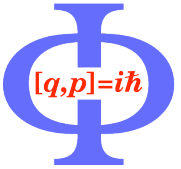


OpenMP implementation of the Householder reduction for large complex Hermitian eigenvalue problems

ParCo2007, Jülich, 05.09.2007

Andreas Honecker
Institut für Theoretische Physik, Georg-August-Universität Göttingen, Germany

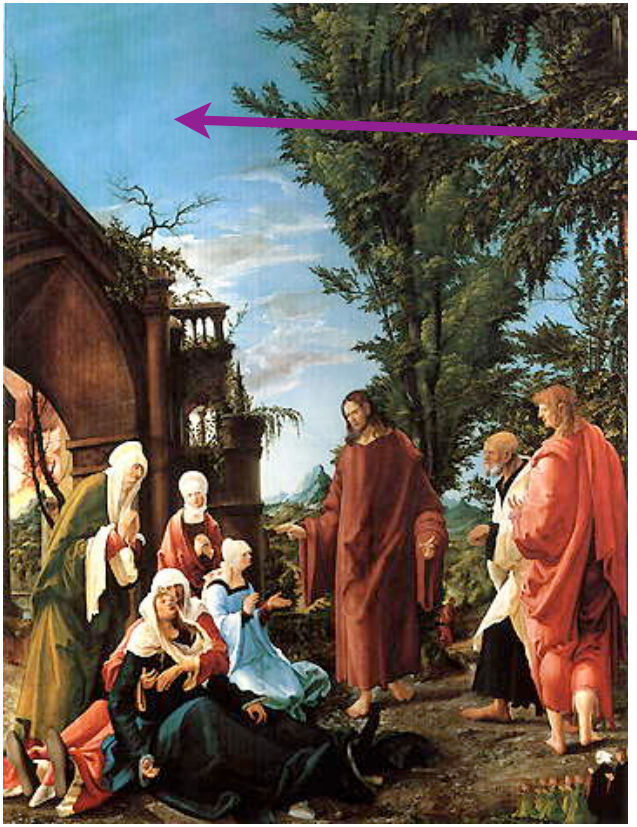
Josef Schüle
Gauss-IT-Zentrum, Technische Universität Braunschweig, Germany

Gauß IT ZENTRUM TU BRAUNSCHWEIG

Outline

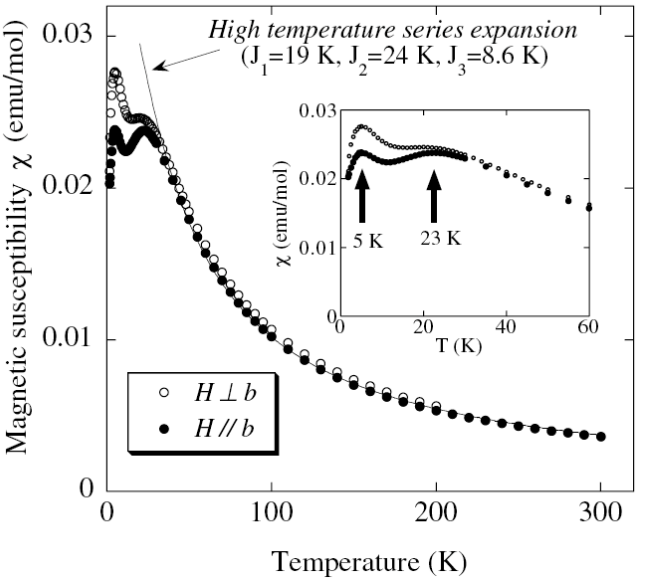
- **Motivation:**
  correlated materials & models
- **Householder with OpenMP**
- **Serial performance**
- **Parallel performance**
- **Discussion & Conclusions**

Motivation: correlated materials



Albrecht Altdorfer (1520):
Christ taking Leave of his Mother

Azurite

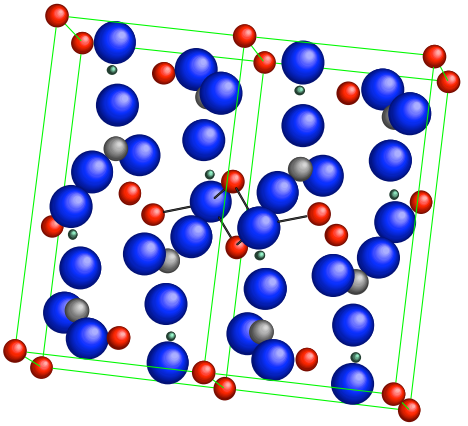


Kikuchi *et al.*, Phys. Rev. Lett. (2005)

Motivation: correlated models

- crystal structure $\Rightarrow$ model

Azurite



- Heisenberg model:

$$H = \sum_{\langle i,j \rangle} J_{i,j}\, \vec{S}_i \cdot \vec{S}_j$$

lives in $\underbrace{\mathbb{C}^{2s} \otimes \mathbb{C}^{2s} \otimes \cdots \otimes \mathbb{C}^{2s}}_{N \text{ times}}$

- use internal & spatial symmetries
- Fourier transformation $\Rightarrow$ **complex** matrices
- Thermodynamics (e.g. magnetic susceptibility): need **all eigenvalues**

