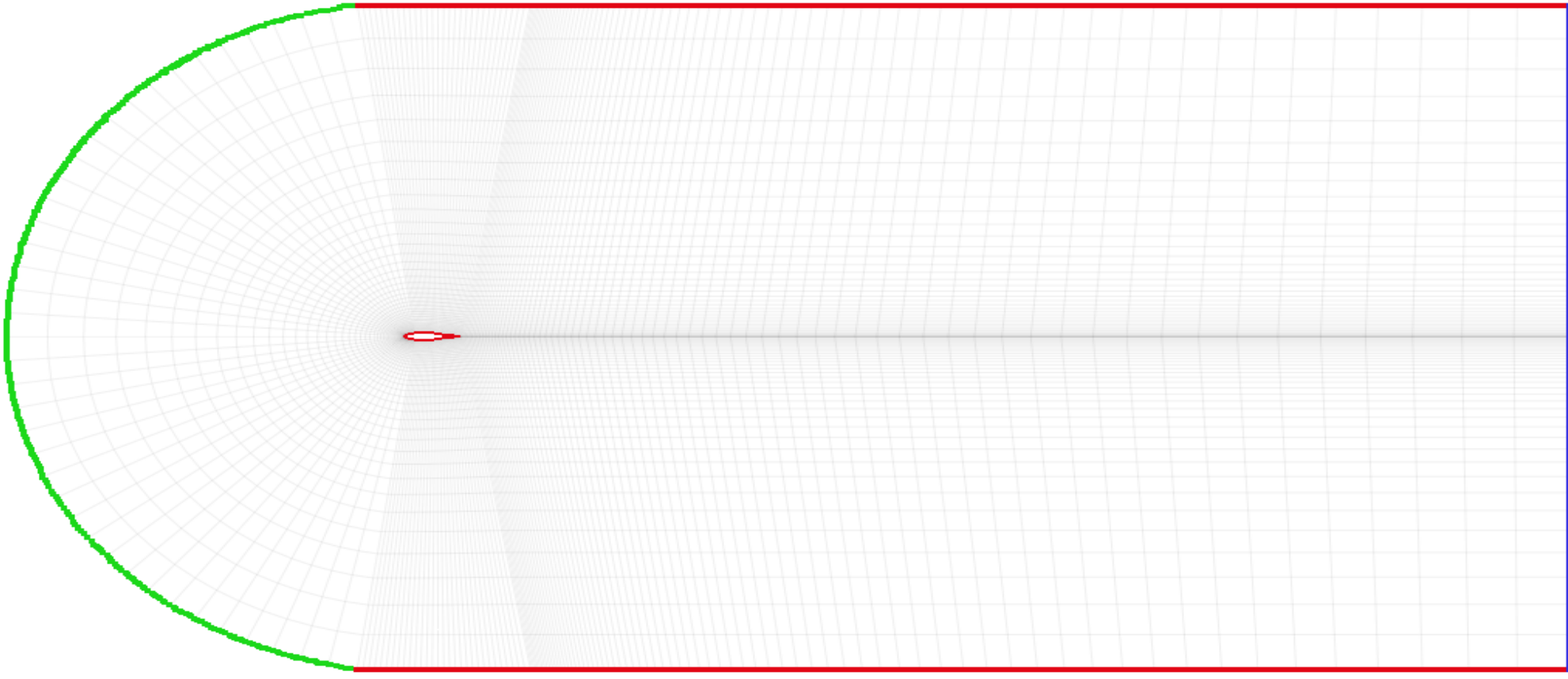
ulm university universität ulm



# Gauss Seidel und CAFFA

## Scientific Computing

Hoffmann & Leidenberger |18. Mai 2007 |
Fakultät für Mathematik und Wirtschaftswissenschaften
Universität Ulm

## Gauss-Seidel

- Poisson Problem 2D
- Gebietsaufteilung
- Implementierung
- Laufzeitvergleich

## Fotran

- Probleme
- BiCG

# Überblick

- Gauss Seidel

# Überblick

- Gauss Seidel
  - Poisson Problem in 2D

# Überblick

- Gauss Seidel
  - Poisson Problem in 2D
  - Gebietsaufteilung

# Überblick

- Gauss Seidel
  - Poisson Problem in 2D
  - Gebietsaufteilung
  - Implementierung

# Überblick

- Gauss Seidel
  - Poisson Problem in 2D
  - Gebietsaufteilung
  - Implementierung
  - Laufzeitvergleich

# Überblick

- Gauss Seidel
  - Poisson Problem in 2D
  - Gebietsaufteilung
  - Implementierung
  - Laufzeitvergleich
- Fortran 90

# Überblick

- Gauss Seidel
  - Poisson Problem in 2D
  - Gebietsaufteilung
  - Implementierung
  - Laufzeitvergleich
- Fortran 90
  - Probleme

# Überblick

- Gauss Seidel
  - Poisson Problem in 2D
  - Gebietsaufteilung
  - Implementierung
  - Laufzeitvergleich
- Fortran 90
  - Probleme
  - BiCG

# Überblick

- Gauss Seidel
  - Poisson Problem in 2D
  - Gebietsaufteilung
  - Implementierung
  - Laufzeitvergleich
- Fortran 90
  - Probleme
  - BiCG
  - BiCG OpenMP

# Überblick

- Gauss Seidel
  - Poisson Problem in 2D
  - Gebietsaufteilung
  - Implementierung
  - Laufzeitvergleich
- Fortran 90
  - Probleme
  - BiCG
  - BiCG OpenMP
  - BiCG MPI

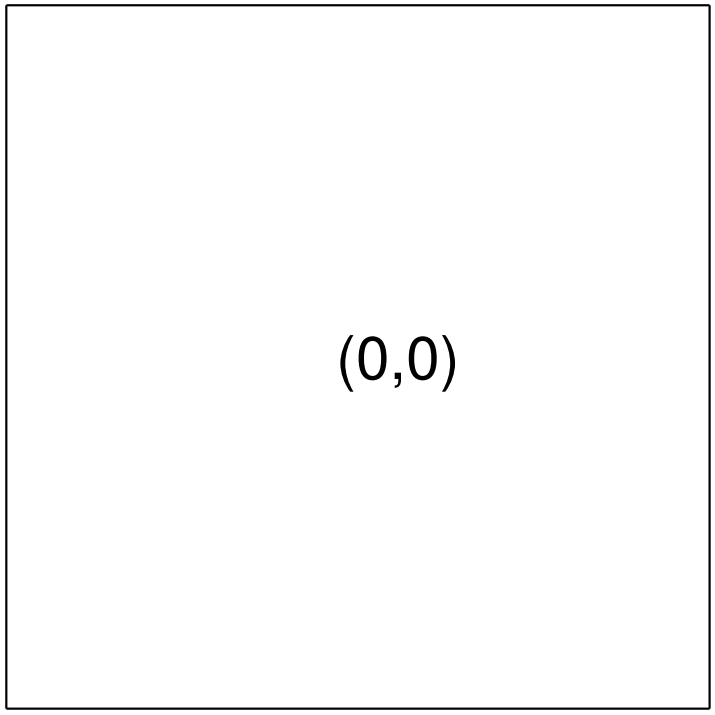
# Poisson Problem 2D

## Problem

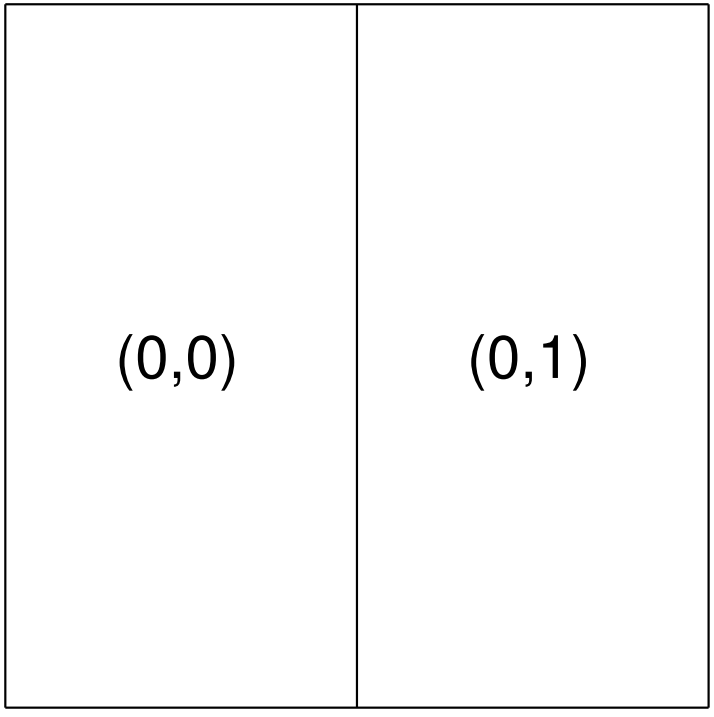
Poisson Problem mit Neumann-Randbedingungen

$$\begin{aligned} -\Delta u &= 2x + 2y - 2 \\ \frac{\partial u}{\partial x}(0, y) &= \frac{\partial u}{\partial x}(1, y) = 0 \\ \frac{\partial u}{\partial y}(x, 0) &= \frac{\partial u}{\partial y}(x, 1) = 0 \end{aligned} \tag{1}$$

