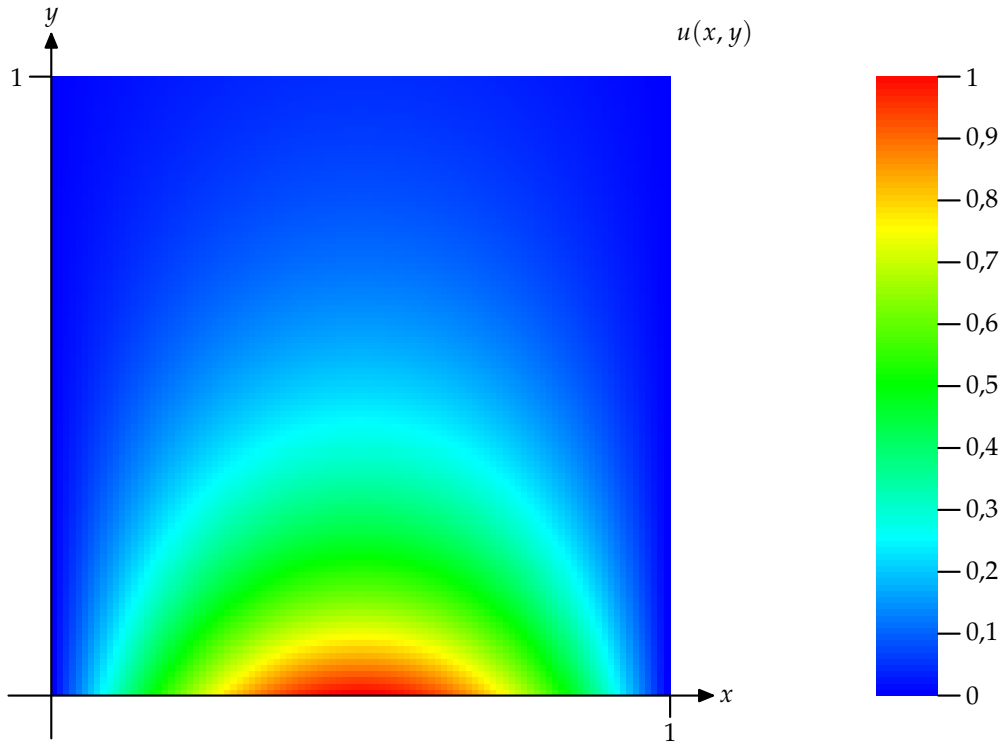
Wie können im Einzelfalle zu lösende Probleme in geeigneter Weise auf n Prozesse aufgeteilt werden?
Problempunkte:

- Mit wievielen Partnern muss ein einzelner Prozess kommunizieren? Lässt sich das unabhängig von der Problemgröße und der Zahl der beteiligten Prozesse begrenzen?
- Wieviele Daten sind mit den jeweiligen Partnern auszutauschen?
- Wie wird die Kommunikation so organisiert, dass Deadlocks sicher vermieden werden?
- Wieweit lässt sich die Kommunikation parallelisieren?

Fallstudie: Poisson-Gleichung mit Dirichlet-Randbedingung



- Gesucht sei eine numerische Näherung einer Funktion $u(x, y)$ für $(x, y) \in \Omega = [0, 1] \times [0, 1]$, für die gilt: $u_{xx} + u_{yy} = 0$ mit der Randbedingung $u(x, y) = g(x, y)$ für $x, y \in \delta\Omega$.
- Das obige Beispiel zeigt eine numerische Lösung für die Randbedingungen $u(x, 0) = sin(\pi x)$, $u(x, 1) = sin(\pi x)e^{-\pi}$ und $u(0, y) = u(1, y) = 0$.

$$
\begin{array}{ccccc}
A_{0,N-1} & - & A_{1,N-1} & \dots & A_{N-2,N-1} - A_{N-1,N-1} \\
| & & | & & | \quad\quad | \\
A_{0,N-2} & - & A_{1,N-2} & \dots & A_{N-2,N-2} - A_{N-1,N-2} \\
\vdots & & \vdots & & \vdots \quad\quad \vdots \\
A_{0,1} & -- & A_{1,1} & \dots & A_{N-2,1} -- A_{N-1,1} \\
| & & | & & | \quad\quad | \\
A_{0,0} & -- & A_{1,0} & \dots & A_{N-2,0} -- A_{N-1,0}
\end{array}
$$

- Numerisch lässt sich das Problem lösen, wenn das Gebiet Ω in ein $N \times N$ Gitter gleichmäßig zerlegt wird.
- Dann lässt sich $u(x, y)$ auf den Gitterpunkten durch eine Matrix A darstellen, wobei

$$A_{i,j} = u(\frac{i}{N}, \frac{j}{N})$$

für $i, j = 0 \dots N$.