Householder reduction to tridiagonal form

- $\mathcal{O}(n^3)$ operations
- $\approx 8\,n^2$ Bytes to store double precision complex Hermitian matrix in „packed“ form

Parallelization strategy

Complex Householder algorithm in C99

```
#define complex_abs_sq(a) (creal(a)*creal(a)+cimag(a)*cimag(a))

for(i=dim-1; i>=1; i--) {                    /* start from the lower right corner of m */
  h=scale=0;
  if(l > 0) {
    for(k=0; k<=l; k++)                      /* sum row of m */
      scale += cabs(m[i][k]);
    if(scale == 0)                           /* skip transformation if this is zero */
      e[l] = conj(m[i][l]);                  /* copy conjugate of matrix element above diagonal */
    else {
      for(k=0; k<=l; k++) {
        m[i][k] /= scale;                    /* rescale entries of m */
        h += complex_abs_sq(m[i][k]);        /* and accumulate norm (squared) of "u" in h */
      }
      f = m[i][l];                           /* store entry m[i][i-1] in f */

      /* we have to choose sigma = m[i][i-1]/m[i][i-1]^* |m[i]|^2 */
      if(complex_abs_sq(f))
        sigma = f / conj(f);
      else
        sigma = 1;                           /* phase doesn't matter for |m[i][i-1]|^2 = 0 */
      sigma *= h;

      /* g = +/- sqrt(sigma), where the sign is chosen such that |m[i][i-1] - g|
         is as large as possible */
      g = csqrt(sigma);
      c1 = f+g;
      c2 = f-g;
      if(complex_abs_sq(c1) > complex_abs_sq(c2))
        g = -g;

      c1 = scale*g;                                  /* store scale*g below diagonal */
      e[l] = conj(c1);                               /* and scale*g^* above */

      /* h = |m[i]|^2 - Re(g^* f) */
      h -= (creal(g)*creal(f) + cimag(g) * cimag(f));
      /* substitute m[i][i-1] by  m[i][i-1] - g = f - g (=>  m[i] = u^t) */
      m[i][l] = f-g;

      for(j=0; j<=l; j++) {
        if(compute_eigenvectors)
          m[j][i] = m[i][j]/h;               /* Store u^t/H in ith column of m */
        p[j] = 0;                            /* Initialize p=0 */
      }

      for(k=0; k<=l; k++) {                  /* First part of A u -> p */
        cptr1 = m[i]+k; cptr2 = m[k]; cptr3 = p;
        for(j=0; j<=k-1; j++)
          /* The elements above the diagonal can be related to those below using hermiticity */
          *cptr3++ += conj(*cptr2++ * *cptr1);        /* g += m[k][j]^* * m[i][k]^* */
      }
      for(j=0; j<=l; j++)
        p[j] /= h;                                   /* Rescale properly */

      for(j=0; j<=l; j++) {                          /* Second part of A u -> p */
        g = 0;                                       /* Form an element of A*u in g */
        cptr1 = m[i]; cptr2 = m[j];
        for(k=0; k<=j; k++)
          g += conj(*cptr1++) * *cptr2++;            /* g += m[j][k]*m[i][k]^* */
        p[j] += (g/h);                               /* Store the result in p[j] */
      }

      f = 0;
      for(j=0; j<=l; j++)
        f += m[i][j] * p[j];                         /* accumulate u * p in f */
      if(! is_zero(cimag(f)/cabs(f)) )
        display_error("Householder: K is not real, as expected");
      hh = creal(f)/(h+h);                           /* form K */

      for(j=0; j<=l; j++) {                          /* Form q and overwrite p with it */
        f = conj(m[i][j]);                           /* store entry m[i][j]^* = u[j] in f */
        g = p[j] = p[j]-hh*f;                        /* p[j] = p[j] - hh*f */
        cptr1 = p; cptr2 = m[j]; cptr3 = m[i];
        for(k=0; k<=j; k++)                          /* Reduce m - we can stop at j */
          *cptr2++ -= (f * conj(*cptr1++) + g * *cptr3++);
      }
    }
    else
      e[l] = conj(m[i][l]);                          /* copy conjugate of matrix element above diagonal */
    p[i] = h;                                        /* store h in p[i] - to check later if it was 0 */
  }
  p[0] = 0;                                          /* last transformation step: nothing to do */

  if(compute_eigenvectors)
    for(i=0; i<dim; i++) {
      l = i-1;
      if(creal(p[i])) {                              /* skip block if we did not do a transformation */
        for(j=0; j<=l; j++)
          p[j] = 0;
        for(k=0; k<=l; k++) {
          cptr1 = m[i]+k; cptr2 = m[k]; cptr3 = p;
          for(j=0; j<=l; j++)                        /* Use u^t und u^t/H stored in m to form PQ */
            *cptr3++ += (*cptr1 * *cptr2++);         /* g += m[i][k]*m[k][j] */
        }
        for(k=0; k<=l; k++) {                        /* Now we need a second loop */
          cptr1 = p; cptr2 = m[k];
          for(j=0; j<=l; j++)
            *cptr2++ -= (*cptr1++ * conj(m[k][i]));  /* m[k][j] -= g * m[k][i]^* */
        }
      }
      d[i] = m[i][i];                                /* this statement must be kept in any case */
      m[i][i] = 1;                                   /* prepare next iteration */
      for(j=0; j<=l; j++)
        m[i][j] = m[j][i] = 0;
    }
  else
    for(i=0; i<dim; i++)
      d[i] = m[i][i];                                /* this statement must be kept in any case */
```