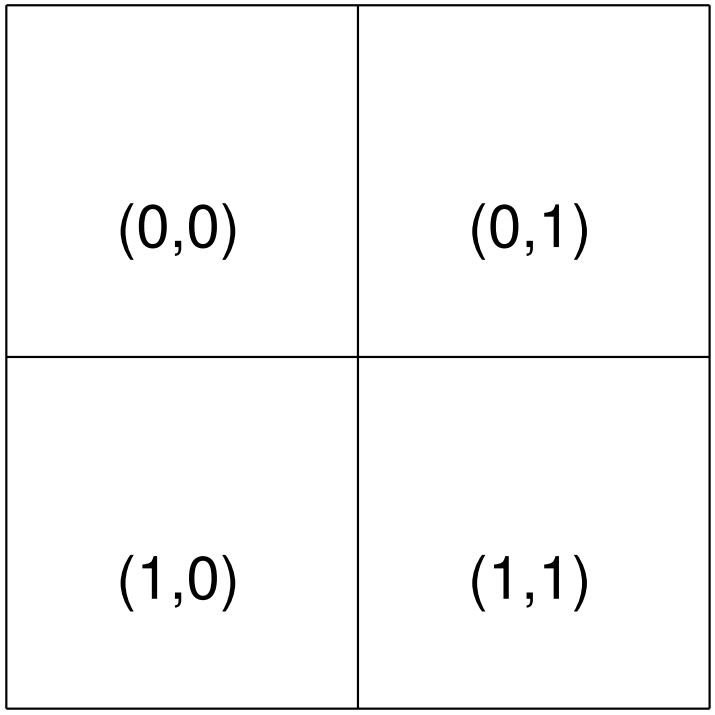
# Gebietsaufteilung MPI-Cart(1/3)



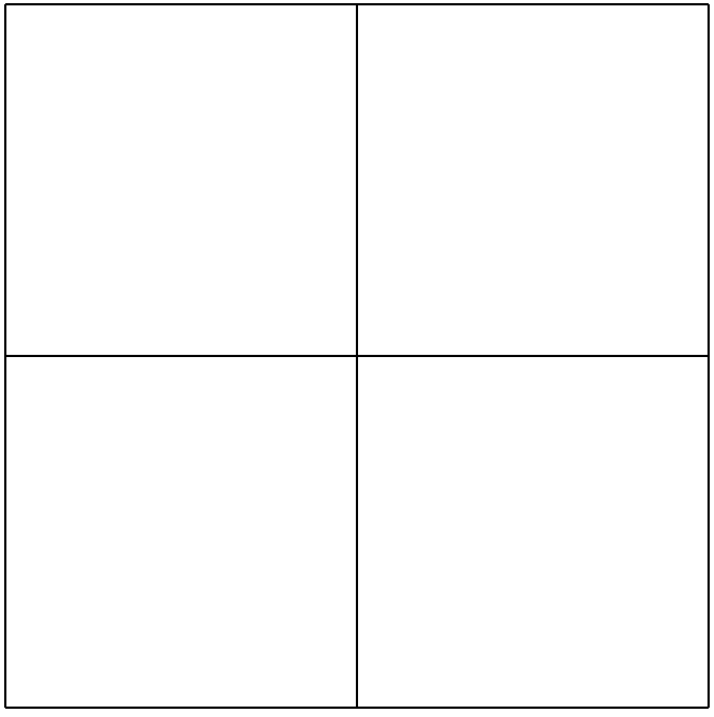
# Gebietsaufteilung MPI-Cart(1/3)



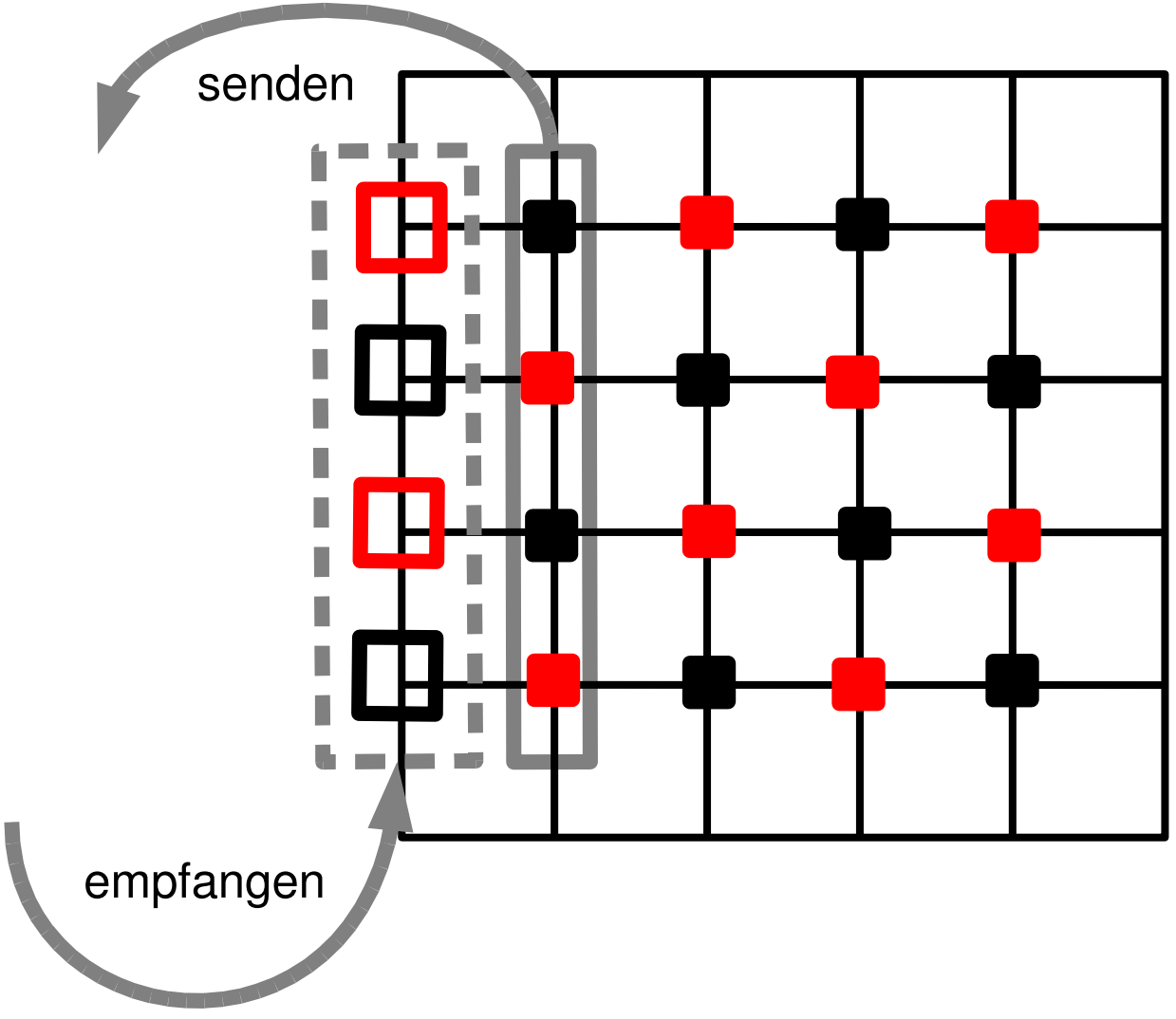
# Gebietsaufteilung MPI-Cart(1/3)



