1 Introduction

This report will describe the results obtained in the second part of assignment 1 for the course "Optimizing and Parallelizing Compilers", DAIMI, fall 96. The assignment chosen consists in developing, optimize and profile a C program using the compiler `gcc` and the profiling tool `gprof`.

2 The benchmark program

As the benchmark problem a "classical" numerical algorithm from the area of numerical linear algebra has been chosen. The procedure *dbisect* given in appendix A computes a selected subset of the eigenvalues of a symmetric tridiagonal matrix by means of bisection, and is based the ALGOL program presented in [1]. The program implements an algorithm based on the following theorem:

Theorem 1 *(Sturm sequence theorem) Given a symmetric tridiagonal matrix with diagonal elements c_1 to c_n and off-diagonal elements b_2 to b_n ($b_1 \equiv 0$) and $b_i \neq 0$, then for any real number λ let the sequence $p_0(\lambda), \ldots, p_n(\lambda)$ be defined by*

$$p_0(\lambda) = 1, \quad p_1(\lambda) = c_1 - \lambda, \tag{1}$$

$$p_i(\lambda) = (c_i - \lambda)p_{i-1}(\lambda) - b_i^2 p_{i-2}(\lambda), \quad i = 2, \ldots, n \ . \tag{2}$$

Then, in general, the number $a(\lambda)$, of disagreements in sign between consecutive numbers of the sequence is equal to the number of eigenvalues smaller than λ.

In practice the sequence $p_i(\lambda)$ is replaced by

$$q_i(\lambda) = p_i(\lambda)/p_{i-1}(\lambda) \ ,$$

in order to avoid under- and overflow. Now $a(\lambda)$ is given by the number of negative $q_i(\lambda)$. It follows immediately from the theorem, that $q_i(\lambda)$ satisfy the following relations

$$q_1(\lambda) = c_1 - \lambda, \tag{3}$$

$$q_i(\lambda) = (c_i - \lambda) - b_i^2/q_{i-1}(\lambda), \quad i = 2, \ldots, n \ . \tag{4}$$

The procedure in *dbisect* initially applies Gerschgorin circle theorem to compute an interval that contains the entire spectrum of the matrix, and then uses the property of the Sturm sequences in a bisection procedure to compute a subset $\lambda_{m_1}, \ldots, \lambda_{m_2}$ of the eigenvalues, where m_1 and m_2 ($m_1 \leq m_2$) are integer parameters supplied by the user.

For the benchmark, the procedure is tested by computing all the eigenvalues of one of the test matrices mentioned in [1], namely the symmetric tridiagonal matrix with elements

$$c_i = i^4, \quad b_i = i - 1 \quad i = 1, \ldots, n \ .$$

This computation is repeated `rep` times in order to be able to make reliable measurements of the running time. The number `rep` of repetitions, the matrix size n and the parameter $eps1$, indicating the precision with which the eigenvalues should be computed, are read from standard input.

There are several characteristic properties of the procedure:

1. The control flow of the bisection is rather complicated because the algorithm attempts to use information computed for one eigenvalue to improve the initial guess for other eigenvalues that are to be computed.

2. Although very simple, it is difficult to obtain good performance for the innermost loop, which computes the modified Sturm sequences given by equations 3 and 4. This is because the loop contains 2 conditionals (test of floating point values), 1 division, 2 array references, but only 3 floating point and 2 integer operations:

```
int sturm(int n, double c[], double b[], double beta[], double x)
{
  int i,a;
  double q;

  a = 0;
  q = 1.0;
  for (i = 0; i < n; i++) {
    if (q != 0.0)
      q = (c[i] - x) - beta[i]/q;
    else
      q = (c[i] - x) - fabs(b[i])/DBL_EPSILON;
    if (q < 0)
      a = a + 1;
  }
  return a;
}
```

   The numbers mentioned are the ones valid when the first branch of the first conditional is taken, which is almost always the case.

3 Timing results with gcc

The program was compiled on four different platforms with four different CPU's (A Sun Sparcstation (SPARC), an SGI Indy (MIPS R4000), an SGI PowerChallenge (MIPS R8000), an HP 705 (HP PA-RISC) and an IBM SP2 (IBM R/S 6000). On all machines,

Table 1: Execution time in seconds for *dbisect* on different platforms, compiled with `gcc`.