Parallelization strategy

Complex Householder algorithm in C99

```
for(i=dim-1; i>=1; i--) {
      ⋮
       for(j=0; j<=l; j++) {
         f = conj(m[i][j]);
         g = p[j] = p[j]-hh*f;
         cptr1 = p; cptr2 = m[j]; cptr3 = m[i];
         for(k=0; k<=j; k++)
           *cptr2++ -= (f * conj(*cptr1++) + g * *cptr3++);
       }
      ⋮
 }
```

☞ $\mathcal{O}(n^3 = \mathtt{dim}^3)$ operations

(16 $n^3/3$ floating point operations for eigenvalues)

Parallelization strategy

Parallelization attempt I

```
for(i=dim-1; i>=1; i--) {
      :
        for(j=0; j<=l; j++)
           p[j] -= hh*conj(m[i][j]);
#pragma omp parallel for private(j,k,f,g,cptr1,cptr2,cptr3)
        for(j=0; j<=l; j++) {
           f = conj(m[i][j]);
           g = p[j];
           cptr1 = p; cptr2 = m[j]; cptr3 = m[i];
           for(k=0; k<=j; k++)
            *cptr2++ -= (f * conj(*cptr1++) + g * *cptr3++);
        }
      :
 }
```

Problem: Load-imbalance

Parallelization strategy

Parallelization attempt II

```
for(i=dim-1; i>=1; i--) {
      ⋮
        for(j=0; j<=l; j++)
           p[j] -= hh*conj(m[i][j]);
#pragma omp parallel for private(proc,j,k,f,g,
                          cptr1,cptr2,cptr3)
        for(proc=0; j<omp_numproc; proc++) {
         for(j=proc; j<=l; j += omp_numproc) {
           f = conj(m[i][j]);
           g = p[j];
           cptr1 = p; cptr2 = m[j]; cptr3 = m[i];
           for(k=0; k<=j; k++)
            *cptr2++ -= (f * conj(*cptr1++) + g * *cptr3++);
        }
      ⋮
 }
```

☞ **Problem:** Level2 CPU Cache for n=dim≥10000

Parallelization strategy

Final solution

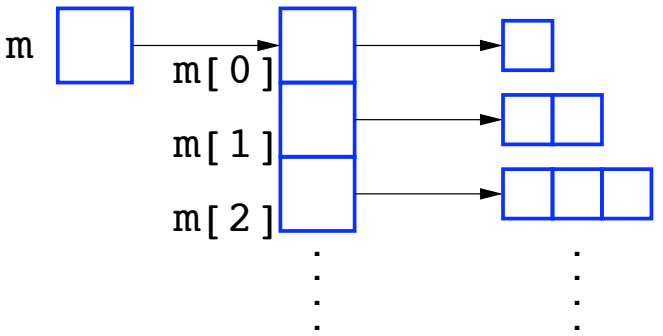
```
for(i=dim-1; i>=1; i--) {
      :
      for(j=0; j<=l; j++)
        p[j] -= hh*conj(m[i][j]);
      nchunks = compute_chunks(sizeof(complex double), l);
#pragma omp parallel for private(chunk,j,k,f,g,cbegin,cend,
                      cendt,cptr1,cptr2,cptr3) if(nchunks>1)
      for(chunk=0; chunk<nchunks; chunk++) {
        cbegin = chunks[chunk].begin;
        cendt = chunks[chunk].end;
        pptr = p+cbegin;
        for(j=0; j<=l; j++) {
          f = conj(m[i][j]);
          g = p[j];
          cptr1 = pptr;
          cptr2 = m[j]+cbegin;
          cptr3 = m[i]+cbegin;
          cend = (j<cendt)?j:cendt;
          for(k=cbegin; k<=cend; k++)
            *cptr2++ -= (f * conj(*cptr1++) + g * *cptr3++);
        }
      :
}
```

☞ **set of administrative routines for handling „chunks“**

Further features of our code

- **checkpoints**
  - ☞ second set of administrative routines
- **matrix** `m`**:** vector of pointers to rows
  - ☞ convenient access also in „packed“ form
  - ☞ free currently unneeded memory

    – restore from checkpoint when needed



Serial Performance: „Small“ Problem

(complex dimension 1184 – main memory: 11MByte)

Choice of programming language

	2.4GHz Intel Core2		1.9GHz IBM Power5		1.6GHz Itanium2	
variant	gcc 4.1.1	Intel icc 10.0	gcc 4.0.2	IBM xlc 8.0	gcc 4.1.0	Intel icc 9.1
C99	5.18	4.60	7.05	17.03	20.42	8.04
C++	5.11	44.23	7.05	10.18	26.70	16.87
plain C	5.48	5.46	7.94	6.43	28.74	14.68
SSE3 (inline assembler)	4.39	4.41				

Eigenvalues only: runtime [seconds]

☞ **C++: risky**

Choice of programming language

	2.4GHz Intel Core2		1.9GHz IBM Power5		1.6GHz Itanium2	
variant	gcc 4.1.1	Intel icc 10.0	gcc 4.0.2	IBM xlc 8.0	gcc 4.1.0	Intel icc 9.1
C99	5.18	4.60	7.05	17.03	20.42	8.04
C++	5.11	44.23	7.05	10.18	26.70	16.87
plain C	5.48	5.46	7.94	6.43	28.74	14.68
SSE3 (inline assembler)	4.39	4.41				

Eigenvalues only: runtime [seconds]

☞ **C99: viable** (soon with OpenMP?)