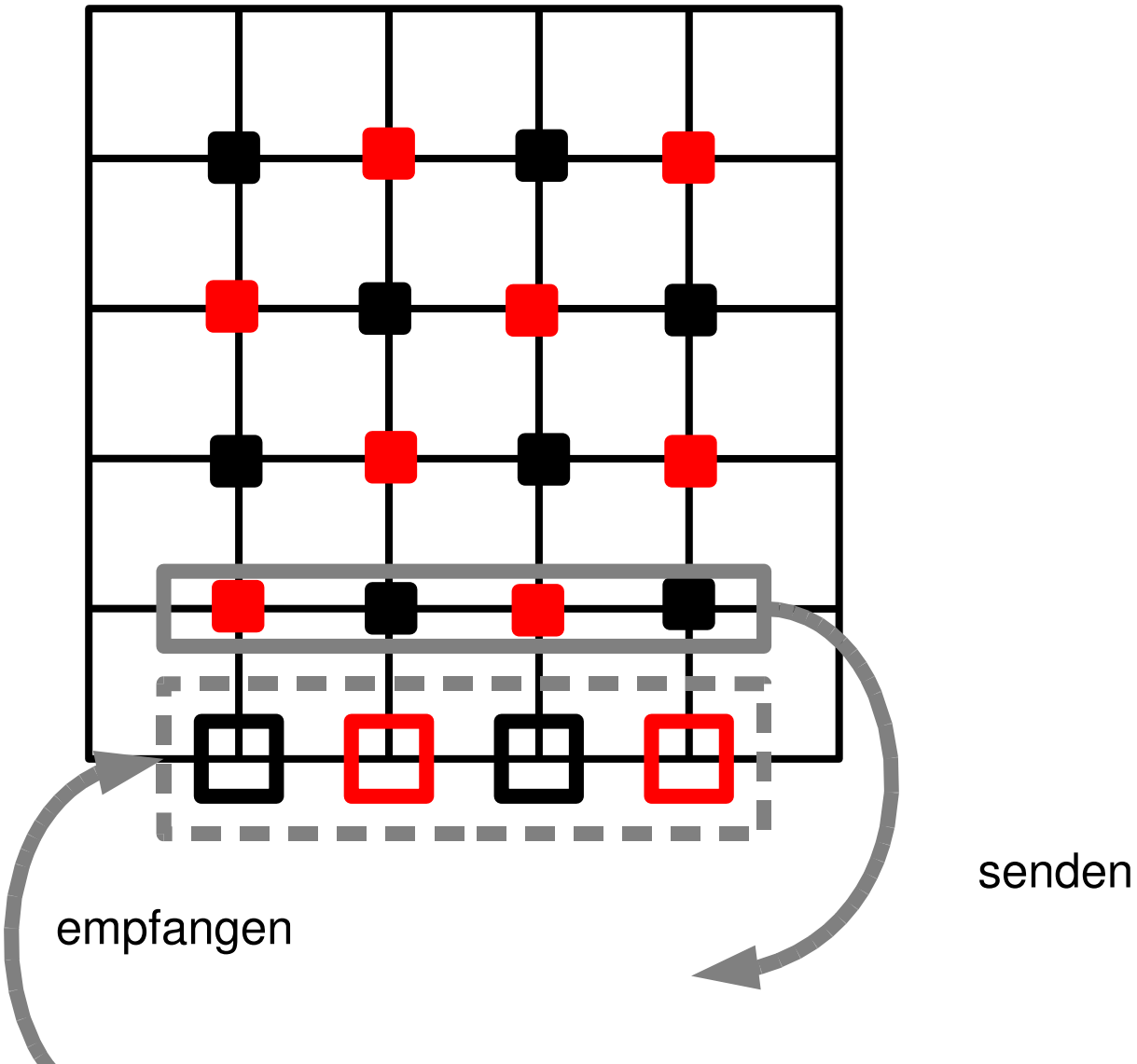
# Gebietsaufteilung MPI-Cart(1/3)



# Gebietsaufteilung (2/3)



# Gebietsaufteilung (3/3)



# Implementierung (1/11)

```
1   enum CartDir { North=0, East=1, South=2, West=3};
2
3  struct MpiCart
4  {
5    MpiCart(){}
6
7    //MpiCart(int _numRows, int _numCols)
8
9    //MpiCart(const MpiCart &rhs)
10
11   //int cartRank(int row, int col)
12
13   MPI::Cartcomm comm;
14   int numRows, numCols;
15   int row, col;
16   int rank;
17   int neighbors[4];
18 };
```

# Implementierung (2/11)

```
1  MpiCart(int _numRows, int _numCols)
2    : numRows(_numRows), numCols(_numCols){
3    int  dims[2];  bool periods[2];
4
5     // create mpi_cart:  2-dimensional, 2 x 2,non-periodic
6    dims[0]    = numRows; dims[1]    = numCols;
7    periods[0] = false;   periods[1] = false;
8    comm = MPI::COMM_WORLD.Create_cart(2,dims,periods,true);
9
10   // find your position in the cart
11   rank = MPI::COMM_WORLD.Get_rank();
12   int coords[2];
13   comm.Get_coords(rank, 2, coords);
14   row = coords[0]; col = coords[1];
15
16   // find your neighbors
17   comm.Shift(0, 1, neighbors[North], neighbors[South]);
18   comm.Shift(1, 1, neighbors[West],  neighbors[East]);
19 }
```

# Implementierung (3/11)

```
1  MpiCart(const MpiCart &rhs)
2   : comm(rhs.comm), numRows(rhs.numRows),
3   numCols(rhs.numCols), row(rhs.row), col(rhs.col)
4  {
5    for (int k=0; k<8; ++k) {
6      neighbors[k] = rhs.neighbors[k];
7    }
8  }
9
10 int cartRank(int row, int col) {
11   if ((row<0) || (col<0) || (row==numRows)
12                  || (col==numCols)) {
13     return MPI_PROC_NULL;
14   }
15   int coords[2];
16   coords[0] = row;
17   coords[1] = col;
18   return comm.Get_cart_rank(coords);
19 }
```

# Implementierung (4/11)

```
1  struct DistributedVector {
2    typedef GeMatrix<FullStorage<double, RowMajor> > Grid;
3
4    //  DistributedVector(){}
5
6    //  DistributedVector(const MpiCart &_mpiCart, int rh)
7
8    // updateBoundary()
9
10         MpiCart          mpiCart;
11         int              N;
12         int              m, n;
13         int              m0, n0;
14         int              firstRow, firstCol;
15         Grid             grid;
16         MPI::Datatype    MpiRow, MpiCol;
17     };
```

# Implementierung (5/11)

```
1   DistributedVector(){}
2
3   DistributedVector(const MpiCart &_mpiCart, int rh)
4      : mpiCart(_mpiCart), N(rh-1)
5   {
6     // global matrix has size (N+1) x (N+1)
7     // compute m and n, such that local matrix has size (-1,m+1) x (-1,n+1)
8     m = (N+1) / mpiCart.numRows - 1;
9     n = (N+1) / mpiCart.numCols - 1;
10    grid.resize(_(-1,m+1),_(-1,n+1));
11
12    m0 = (m+1)*mpiCart.row;//row offset
13    n0 = (n+1)*mpiCart.col;//col offset
14
15    firstRow = (mpiCart.row==0) ? 1 : 0; //first non-zero value
16    firstCol = (mpiCart.col==0) ? 1 : 0; //first non-zero value
17
18    // types to send rows and cols
19    MpiRow = MPI::DOUBLE.Create_contiguous(n+2);
20    MpiRow.Commit();
21    MpiCol = MPI::DOUBLE.Create_vector(m+1, 1, grid.leadingDimension());
22    MpiCol.Commit();
23  }
```

# Implementierung (6/11)

```
1  void
2  updateBoundary()
3  {
4    MPI::Request req[8];
5    MPI::Status  stat[8];
6
7    req[0] = mpiCart.comm.Isend(&grid(0,n), 1, MpiCol, mpiCart.neighbors[East],  0);
8    req[1] = mpiCart.comm.Isend(&grid(m,0), 1, MpiRow, mpiCart.neighbors[South], 0);
9    req[2] = mpiCart.comm.Isend(&grid(0,0), 1, MpiCol, mpiCart.neighbors[West],  0);
10   req[3] = mpiCart.comm.Isend(&grid(0,0), 1, MpiRow, mpiCart.neighbors[North], 0);
11
12   req[4] = mpiCart.comm.Irecv(&grid(  0,n+1), 1, MpiCol, mpiCart.neighbors[East],  0);
13   req[5] = mpiCart.comm.Irecv(&grid(m+1,  0), 1, MpiRow, mpiCart.neighbors[South], 0);
14   req[6] = mpiCart.comm.Irecv(&grid(  0, -1), 1, MpiCol, mpiCart.neighbors[West],  0);
15   req[7] = mpiCart.comm.Irecv(&grid( -1,  0), 1, MpiRow, mpiCart.neighbors[North], 0);
16
17   MPI::Request::Waitall(8, req, stat);
18 }
```