	MIPS R4400	HP PA-RISC	IBM RS/6000	SPARC
no opt.	14.71	24.15	12.92	16.73
-O	10.49	12.65	6.73	11.72
-O2	10.28	13.33	7.34	11.61
best flags only	10.04	12.65	6.53	10.69
+ manual	9.64	10.67	5.57	9.13
+ reciprocal	10.34	10.85	5.57	8.29
Largest improvement	1.53	2.23	2.32	2.02
`gcc` version	2.6.3	2.7.2	2.6.3	2.7.2

except the SGI PowerChallenge, the program was compiled using the `gcc` compiler, with no optimization, normal optimization (-O) and high optimization (-O2) respectively. In the benchmark test (input file `bisect_test.in1`), the eigenvalues of a 500×500 symmetric tridiagonal matrix were computed with the tolerance equal to the machine round-off $\approx 2.2 \times 10^{-16}$. We have chosen to include only a single set of test data, since changing the problem size or the structure of the matrix does not really affect where the time is spent in the program.

The execution times for each version of the program were measured for 10 separate runs, and the lowest execution times were listed in table 1 (the variations were very small - typically a few 100'th of a second).

For all the platforms there is quite a dramatic improvement when going from no optimization to "-O". On the other hand: On two of the four machines (HP and IBM), the execution time increases when going from "-O" to "-O2", while on the other two (SUN and SGI) the performance gained from using "-O2" is relatively small. The reason for this is, that almost all of the time is spent in the single, very simple loop shown above - and apparently, the optimizer at level "-O" does a good job optimizing this simple loop. The reason for the surprising performance degradation observed when compiling with "-O2" rather than "-O" on two of the platforms becomes clear if one studies the assembler code (not included in this report) generated. It turns out that with "-O2", the optimizer tries to make the code for both branches of the first conditional fast. In practice the second branch, which handles the situation when q_i is zero and the result would otherwise overflow, is executed very rarely; hence sacrifying a few clock-cycles/instructions in the first branch in the hope to regain this by being able to generate very efficient code for the second branch is not a clever idea.

4 Profiling

The execution of em dbisect on a SPARCStation was profiled using the gprof program. Unfortunately, for this simple program, the profiler does not give much new information;

Table 2: Execution time in seconds for *dbisect* on different platforms, compiled with native compiler.

	MIPS R4400	HP PA-RISC	IBM RS/6000	MIPS R8000
no opt.	12.57	22.61	9.98	8.97
-O2	9.81	11.14	5.27	4.81
-O3	9.91	10.42	5.17	4.87
best flags	9.80	10.36	4.95	4.81
+ manual	9.38	9.26	4.80	4.55
+ reciprocal	9.47	8.41	4.99	3.47
+ unsafe	—	—	3.67	2.7
Largest improvement	1.34	2.69	2.72	3.32
Improvement over `gcc`	1.03	1.27	1.52	—

Table 3: The "best flags" used for the various native compilers.

```
MIPS R4400 : cc -O3 -sopt
HP PA-RISC : cc -Aa +O3 +Olibcalls
MIPS R8000 : cc -O3  (safe)
MIPS R8000 : cc -O3 -OPT:IEEE_arithmetic=3 (unsafe)
IBM RS/6000: xlc -O3 -qstrict -qipa -qarch=pwr2 -Q  (safe)
IBM RS/6000: xlc -O3 -qipa -qarch=pwr2 -Q  (unsafe)
```

the output is essentially:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 98.81     44.84    44.84   114825     0.39     0.39  sturm
  0.75     45.18     0.34        5    68.00  9036.00  dbisect
```

As expected, almost all of the time is spent computing Sturm sequences.

5 Improvement based on the profile

The performance of the benchmark can be improved in 3 ways: a) invoke some of the more advanved optimization flags of the `gcc` compiler and b) manually to modify the source and finally c) use a better compiler. I have attempted all three procedures with some succes.

5.1 Improvements using compiler options

With `gcc` it turned out that compiling using

`gcc -O6 -funroll-loops`,

gave the best performance on all machines. The actual executon times are listed in table 1 in the row labeled “best flags”. We notice a small improvement over “-O” or “-O2”.

5.2 Improvements from modifications in the source code

As we saw in the output from the profiler, essentially all the time is spent computing Sturm sequences, this is the only part of the program that should be changed. The other parts have so little influence on the overall execution time that it is a waste of time trying to optimize them.

Looking at the function “sturm” one notices that one of the conditions can be avoided in the case where $q < 0$, by moving this test and the incrementation of `a` to the top of the loop. Since we expect this to be the case approximately half of the time, this promises some improvement. After the change, the `sturm` function looks like this:

```
int sturm(int n, double c[], double b[], double beta[], double x)
{
  int i;
```

```
  int a;
  double q;

  a = 0;
  q = 1.0;

  for(i=0; i<n; i++) {
    if (q < 0) {
      a = a + 1;
      q =  (c[i] - x) - beta[i]/q;
    }
    else if (q > 0.0)
      q =  (c[i] - x) - beta[i]/q;
    else
      q = (c[i] - x) - fabs(b[i])/ DBL_EPSILON;
  }
  if (q < 0)
    a = a + 1;

  return a;
}
```

The resulting benchmark timings for the changed programs is shown in table 1 in the row labeled "+manual". As can be seen in the table, this change improves the performance much more than simply going from "-O" to "-O2". Especially on the SPARC the difference is quite significant.

Finally we have attempted to replace the division with q by calculation if the reciprocal of q followed by a multiplication. This might seem odd: should replacing a division with a division and a multiplication? Well, one advantage could be, that the calculation of the reciprocal can be put almost anywhere in the code before it is needed, thus giving the instruction scheduler more freedom. If moreover the machine has a single multiply-and-add instruction, the cost of the extra multiplication can probably be completely overlapped. Some CPUs (like the MIPS R8000) have special instructions for computing the reciprocal, which take fewer clock-cycles than a division.

After incorporating this last improvements, the program looks like this:

```
int sturm(int n, double c[], double b[], double beta[], double x)
{
  int i;
  int a;
  double q,iq;

  a = 0;
```

```
  q = 1.0;

  for(i=0; i<n; i++) {
    if (q < 0) {
      a = a + 1;
      iq = 1.0/q;
      q =  (c[i] - x) - beta[i]*iq;
    }
    else if (q > 0.0)
      iq = 1.0/q;
      q =  (c[i] - x) - beta[i]*iq;
    else
      q = (c[i] - x) - fabs(b[i])/ DBL_EPSILON;
  }
  if (q < 0)
    a = a + 1;