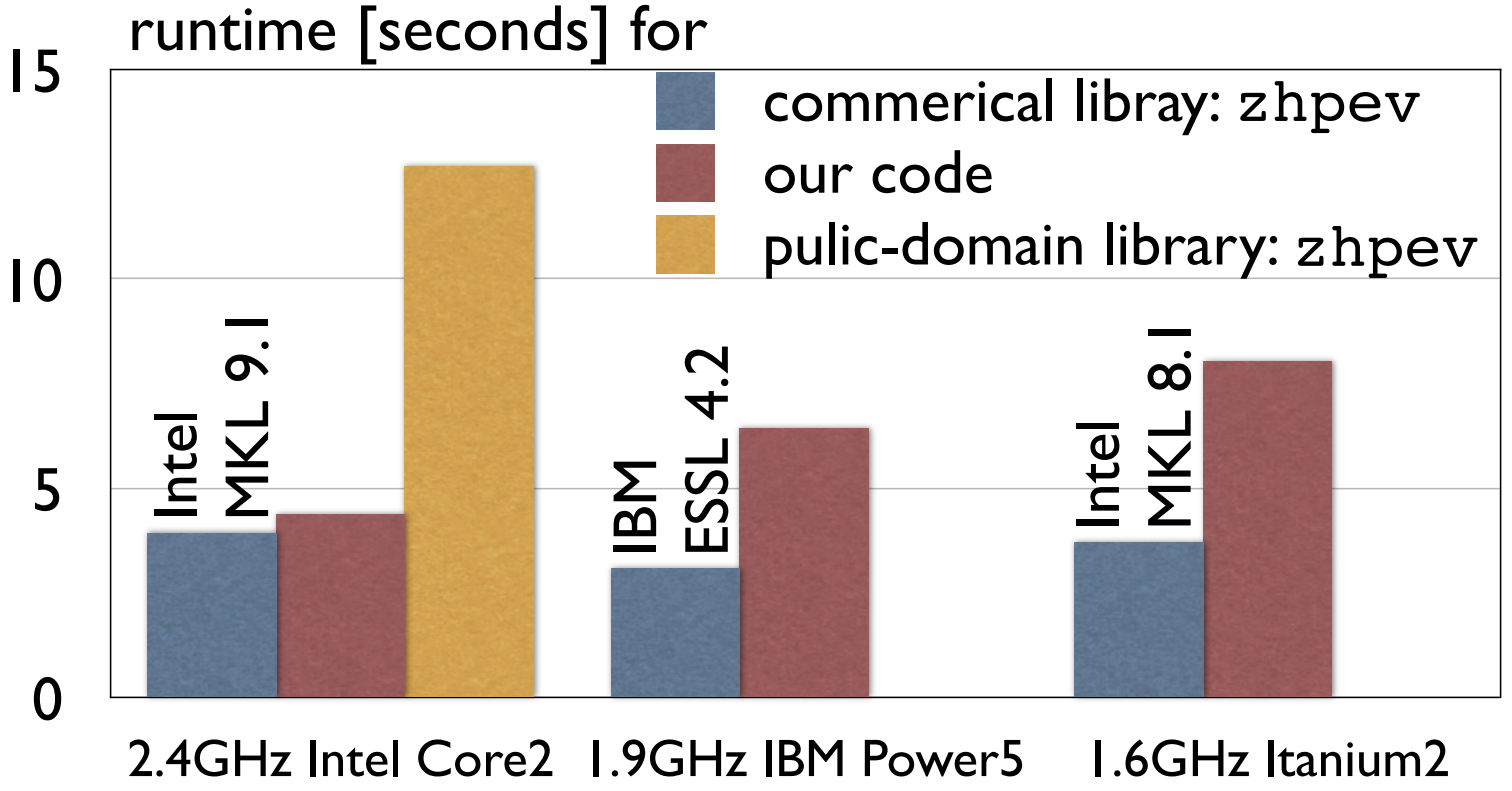
Choice of programming language

	2.4GHz Intel Core2		1.9GHz IBM Power5		1.6GHz Itanium2	
variant	gcc 4.1.1	Intel icc 10.0	gcc 4.0.2	IBM xlc 8.0	gcc 4.1.0	Intel icc 9.1
C99	5.18	4.60	7.05	17.03	20.42	8.04
C++	5.11	44.23	7.05	10.18	26.70	16.87
plain C	5.48	5.46	7.94	6.43	28.74	14.68
SSE3 (inline assembler)	4.39	4.41				

Eigenvalues only: runtime [seconds]

used in the following

Comparison with other libraries



- ☞ our code: within a factor 1...2 of commercial libraries
- ☞ our code: faster than public-domain LAPACK