# Implementierung (7/11)

```
1   int size = MPI::COMM_WORLD.Get_size();
2   int p1, p2;
3   t2 = clock();
4   if(size==1) {
5       p1 = 1;
6       p2 = 1;
7   }else if(size==2){
8       p1 = 2;
9       p2 = 1;
10  }else{
11      p1 = static_cast<int>(floor(
12          sqrt(static_cast<double>(size))));
13
14      if(p1%2!=0){
15          p1 -= 1;
16      }
17      p2 = size/p1;
18  }
```

# Implementierung (8/11)

```
1  void gaussSeidelRedBlack(double omega,
2       const DistributedVector &f, DistributedVector &u) {
3          const DistributedVector::Grid &F = f.grid;
4          DistributedVector::Grid       &U = u.grid;
5          double c   = omega/4.;
6          double hh  = 1./((u.N+1)*(u.N+1));
7          // ghost nodes ...
8          // red nodes ...
9          u.updateBoundary();
10         // black nodes ...
11         u.updateBoundary();
12         // ghost nodes ...
13         int    rhh = (u.N+1)*(u.N+1);
14         double s   = sum(u)/rhh;
15         for (int i=0; i<=u.m; ++i) {
16          for (int j=0; j<=u.n; ++j) {
17              U(i,j) -= s;
18          } }
19     }
```

# Implementierung (9/11)

```
1  if (u.mpiCart.row==0) {
2      U(-1,_) = U(0,_);
3  }
4  if (u.mpiCart.row==u.mpiCart.numRows-1) {
5      U(u.m+1,_) = U(u.m,_);
6  }
7  if (u.mpiCart.col==0) {
8      U(_,-1) = U(_,0);
9  }
10 if (u.mpiCart.col==u.mpiCart.numCols-1) {
11     U(_,u.n+1) = U(_,u.n);
12 }
```

# Implementierung (10/11)

```
1  // red nodes
2  for (int i=0; i<=u.m; ++i) {
3      for (int j=0; j<=u.n; j += 2) {
4          U(i,j) = (1-omega)*U(i,j)
5                  + c*(F(i,j)*hh + U(i-1,j)
6                  + U(i+1,j) + U(i,j-1) + U(i,j+1));
7      }
8      ++i;
9      if(i<=u.m) {
10         for (int j=1; j<=u.n; j += 2) {
11             U(i,j) = (1-omega)*U(i,j)
12                    + c*(F(i,j)*hh + U(i-1,j)
13                    + U(i+1,j) + U(i,j-1)
14                    + U(i,j+1));
15         }
16     }
17 }
```

# Implementierung (11/11)

```
1  // black nodes
2  for (int i=0; i<=u.m; ++i) {
3      for (int j=1; j<=u.n; j += 2) {
4          U(i,j) = (1-omega)*U(i,j)
5                  + c*(F(i,j)*hh + U(i-1,j)
6                  + U(i+1,j) + U(i,j-1) + U(i,j+1));
7      }
8      ++i;
9      if(i<=u.m) {
10         for (int j=0; j<=u.n; j += 2) {
11             U(i,j) = (1-omega)*U(i,j)
12                    + c*(F(i,j)*hh + U(i-1,j)
13                    + U(i+1,j) + U(i,j-1)
14                    + U(i,j+1));
15         }
16     }
17 }
```

# Laufzeitvergleich (1/10)

Laufzeitvergleich für das parallelisierte Gauss-Seidel, auf der Pacioli.

1. Beispiel für 16.769.025 Freiheitsgrade und 400 Iterationen
   - Speed-up Faktor
   - Zeit pro CPU über Anzahl der Prozessoren
   - Zeit pro CPU über Anzahl der Prozessoren auf einer doppel-log-Skala
2. Beispiel für 67.092.481 Freiheitsgrade und 200 Iterationen
   - Speed-up Faktor
   - Zeit pro CPU über Anzahl der Prozessoren
   - Zeit pro CPU über Anzahl der Prozessoren auf einer doppel-log-Skala

Vergleich von beiden Beispielen

- Speed-up Faktor
- Zeit pro CPU über Anzahl der Prozessoren
- Zeit pro CPU über Anzahl der Prozessoren auf einer doppel-log-Skala

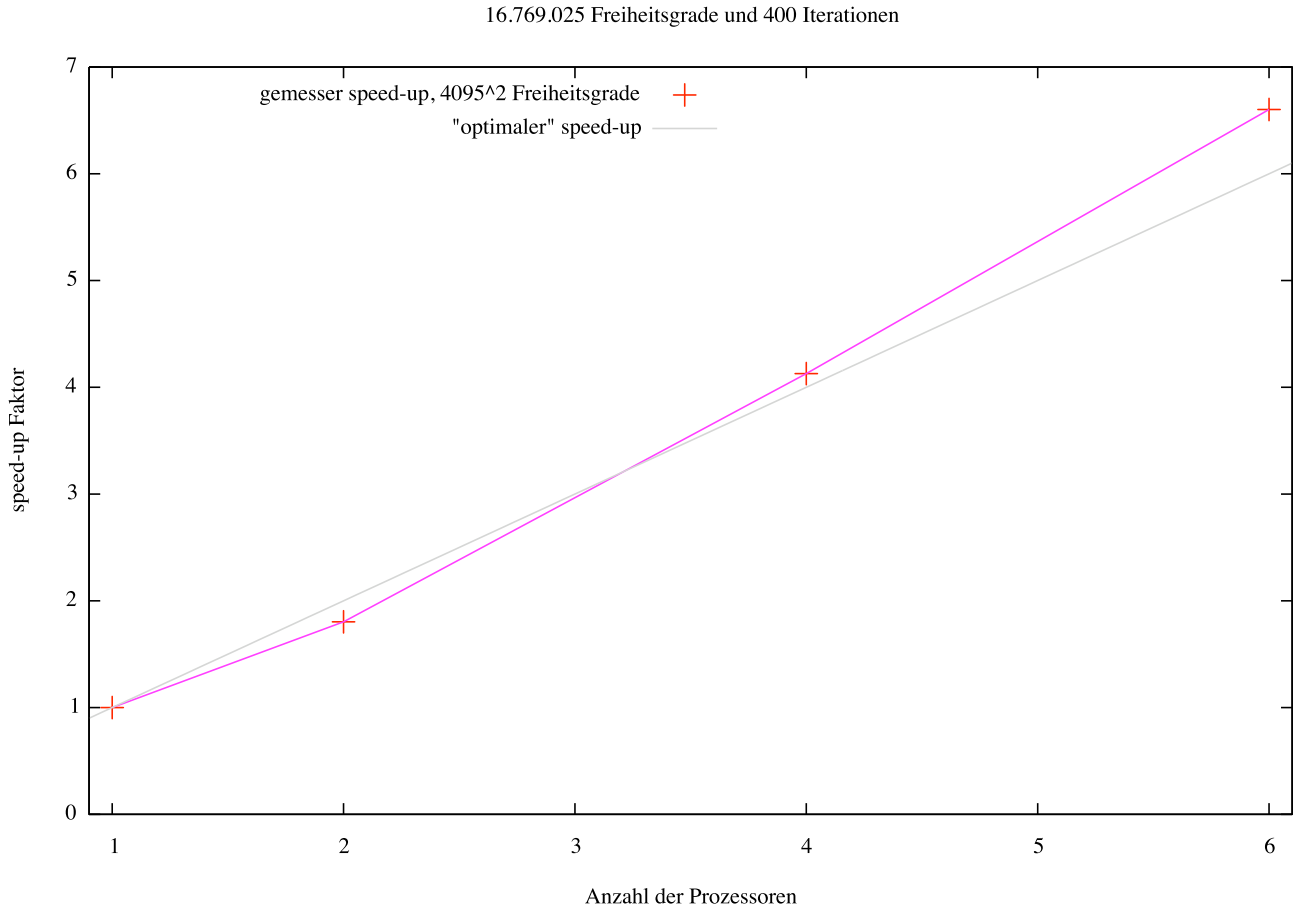
# Laufzeitvergleich 16.769.025 Freiheitsgrade (2/10)

Beispiel 1:

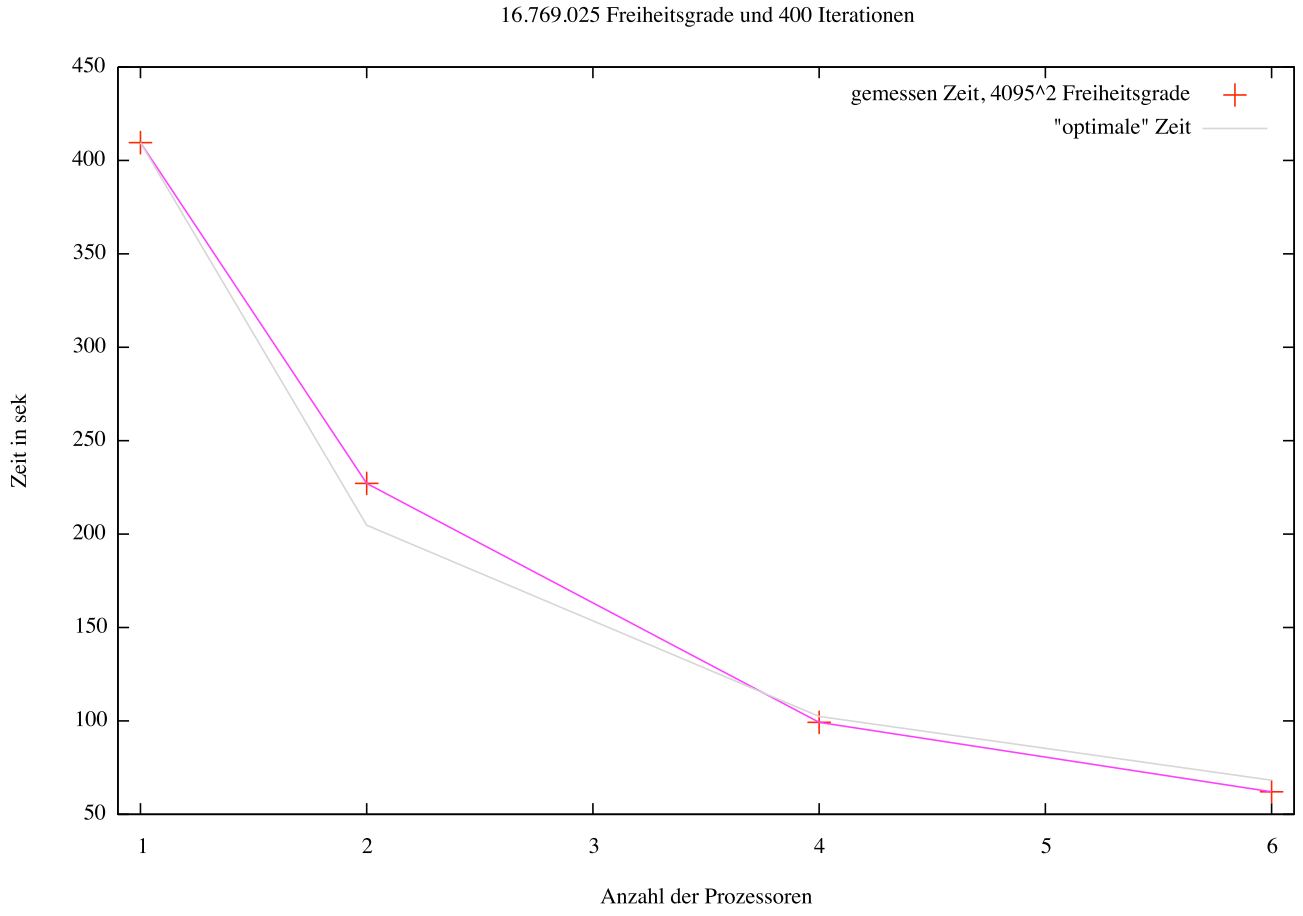
- 16.769.025 Freiheitsgrade
- 400 Iterationen

| # Prozessoren | Laufzeit [sek] | Laufzeit [min,sek] | Speed-up Faktor |
|---|---|---|---|
| 1 | 419.51 | 6.49 | 1 |
| 2 | 227.13 | 3.47 | 1.803 |
| 4 | 99.21 | 1.39 | 4.128 |
| 6 | 62.03 | 1.02 | 6.602 |

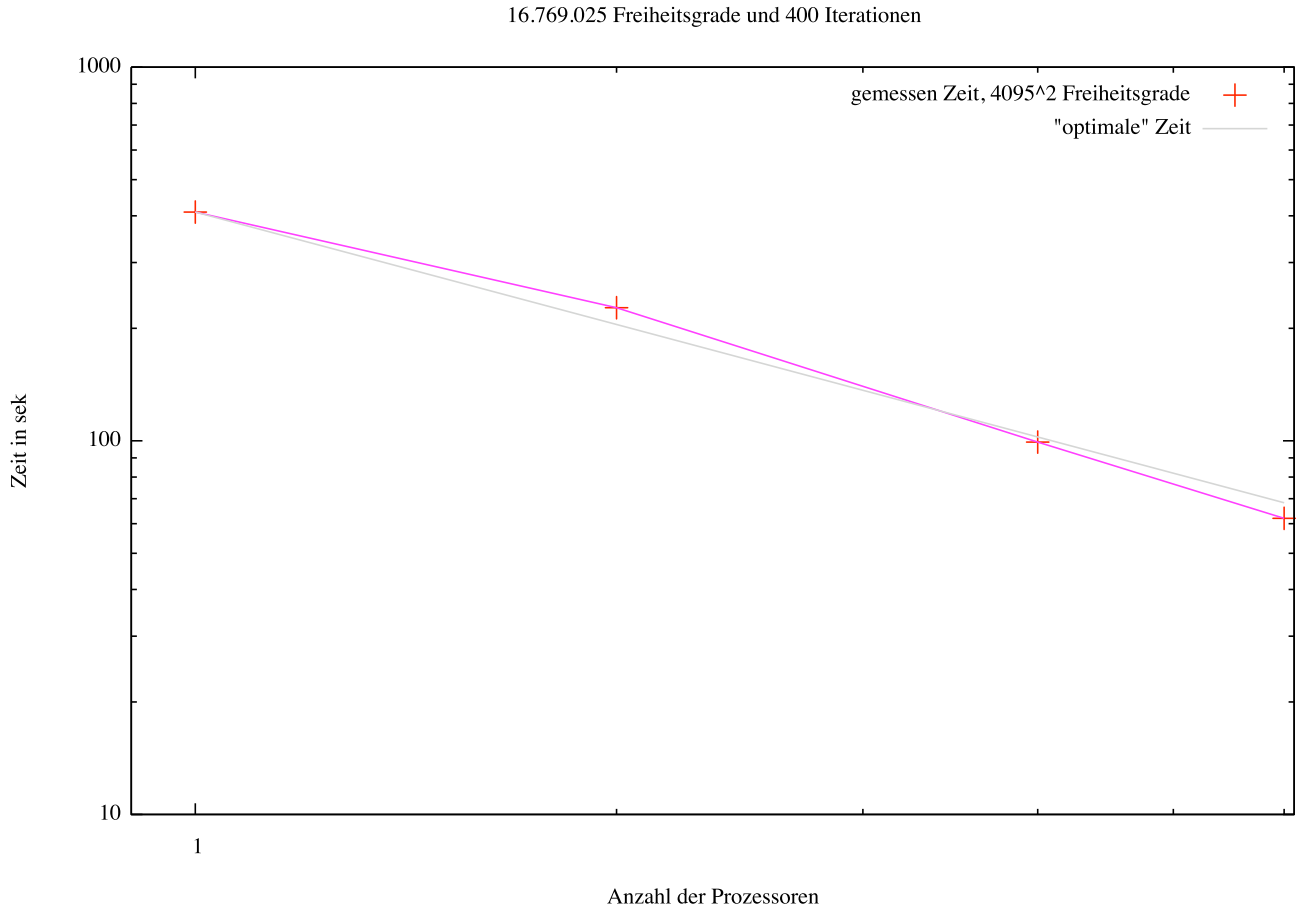
# Laufzeitvergleich 16.769.025 Freiheitsgrade (3/10)