- Hierbei lässt sich A schrittweise approximieren durch die Berechnung von A_0, $A_1 \ldots$, wobei A_0 am Rand die Werte von $g(x, y)$ übernimmt und ansonsten mit Nullen gefüllt wird.
- Es gibt mehrere iterative numerische Verfahren, wovon das einfachste das Jacobi-Verfahren ist mit dem sogenannten 5-Punkt-Differenzenstern:

$$A_{k+1_{i,j}} = \frac{1}{4}\left(A_{k_{i-1,j}} + A_{k_{i,j-1}} + A_{k_{i,j+1}} + A_{k_{i+1,j}}\right)$$

  für $i, j \in 1 \ldots N-1, k = 1, 2, \ldots$
  (Zur Herleitung siehe Alefeld et al, *Parallele numerische Verfahren*, S. 18 ff.)
- Die Iteration wird solange wiederholt, bis

$$\max_{i,j=1\ldots N-1} \left|A_{k+1_{i,j}} - A_{k_{i,j}}\right| \leq \epsilon$$

  für eine vorgegebene Fehlergrenze ϵ gilt.

mpi-jacobi.cpp

```
template<typename T, template<typename> typename Matrix,
   Require<Ge<Matrix<T>>> = true>
T jacobi_iteration(const Matrix<T>& A, Matrix<T>& B) {
   assert(A.numRows() > 2 && A.numCols() > 2);
   T maxdiff = 0;
   for (std::size_t i = 1; i + 1 < B.numRows(); ++i) {
      for (std::size_t j = 1; j + 1 < B.numCols(); ++j) {
         B(i, j) = 0.25 *
            (A(i - 1, j) + A(i + 1, j) + A(i, j - 1) + A(i, j + 1));
         T diff = std::fabs(A(i, j) - B(i, j));
         if (diff > maxdiff) maxdiff = diff;
      }
   }
   return maxdiff;
}
```

- *A.numRows* bzw. *B.numRows* entsprechen hier N. A repräsentiert A_k und B die Näherungslösung des folgenden Iterationsschritts A_{k+1}.