Parallel Performance: „Large“ Problem

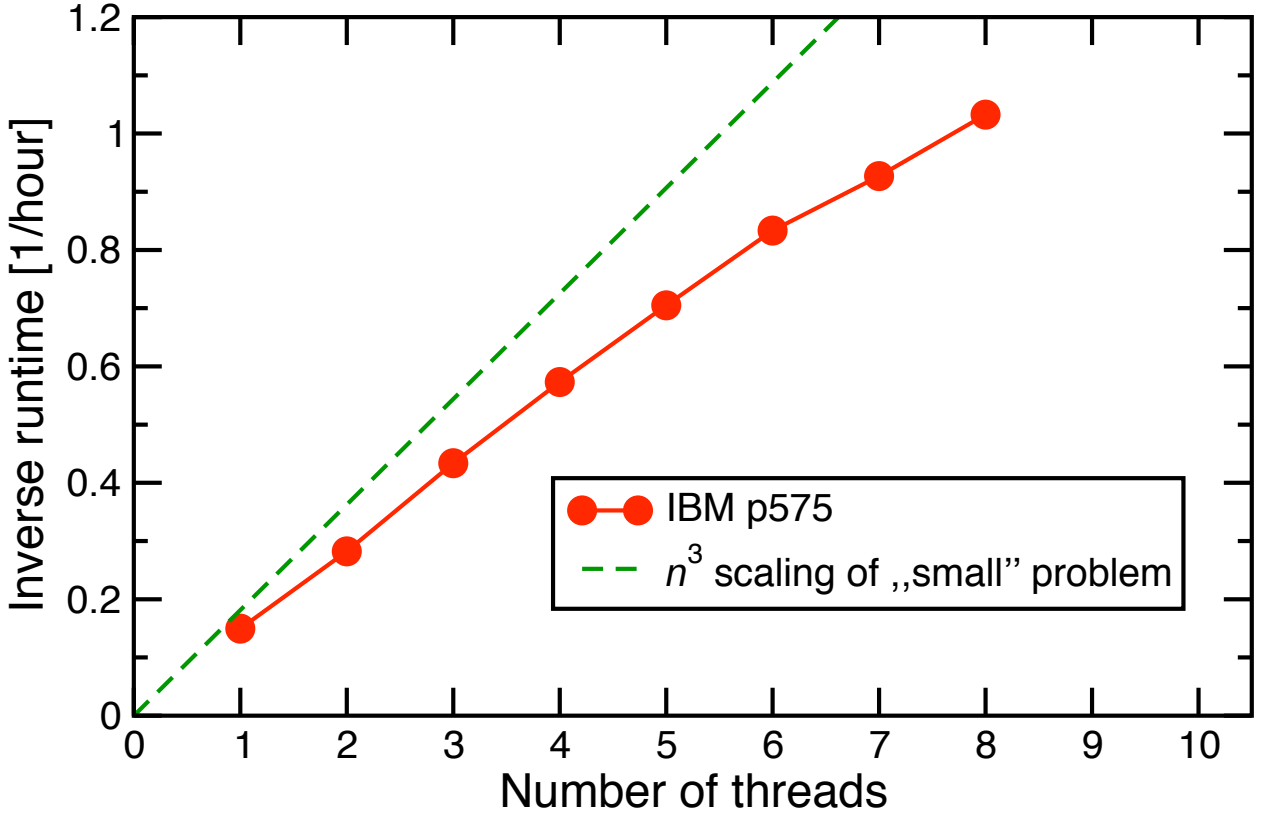
(complex dimension 41835 – main memory: 13GByte)

IBM p575, 1 node: all eigenvalues

(8 × 1.9GHz Power5 CPUs)

