be unusable or sub-optimal on a different system.

Alternatively, the routines could be written in a high-level language and fed to an optimizing compiler. A high-level language's semantics might obstruct aggressive compiler optimizations. Moreover, it takes significant time and investment before compiler research appears in production compilers, so these capabilities are often simply unavailable.

The goal of the PhiPac methodology is to produce, with minimal effort, high-performance matrix-vector libraries for a wide range of systems. The PhiPac methodology has three components. First, a generic model of current C compilers and microprocessors was developed that provides guidelines for producing portable high-performance ANSI C code. Second, rather than hand-code particular routines, parameterized generators are written, that produce code according to these guidelines. Third, scripts are written that automatically tune code for a particular system by varying the generators' parameters and benchmarking the resulting routines.

The pre-alpha release of the code concentrates on matrix-matrix multiplication. This is the most important routine for the newly developed Lapack-style routines, too, because by introducing several levels of blocking other Blas-calls have been replaced to a large extent by calls to the `BLAS-3/dgemm`.

Only the REAL data format is supplied by the first release of PhiPac. Code generators for the COMPLEX data format have to be written separately.

C.2.1 PHiPAC Coding Methodology

The PhiPac machine model is an abstraction of modern microprocessor-based systems together with their ANSI C compilers. The compiler must also be part of the model because the generated result is ANSI C code. A list of common micro-architectural features was distilled from analyzing a range of machines such as workstations and microprocessor-based SMP and MPP nodes:

- Multiple integer registers, typically 32.
- Multiple floating–point registers, typically 16–32.
- A load/store architecture where it is advantageous to reuse register operands.
- A cached memory hierarchy, with up to three levels of data cache and where cache lines hold multiple words. Cache misses are costly.
- A translation look-aside buffer holding a limited number of page table entries. TLB misses are expensive.
- The cheapest memory addressing mode is base register plus immediate offset.
- Branches are costly. Branches are cheaper using equality or zero comparisons rather than magnitude comparisons.
- Integer multiplication and division is slower than integer addition.
- Independent floating-point add, floating-point multiply, and load/store units.
- Floating-point division is slower than floating-point addition and multiplication.
- Multiple instruction latency on floating-point operations, either due to long pipelines or super-scalar instruction issue.

Production ANSI C compilers were studied and it was determined that the register allocation, instruction selection, and instruction scheduling was reliable. However, more sophisticated optimizations, including pointer alias disambiguation, register and cache blocking, loop unrolling, and software pipelining, were found to be best performed manually.

The authors emphasize that for both micro-architectures and compilers they are determining a *lowest common denominator*. Some micro-architectures or compilers will have superior characteristics in certain attributes, but, if the coding is done assuming these exist, performance will suffer on systems where they do not. Conversely, coding for the lowest common denominator should not adversely affect performance on more capable platforms.

Appendix D

File Naming Convention

The names of the hierarchically blocked algorithms mimic the LAPACK naming scheme. The first letter (`d` or `z`) corresponds to the datatype (double precision and complex, respectively). The next 2 letters describe the matrix type. The second letter stands for the type of the matrix (`p` is used for symmetric, positive definite matrices, `s` for symmetric matrices, `h` for hermitian matrices), the third letter represents the storage scheme (`p` is used for LAPACK packed storage scheme, `h` is used for block packed storage scheme). So the counterpart of the unpacked Cholesky LAPACK factorization `LAPACK/dpotrf` or the packed routine `LAPACK/dpptrf` or the band storage routine `LAPACK/dpbtrf` is named `dphtrf`.

These routines are driver routines which first call a conversion routine from packed to block packed format. Then they call a routine with a '4' at the end of the name. In the above example the routine `dphtrf` calls the routine `dphtr4`.

Other routines required are named in the same way. (With an `h` for the storage scheme and a '4' at the end.)

The subroutines are stored in files with the same name, only converted to lower case. The routine `dphtrf` is stored in the file dphtrf.f

D.1 Conversion Routines

`dbp2p` converts from block packed to packed storage format.

`dp2bp` converts from packed to block packed storage format.

`zbp2p` converts from block packed to packed storage format.

`zp2bp` converts from packed to block packed storage format.

D.2 Cholesky Factorization Routines

`dphtrf` is the Cholesky driver routine.

`dphtr4` is the Cholesky factorization routine.

`zphtrf` is the Cholesky driver routine.

`zphtr4` is the Cholesky factorization routine.

D.3 Transformation Routines

`dshgst` is the transformation driver routine.

`dshgs4` is the transformation routine.

`zhhgst` is the transformation driver routine.

`zhhgs4` is the transformation routine.

D.4 Eigenproblem Routines

`dshevx` is the eigenproblem solver routine.

`dshtrd` is the tridiagonalization routine.

`dlatrd4` is a subroutine of the tridiagonalization routine for transformation of only a few columns of the matrix.

`dlarfg4` computes the transformation for only one column of the matrix. Called from `dlatrd4`.

`dohmt4` is the back-transformation driver routine.

`dohml4` is the back-transformation routine.

`dlarfb2` is the instrumented routine `dlarfb`, called from `dohml4`.

D.5 Auxiliary Function

`dlaij2k` returns the index k of element (i, j) in block packed storage format.

Appendix E

Reliability of the Results

The packed, unpacked and block packed routines perform the same computation. However, sometimes they do not perform the computational steps in the same order. Accordingly, there are differences in the accuracy of the results. For most routines, the observed relative errors are around 10^{-14} or 10^{-15}. Only for the eigenvector computation larger relative errors of about 10^{-11}, or sometimes even 10^{-9}, have been observed. This does not mean that the results of one routine are better than those of other routines, but only that the results are reliable only up to a certain degree for all comparable routines.