  return a;
}
```

Replacing the division by a reciprocal and a multiplication is a classic example of an "unsafe" optimization, because it changes the round-off errors that are made during the calculation, which might in turn affect the accuracy of the computed eigenvalues. In the benchmark example, however, it did not change the result and we will not elaborate further on this (otherwise interesting) subject.

The resulting benchmark timings for the modified program is shown in table 1 in the row labeled "+reciprocal". As can be seen in the table, this change does not improve the performance very much. In fact on two of the machines (SGI & HP), the performance deteriorates. It seems that although they all have multiply-add instruction, only on the SPARC `gcc` is able to benefit from the extra freedom in the instruction scheduling.

5.3 Improvements from using a better compiler

The `gcc` compiler is a general portable compiler designed to be portable to wide variety of platforms. Therefore one cannot expect (although it is sometimes the case!) the `gcc` compiler to perform quite as well as the native optimizing compilers that is supplied by the manufacturer, and which supposedly takes advantage of every little detail in the architecture. Therefore an obvious way to improve the performance of the benchmark is to use such native compiler, and compare the resulting code with that produced by `gcc`. To do this I have performed the same series of tests, as was done using `gcc` (including running the modified programs) we have done this on a number machines which have a native optimizing compiler. Two of these are actually nodes on parallel supercomputers

(The SGI PowerChallenge, and the IBM SP-2), and many resources have been put into producing good compilers for these high performance machines.

The result of this test is seen in table 2. If we look at the timing results for the two machines that have both a native compiler and `gcc`, the advantage of using the ative compilers varies a great deal among the different machines: on the MIPS R4400 there is hardly any difference, while on the HP PA-RISC we see almost a 20 % difference.

On the SGI PowerChallenge we notice that this machine can really take advantage of the reciprocal calculation. This is because the MIPS R8000 CPU has a fast reciprocal instruction. Furthermore, when we allow the IBM and SGI compilers to perform optimization that might affect the generation of rounding errors, the compilers are in fact able to generate code, that runs approximately 3 times faster than the unoptimized code produced by the same compiler.

References

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

A Source codes

A.1 dbisect.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>
#include "dbisect.h"

#define FUDGE  (double) 1.01



int sturm(int n, double c[], double b[], double beta[], double x)

/***************************************************************************

Purpose:
------------
  Calculates the sturm sequence given by

    q_1(x)  =  c[1] - x

    q_i(x)  =  (c[i] - x) - b[i]*b[i] / q_{i-1}(x)

  and returns a(x) = the number of negative q_i. a(x) gives the number
  of eigenvalues smaller than x of the symmetric tridiagonal matrix
  with diagonal c[0],c[1],...,c[n-1] and off-diagonal elements
  b[1],b[2],...,b[n-1].


Input parameters:
------------------------
  n :
         The order of the matrix.

  c[0]..c[n-1] :
        An n x 1 array giving the diagonal elements of the tridiagonal matrix.

  b[1]..b[n-1] :
        An n x 1 array giving the sub-diagonal elements. b[0] may be
        arbitrary but is replaced by zero in the procedure.
```

```
  beta[1]..beta[n-1] :
         An n x 1 array giving the square of the  sub-diagonal elements,
         i.e. beta[i] = b[i]*b[i]. beta[0] may be arbitrary but is replaced by
         zero in the procedure.

  x :
         Argument for the Sturm sequence.


Returns:
------------------------
  integer a = Number of eigenvalues of the matrix smaller than x.


Notes:
------------------------
On SGI PowerChallenge this function should be compiled with option
"-O3 -OPT:IEEE_arithmetic=3" in order to activate the optimization
mentioned in the code below.


**********************************************************************/

{
  int i;
  int a;
  double q,iq;

  a = 0;
  q = 1.0;

  for(i=0; i<n; i++) {

#ifndef TESTFIRST

    if (q != 0.0) {

#ifndef RECIPROCAL
      q =  (c[i] - x) - beta[i]/q;
#else
  /* A potentially NUMERICALLY DANGEROUS optimizations is used here.
   * The previous statement should read:
```

```
   *         q = (c[i] - x) - beta[i]/q
   * But computing the reciprocal might help on some architectures
   * that have multiply-add and/or reciprocal instuctions.
   */
     iq = 1.0/q;
      q =  (c[i] - x) - beta[i]*iq;