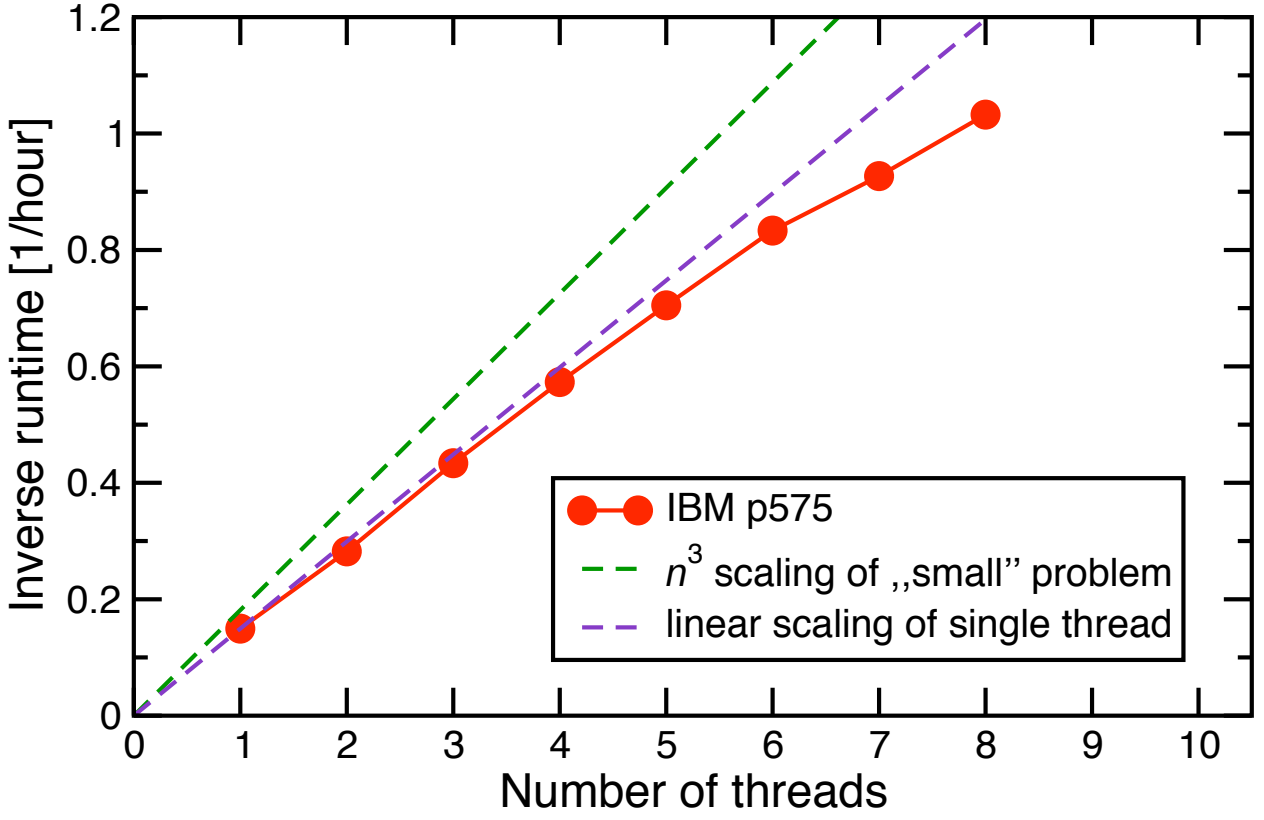
- **our implementation:**
  - real time for Householder step: 14.18 hours
  - main memory: 13.4GByte resident
  - tridiagonal problem (`dsterf`): 91 seconds
- **IBM parallel ESSL 3.3, `pzheevx`**
  - real time for diagonalization: 48.17 hours
  - main memory: 26GByte allocated (?), 14GByte resident