# Laufzeitvergleich 16.769.025 Freiheitsgrade (4/10)



# Laufzeitvergleich 16.769.025 Freiheitsgrade (5/10)




Eine doppelt-log Skala bietet sich an da:[1]

$$y = \frac{c}{x}$$

$$\log(y) = \log(\frac{c}{x}) = \log(c) - \log(x)$$

[1] Dank an Christian Pape

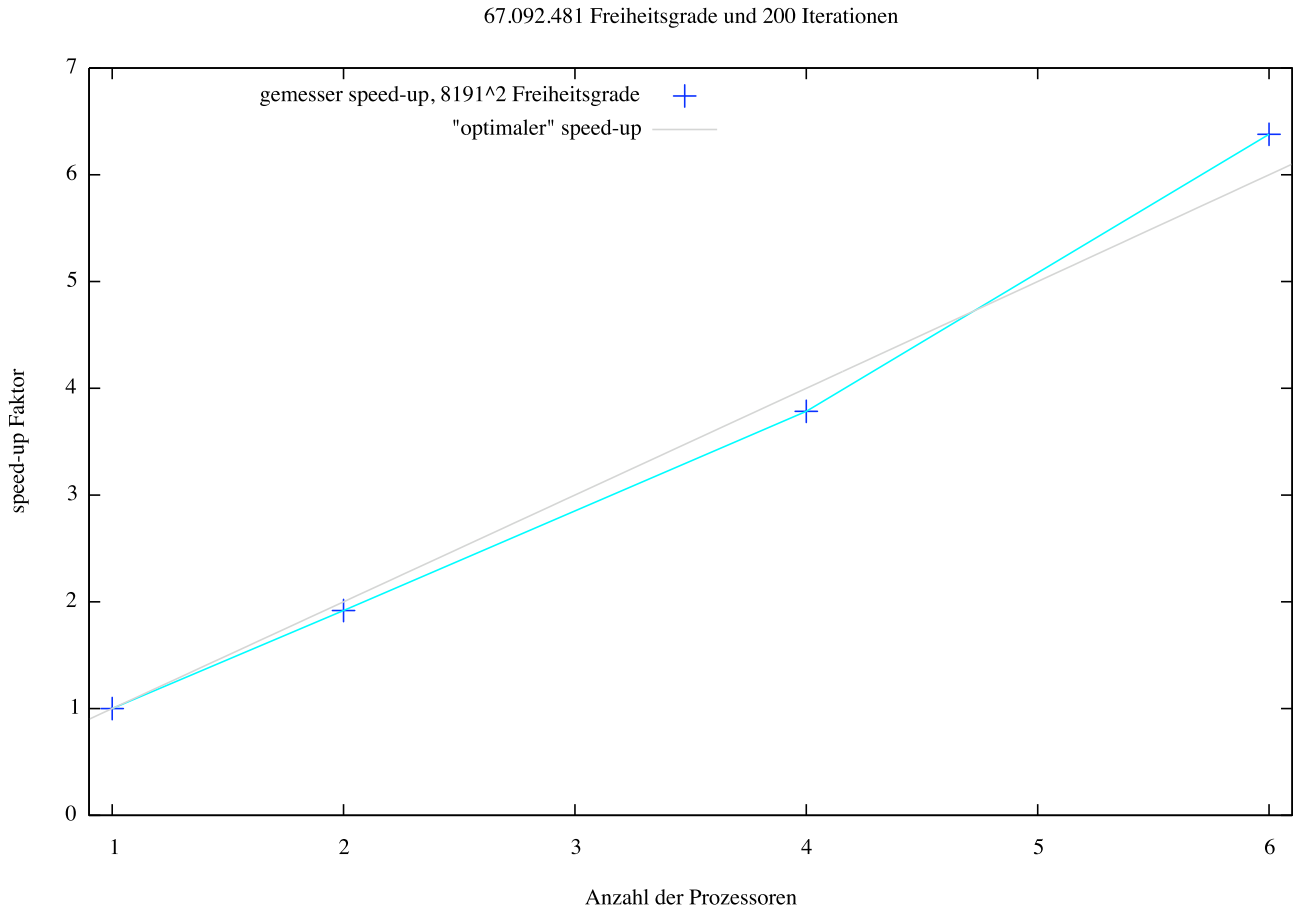
# Laufzeitvergleich 16.769.025 Freiheitsgrade (6/10)

Beispiel 2:

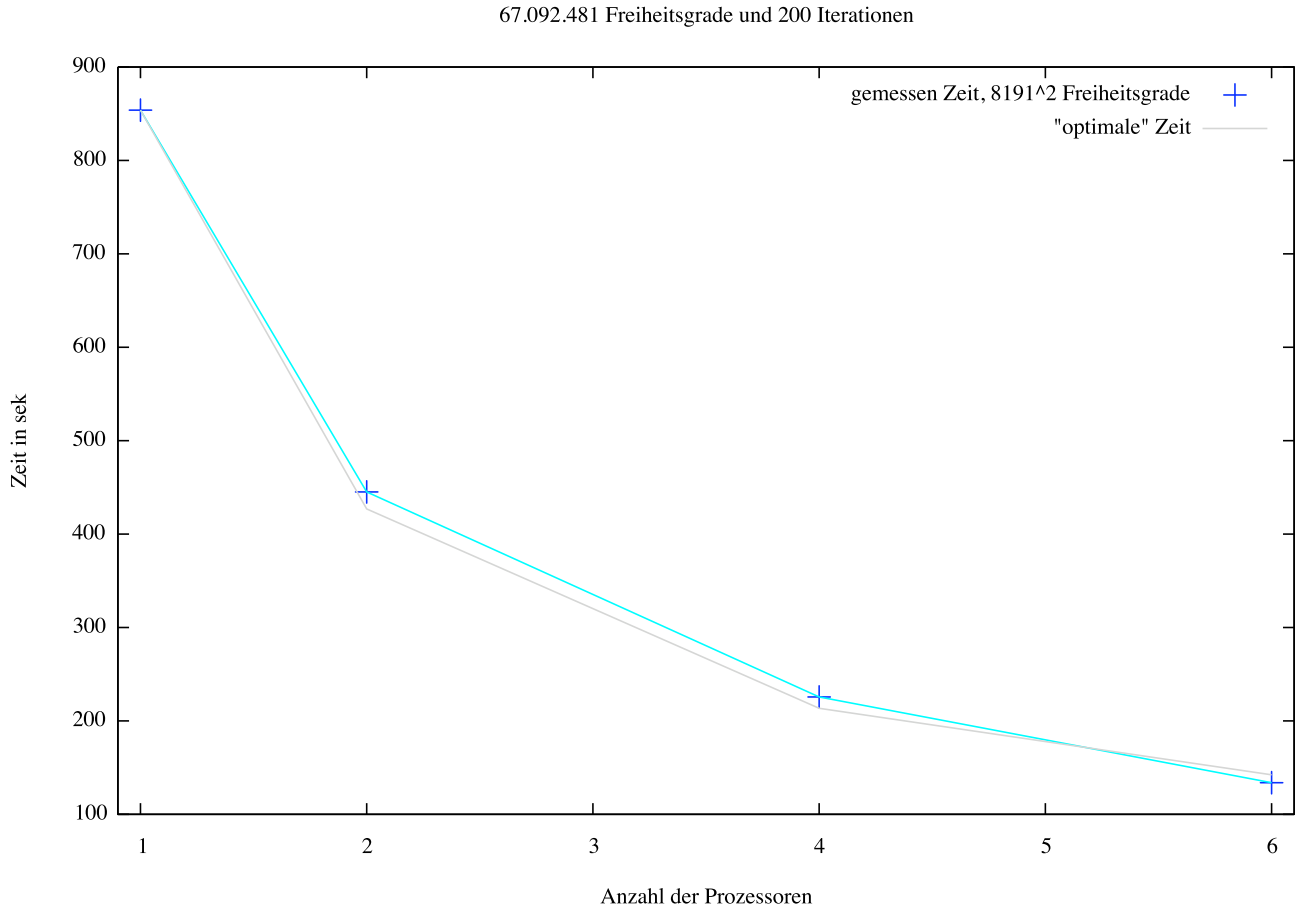
- 67.092.481 Freiheitsgrade
- 200 Iterationen

| # Prozessoren | Laufzeit [sek] | Laufzeit [min,sek] | Speed-up Faktor |
|---|---|---|---|
| 1 | 853.86 | 14.13 | 1 |
| 2 | 445.10 | 7.25 | 1.918 |
| 4 | 225.68 | 3.45 | 3.784 |
| 6 | 133.85 | 2.13 | 6.379 |