#endif

    }
    else {
      q = (c[i] - x) - fabs(b[i])/DBL_EPSILON;
    }

    if (q < 0)
      a = a + 1;
  }

#else

    if (q < 0) {
      a = a + 1;

#ifndef RECIPROCAL
      q =  (c[i] - x) - beta[i]/q;
#else
      /* A potentially NUMERICALLY DANGEROUS optimizations is used here.
       * The previous statement should read:
       *         q = (c[i] - x) - beta[i]/q
       * But computing the reciprocal might help on some architectures
       * that have multiply-add and/or reciprocal instuctions.
       */
      iq = 1.0/q;
      q =  (c[i] - x) - beta[i]*iq;
#endif

    }
    else if (q > 0.0) {
#ifndef RECIPROCAL
      q =  (c[i] - x) - beta[i]/q;
#else
      /* A potentially NUMERICALLY DANGEROUS optimizations is used here.
       * The previous statement should read:
       *         q = (c[i] - x) - beta[i]/q
```

```
       * But computing the reciprocal might help on some architectures
       * that have multiply-add and/or reciprocal instuctions.
       */
      iq = 1.0/q;
      q =  (c[i] - x) - beta[i]*iq;
#endif
    }
    else {
      q = (c[i] - x) - fabs(b[i])/DBL_EPSILON;
    }

  }
  if (q < 0)
    a = a + 1;
#endif

  return a;
}
```

```
void dbisect(double c[], double b[], double beta[],
             int n, int m1, int m2, double eps1, double *eps2, int *z,
             double x[])


/***************************************************************************

Purpose:
------------

  Calculates eigenvalues lambda_{m1}, lambda_{m1+1},...,lambda_{m2} of
  a symmetric tridiagonal matrix with diagonal c[0],c[1],...,c[n-1]
  and off-diagonal elements b[1],b[2],...,b[n-1] by the method of
  bisection, using Sturm sequences.


  Input parameters:
------------------------

  c[0]..c[n-1] :
        An n x 1 array giving the diagonal elements of the tridiagonal matrix.
```

```
  b[1]..b[n-1] :
        An n x 1 array giving the sub-diagonal elements. b[0] may be
        arbitrary but is replaced by zero in the procedure.

  beta[1]..beta[n-1] :
         An n x 1 array giving the square of the  sub-diagonal elements,
         i.e. beta[i] = b[i]*b[i]. beta[0] may be arbitrary but is replaced by
         zero in the procedure.

  n :
         The order of the matrix.

  m1, m2 :
         The eigenvalues lambda_{m1}, lambda_{m1+1},...,lambda_{m2} are
         calculated (NB! lambda_1 is the smallest eigenvalue).
         m1 <= m2must hold otherwise no eigenvalues are computed.
         returned in x[m1-1],x[m1],...,x[m2-1]

  eps1 :
         a quantity that affects the precision to which eigenvalues are
         computed. The bisection is continued as long as
         x_high - x_low >  2*DBL_EPSILON(|x_low|  + |x_high|) + eps1       (1)
         When (1) is no longer satisfied, (x_high + x_low)/2  gives the
         current eigenvalue lambda_k. Here DBL_EPSILON (constant) is
         the machine accuracy, i.e. the smallest number such that
         (1 + DBL_EPSILON) > 1.

  Output parameters:
------------------------


  eps2 :
        The overall bound  for the error in any eigenvalue.
  z :
        The total number of iterations to find all eigenvalues.
  x :
        The array  x[m1],x[m1+1],...,x[m2] contains the computed eigenvalues.

**********************************************************************/
{
  int i;
  double h,xmin,xmax;
  beta[0]  = b[0] = 0.0;
```

```
/* calculate Gerschgorin interval */
xmin = c[n-1] - FUDGE*fabs(b[n-1]);
xmax = c[n-1] + FUDGE*fabs(b[n-1]);
for(i=n-2; i>=0; i--) {
  h = FUDGE*(fabs(b[i]) + fabs(b[i+1]));
  if (c[i] + h > xmax)  xmax = c[i] + h;
  if (c[i] - h < xmin)  xmin = c[i] - h;
}

/* estimate precision of calculated eigenvalues */
*eps2 = DBL_EPSILON * (xmin + xmax > 0 ? xmax : -xmin);
if (eps1 <= 0)
  eps1 = *eps2;
*eps2 = 0.5 * eps1 + 7 * *eps2;
{ int a,k;
  double x1,xu,x0;
  double *wu;

  if( (wu = (double *) calloc(n+1,sizeof(double))) == NULL) {
    fputs("bisect: Couldn't allocate memory for wu",stderr);
    exit(1);
  }

  /* Start bisection process  */
  x0 = xmax;
  for(i=m2; i >= m1; i--) {
    x[i] = xmax;
    wu[i] = xmin;
  }
  *z = 0;
  /* loop for the k-th eigenvalue */
  for(k=m2; k>=m1; k--) {
    xu = xmin;
    for(i=k; i>=m1; i--) {
      if(xu < wu[i]) {
        xu = wu[i];
        break;
      }
    }
    if (x0 > x[k])
      x0 = x[k];
```

```
      x1 = (xu + x0)/2;
      while ( x0-xu > 2*DBL_EPSILON*(fabs(xu)+fabs(x0))+eps1 ) {
        *z = *z + 1;

        /* Sturms Sequence  */

        a = sturm(n,c,b,beta,x1);

        /* Bisection step */
        if (a < k) {
          if (a < m1)
            xu = wu[m1] = x1;
          else {
            xu = wu[a+1] = x1;
            if (x[a] > x1) x[a] = x1;
          }
        }
        else {
          x0 = x1;
        }
        x1 = (xu + x0)/2;
      }
      x[k] = (xu + x0)/2;
    }
    free(wu);
  }
}
```