Scaling: first 1000 iterations



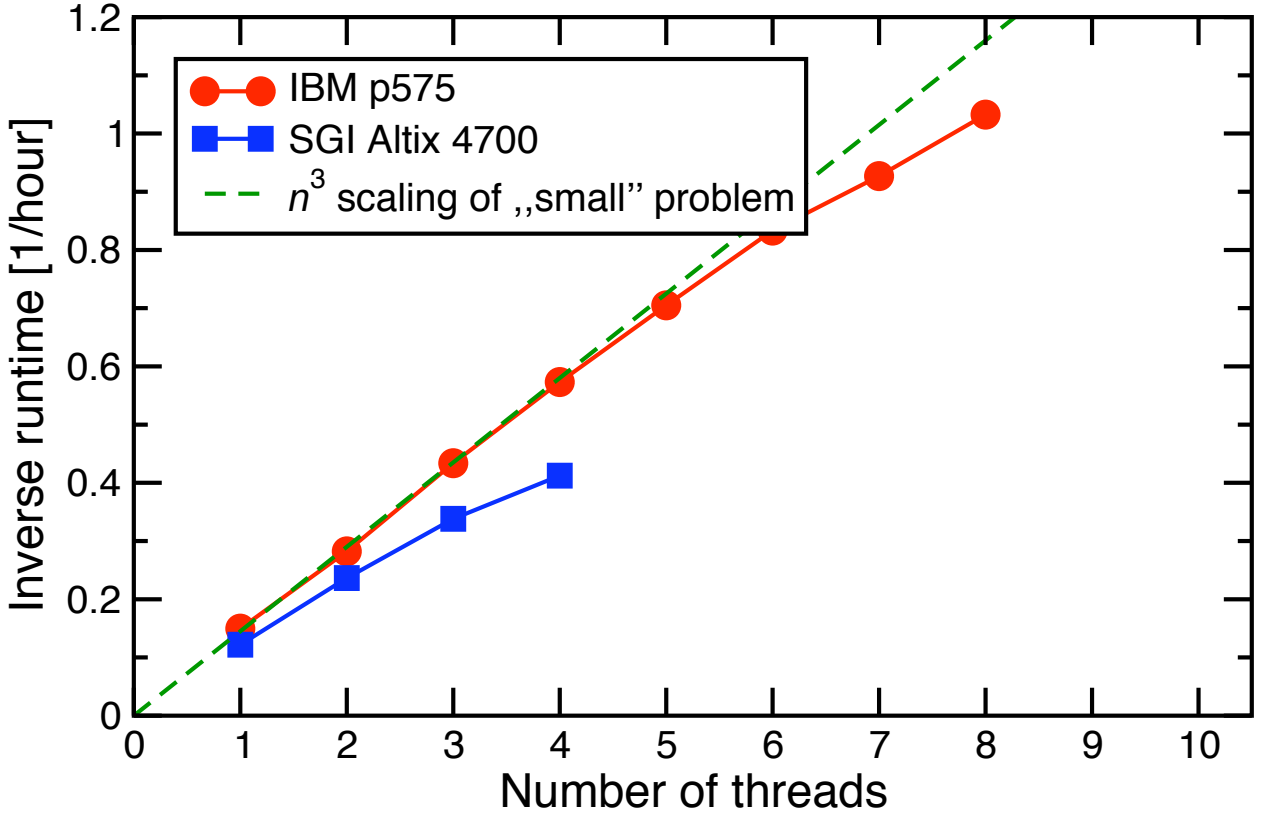
☞ **good scaling with problem size**

Scaling: first 1000 iterations



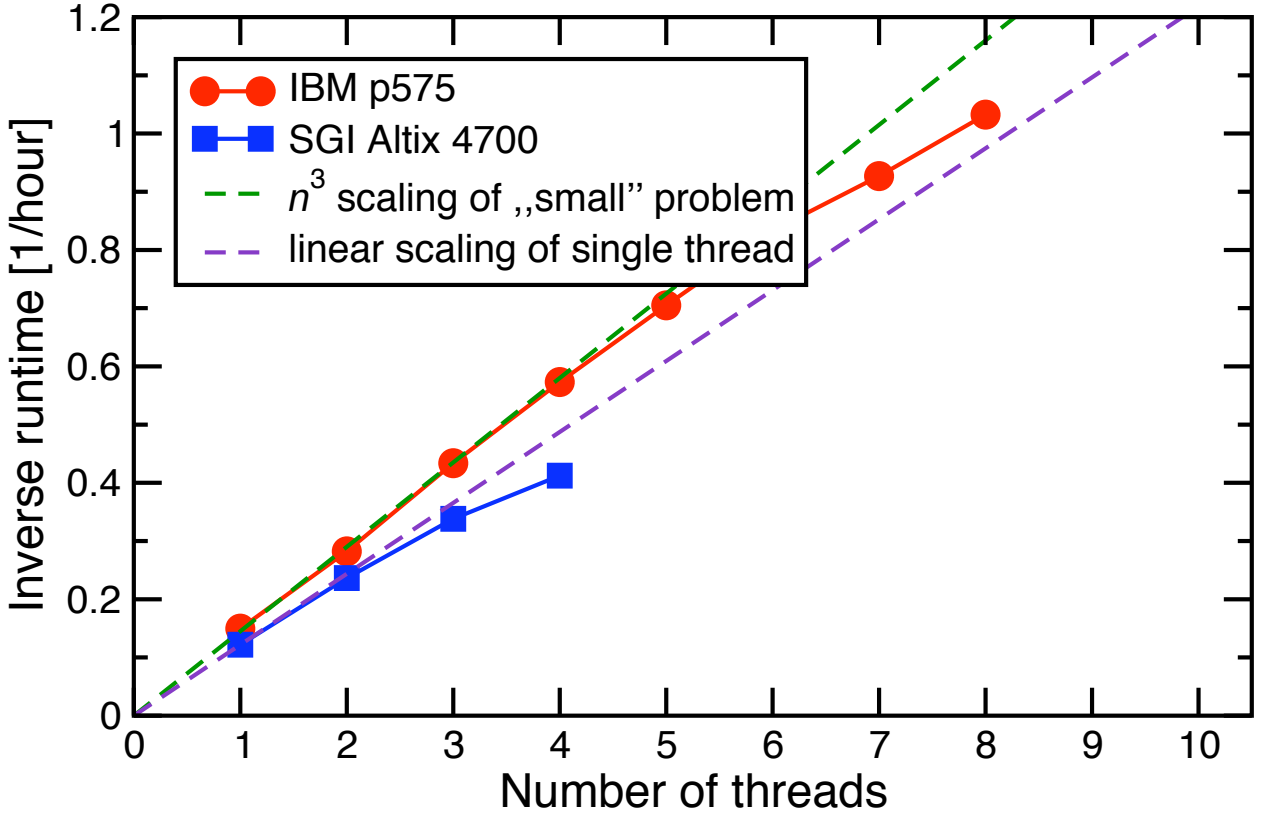
☞ **good parallel scaling on IBM p595**

Scaling: first 1000 iterations



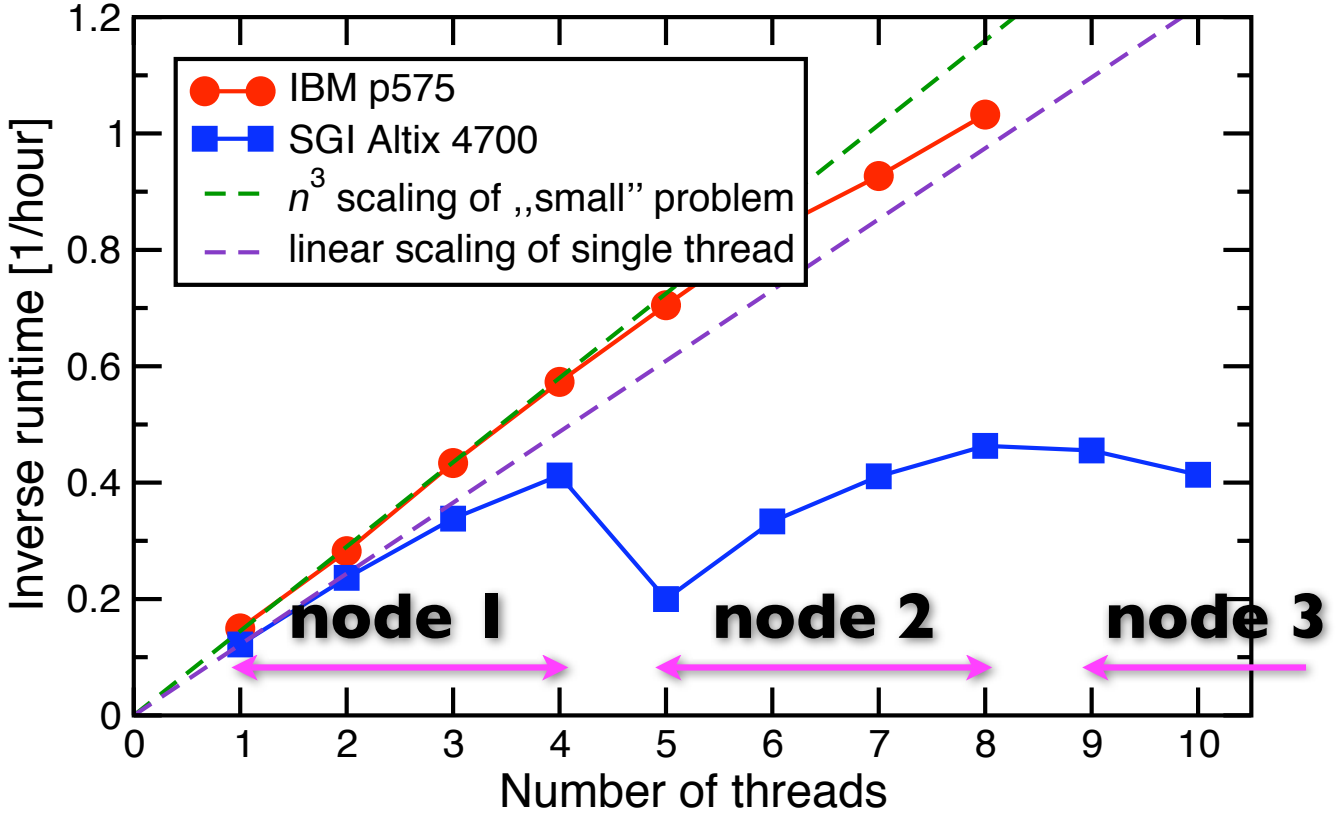
☞ good scaling with problem size on SGI Altix 4700

Scaling: first 1000 iterations



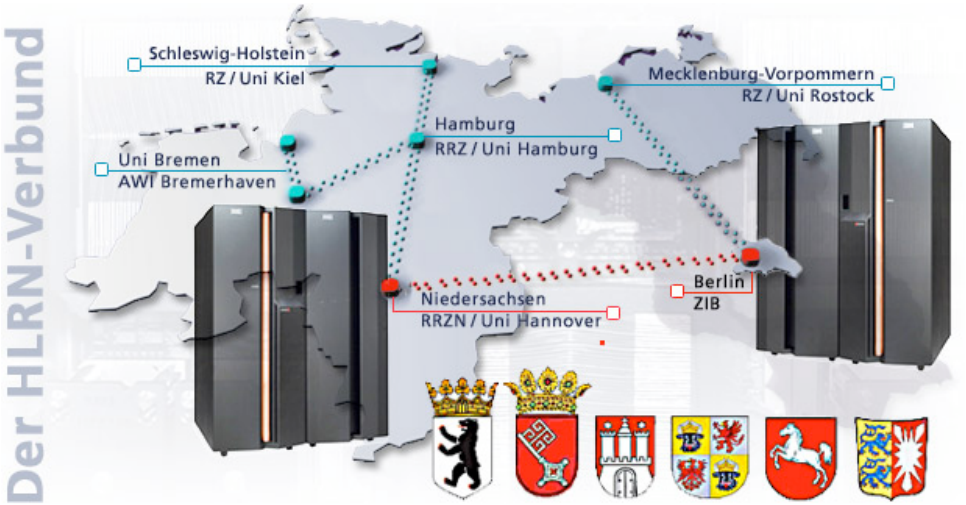
☞ reasonable parallel scaling on SGI Altix 4700 (1 node)

Scaling: first 1000 iterations



☞ works with distributed memory on SGI Altix 4700

All eigenvalues: current record



- **complex dimension *n*=162243**
- initially: 197GByte main memory (matrix)
- 400GByte hard disc (fail-safe checkpoint)
- 1 node with 32 × 1.3GHz Power4 CPUs
- total CPU time: ≈21000 hours

Discussion

- **goal:** stand-alone **public-domain parallel diagonalization package**
- **tridiagonal problem: QR/QL algorithm**
  - effort negligible for eigenvalues only
  - effort <50% for eigenvectors
  - eigenvectors: parallelization using delayed execution
  - need to work on numerical performance (→ LAPACK)
  - efforts by other groups? Cuppen, Dongarra, Dhillon, Parlett, ...
- **real variants** easy to derive (performance?)

Conclusions

for more details see

http://www.theorie.physik.uni-goettingen.de/~honecker/householder/

Thanks

Deutsche
Forschungsgemeinschaft

DFG

Heisenberg-Programm