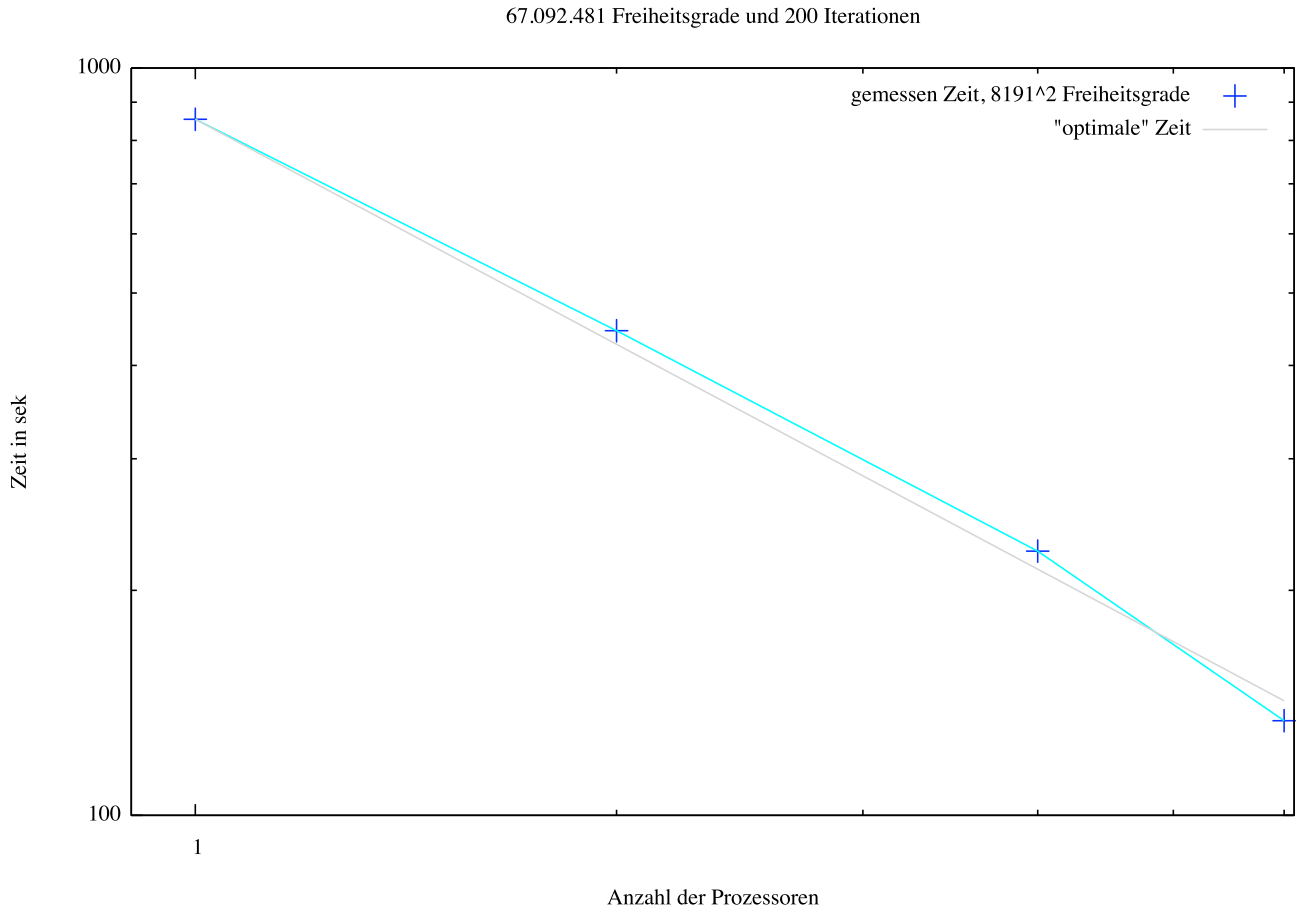
# Laufzeitvergleich 67.092.481 Freiheitsgrade (7/10)



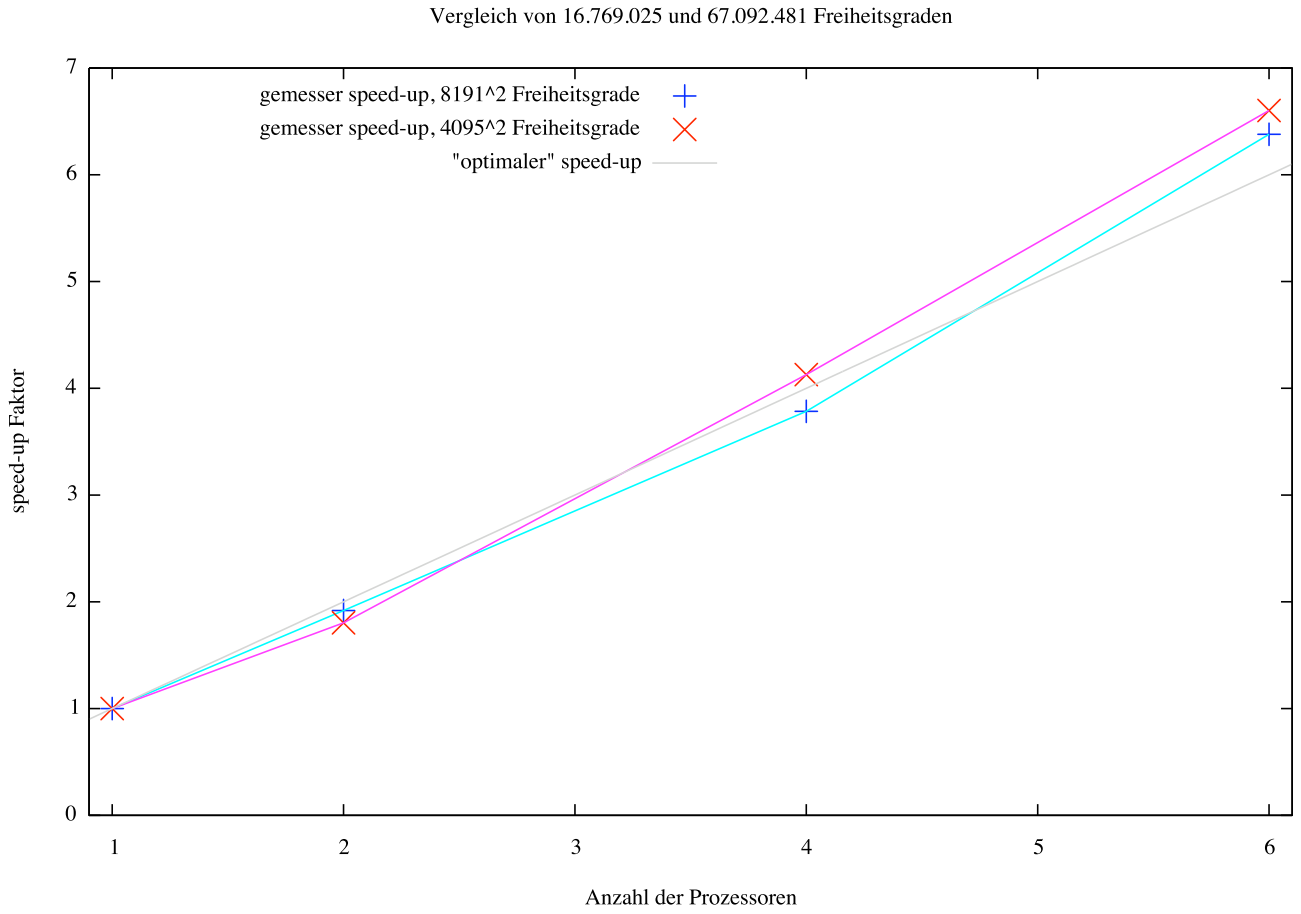
# Laufzeitvergleich 67.092.481 Freiheitsgrade (8/10)



# Laufzeitvergleich 67.092.481 Freiheitsgrade (9/10)



# Laufzeitvergleich Vergleich (10/10)



# Probleme

- OpenMP: mit Batch-System deutlich langsamer ????

# Probleme

- OpenMP: mit Batch-System deutlich langsamer ????
- Cluster: *mpif90* funktioniert nicht

# BiCG

- Wähle $x_0 \in \mathbb{R}^n,\ \varepsilon > 0$
- $r_0 = r_0^* = p_0 = p_0^* := b - Ax_0$ j=0
- while($\|r_j\| > \varepsilon$)
  - $\alpha_j := \frac{<r_j, r_j^*>_2}{<Ap_j, p_j^*>_2}$
  - $x_{j+1} := x_j + \alpha_j p_j$
  - $r_{j+1} := r_j - \alpha_j A p_j$
  - $r_{j+1}^* := r_j^* - \alpha_j A p_j^*$
  - $\beta_j := \frac{<r_{j+1}, r_{j+1}^*>_2}{<r_j, r_j^*>_2}$
  - $p_{j+1} := r_{j+1} - \beta_j p_j$
  - $p_{j+1}^* := r_{j+1}^* - \beta_j p_j^*$
  - j++
- end

# BiCG OpenMP

```
1  program bicgtest
2
3  integer, parameter :: length = 400
4  real(kind(1.d0))   :: tol
5  real(kind(1.d0))   :: A(length,length)
6  real(kind(1.d0))   :: x(length),b(length)
7  integer            :: n_p_sec, ia, ie, i,j
8
9  call system_clock(count_rate=n_p_sec)!
10 call initMatrixandVectors(length, A, x, b)
11 tol = 0.001
12
13 call system_clock(count=ia)
14 call bicg(length,tol,A,b,x)
15 call system_clock(count=ie)
16
17 write(*,*) ((ie-ia)/real(n_p_sec))
18
19 end program bicgtest
```

# BiCG OpenMP

```
1  subroutine initMatrixandVectors(length, A, x, b)
2    integer, intent(in)             :: length
3    real(kind(1.d0)), intent(out) :: A(length,length)
4    real(kind(1.d0)), intent(out) :: x(length),b(length)
5    do i=1,length
6      do j=1,length
7          if(i <= j) then
8              A(i,j) = 1.
9          else
10             A(i,j) = 0.
11         end if
12     end do
13     b(i) = 1.
14     x(i) = 0.
15   end do
16 end subroutine initMatrixandVectors
```

```
1  subroutine bicg(length,tol,A,b,x)
2
3      integer, intent(in)             :: length
4      real(kind(1.d0)), intent(in)    :: tol
5      real(kind(1.d0)), intent(in)    :: A(length,length)
6      real(kind(1.d0)), intent(in)    :: b(length)
7      real(kind(1.d0)), intent(inout) :: x(length)
8      real(kind(1.d0)) :: A_(length,length)
9      real(kind(1.d0)) :: r(length), r_(length)
10     real(kind(1.d0)) :: p(length), p_(length)
11     real(kind(1.d0)) :: tmpV(length), tmpV_(length)
12     real(kind(1.d0)) :: alpha, beta, norm, tmp1, tmp2
13     real(kind(1.d0)) :: dotProd, dotProdOld
14     integer          :: j
15
16     call OMP_SET_NUM_THREADS(2)
17     !$OMP PARALLEL
18     !$OMP SECTIONS
19     !$OMP SECTION
20     A_ = transpose(A)
21     !$OMP SECTION
22     r  = b -matmul(A,x)
23     r_ = r
24     p  = r
25     p_ = r
26     !$OMP END SECTIONS
27     !$OMP END PARALLEL
28
29     dotProdOld = dot_product(r,r_)
30     !
31     !loop...
32     !
33 end subroutine bicg
```

```
1   do j=1,length
2       if(dot_product(r,r) < tol)
3           return
4       !$OMP PARALLEL
5       !$OMP SECTIONS
6       !$OMP SECTION
7       tmpV  = matmul(A,p)
8       !$OMP SECTION
9       tmpV_ = matmul(A_,p_)
10      !$OMP END SECTIONS
11      !$OMP END PARALLEL
12      tmp2  = dot_product(tmpV, p_)
13      alpha = dotProdOld/tmp2
14      !$OMP PARALLEL
15      !$OMP SECTIONS
16      !$OMP SECTION
17      x  = x  + alpha*p
18      !$OMP SECTION
19      r  = r  - alpha*tmpV
20      !$OMP SECTION
21      r_ = r_ - alpha*tmpV_
22      !$OMP END SECTIONS
23      !$OMP END PARALLEL
24      dotProd    = dot_product(r,r_)
25      beta       = dotProd/dotProdOld
26      dotProdOld = dotProd
27      !$OMP PARALLEL
28      !$OMP SECTIONS
29      !$OMP SECTION
30      p  = r  + beta*p
31      !$OMP SECTION
32      p_ = r_ + beta*p_
33      !$OMP END SECTIONS
34      !$OMP END PARALLEL
35  end do
```

# BiCG OpenMP - Laufzeit

- Matrixgröße : 1000x1000
- BiCG - seriell : 7.1327
- BiCG - OpenMP : 5.9519

# BICG MPI

```
1  program bicgtest
2  use mpi
3  implicit none
4  integer :: ierror,rank,mpiSize,gSize,lSize,alloc_status,&
5            dealloc_status, n_p_sec, ia, ie, i,j,&
6            status(MPI_STATUS_SIZE)
7  real(kind(1.d0))    :: tol,t
8  real(kind(1.d0)), allocatable, dimension (:,:) :: A, A_
9  real(kind(1.d0)), allocatable, dimension (:)   :: x, b, sol
10
11     gSize = 100
12     !init stop-time
13     call system_clock(count_rate=n_p_sec)!
14     call system_clock(count=ia)
15
16     !init mpi
17     call MPI_INIT(ierror)
18     call MPI_COMM_SIZE(MPI_COMM_WORLD, mpiSize, ierror)
19     call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
20
21     lSize = gSize/mpiSize
22     tol = 0.001
23
24     allocate(A(lSize,gSize),stat=alloc_status)
25     allocate(A_(lSize,gSize),stat=alloc_status)
26     allocate(x(lSize),stat=alloc_status)
27     allocate(b(lSize),stat=alloc_status)
28     allocate(sol(gSize),stat=alloc_status)
29     !...
```

# BICG MPI

```
1  !...
2   do i=1,lSize
3      do j=1,gSize
4           if(i <= j) then
5               A(i,j) = 1.
6           else
7               A(i,j) = 0.
8           end if
9           if(j <= i) then
10              A_(i,j) = 1.
11          else
12              A_(i,j) = 0.
13          end if
14          sol(i) = 0
15     end do
16     b(i) = 1.
17     x(i) = 0.
18  end do
19
20  call bicg(gSize,lSize,mpiSize,rank,tol,A,A_,b,x)
21
22  sol((rank*lSize)+1:((rank+1)*lSize)) = x
23  do i=0,mpiSize-1   !update sol
24    CALL MPI_BCAST(sol((i*lSize)+1:((i+1)*lSize)), lSize, &
25           MPI_DOUBLE_PRECISION, i, MPI_COMM_WORLD, ierror)
26  end do
27
28  call MPI_FINALIZE(ierror)   !finalize mpi
29
30  call system_clock(count=ie) !finalize stop-time
31  write(*,*) ((ie-ia)/real(n_p_sec)) !output: used time
32 end program bicgtest
```

# BICG MPI

```
1  subroutine bicg(gSize,lSize,mpiSize,rank,tol,A,A_,b,x)
2
3      integer, intent(in)             :: gSize,lSize,rank,mpiSize
4      real(kind(1.d0)), intent(in)    :: tol
5      real(kind(1.d0)), intent(in)    :: A(lSize,gSize),A_(lSize,gSize)
6      real(kind(1.d0)), intent(in)    :: b(lSize)
7      real(kind(1.d0)), intent(inout) :: x(lSize)
8      !
9      real(kind(1.d0)) :: r(lSize), r_(lSize)
10     real(kind(1.d0)) :: p(gSize), p_(gSize)
11     real(kind(1.d0)) :: tmpV(lSize), tmpV_(lSize)
12     real(kind(1.d0)) :: alpha, beta, norm, tmp1, tmp2
13     real(kind(1.d0)) :: dotProd, dotProdOld
14     integer          :: i,j, ierror, si, ei
15
16     si       = (rank*lSize)+1
17     ei       = (rank+1)*lSize
18     r        = b
19     r_       = r
20     p(si:ei) = r
21     p_(si:ei) = r
22
23     do i=0,mpiSize-1     !update p, p_
24         CALL MPI_BCAST(p((i*lSize)+1:((i+1)*lSize)), lSize, &
25                        MPI_DOUBLE_PRECISION, i, MPI_COMM_WORLD, ierror)
26         CALL MPI_BCAST(p_((i*lSize)+1:((i+1)*lSize)), lSize, &
27                        MPI_DOUBLE_PRECISION, i, MPI_COMM_WORLD, ierror)
28     end do
29     dotProdOld = dot_product(r,r_)
30 !...
```

# BICG MPI

```
!...
    do j=1,gSize
        if(dot_product(r,r) < tol) then
            return
        end if
        tmpV  = matmul(A,p)
        tmpV_ = matmul(A_,p_)
        tmp2  = dot_product(tmpV, p_(si:ei))
        !reduce tmp2
        call MPI_ALLREDUCE( tmp2, tmp2, 1, MPI_DOUBLE_PRECISION, &
                            MPI_SUM, MPI_COMM_WORLD, ierror)
        alpha = dotProdOld/tmp2
        x  = x  + alpha*p(si:ei)
        r  = r  - alpha*tmpV
        r_ = r_ - alpha*tmpV_
        dotProd    = dot_product(r,r_)
        !reduce dotProd
        call MPI_ALLREDUCE( dotProd, dotProd, 1, MPI_DOUBLE_PRECISION, &
                            MPI_SUM, MPI_COMM_WORLD, ierror)
        beta       = dotProd/dotProdOld
        dotProdOld = dotProd
        p(si:ei)  = r  + beta*p(si:ei)
        p_(si:ei) = r_ + beta*p_(si:ei)
        !update p, p_
        do i=0,mpiSize-1
            CALL MPI_BCAST(p((i*lSize)+1:((i+1)*lSize)), lSize, &
                           MPI_DOUBLE_PRECISION, i, MPI_COMM_WORLD, ierror)
            CALL MPI_BCAST(p_((i*lSize)+1:((i+1)*lSize)), lSize, &
                           MPI_DOUBLE_PRECISION, i, MPI_COMM_WORLD, ierror)
        end do
    end do
end subroutine bicg
```