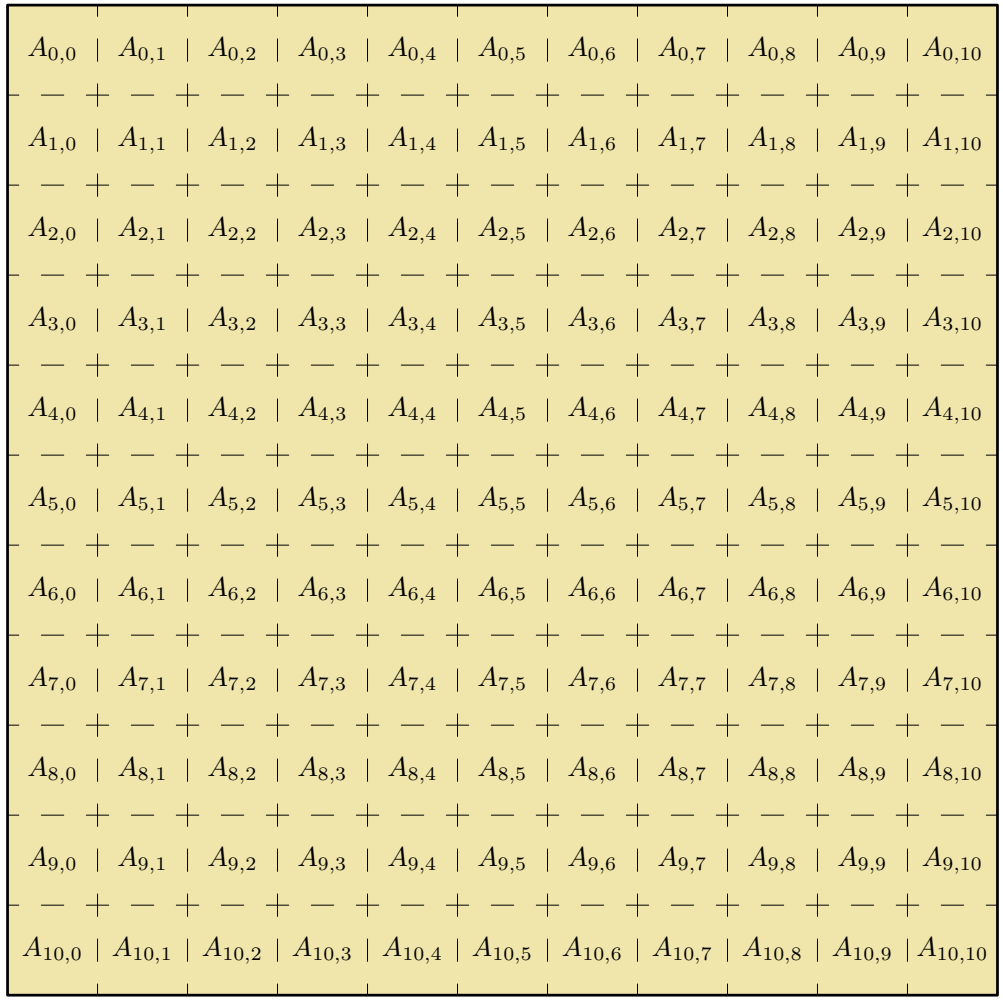
Initialisierung von A_0

mpi-jacobi.cpp

```
Matrix A(100, 100, Order::RowMajor);
if (rank == 0) {
   for (auto [i, j, Aij]: A) {
      if (j == 0) {
         Aij = std::sin(PI<T> * (T(i)/(A.numRows()-1)));
      } else if (j == A.numCols() - 1) {
         Aij = std::sin(PI<T> * (T(i)/(A.numRows()-1))) *
            E_POWER_MINUS_PI<T>;
      } else {
         Aij = 0;
      }
   }
}
```

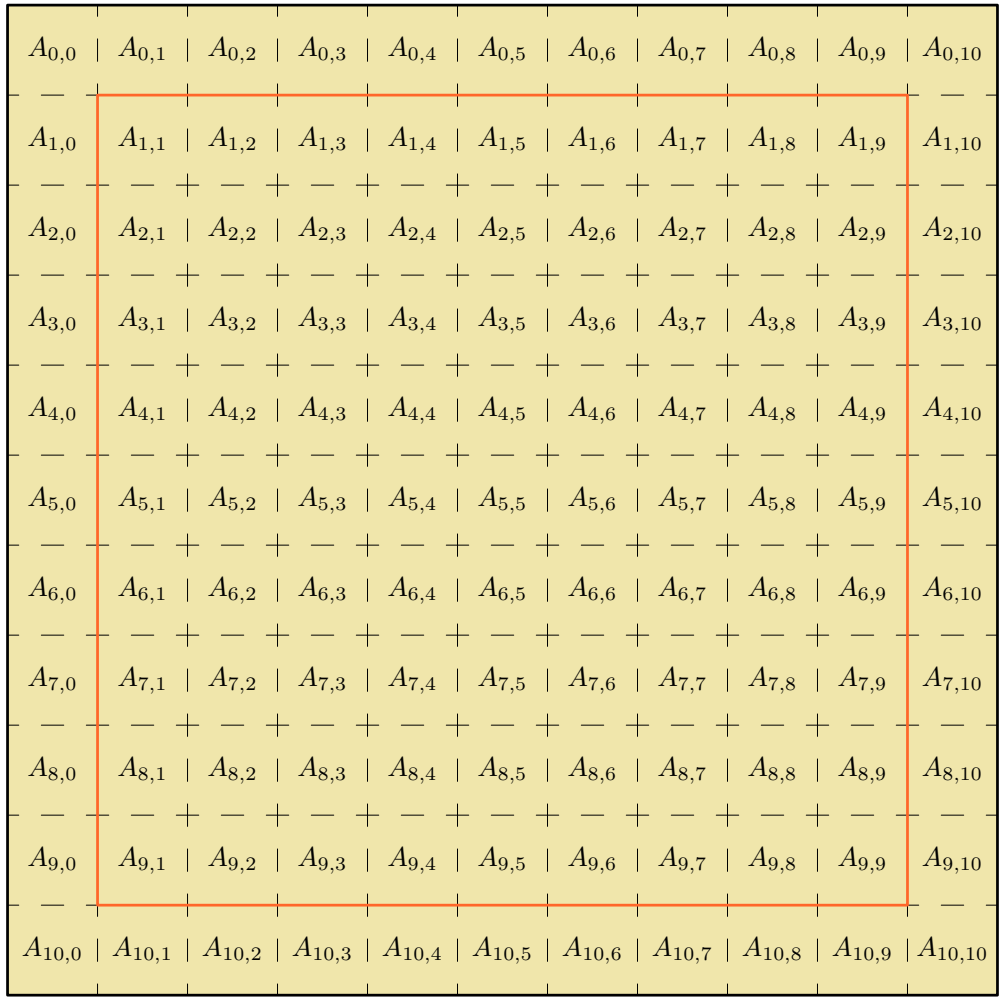
- Der gesamte Innenbereich wird mit 0 initialisiert.
- Für den Rand gelten die Randbedingungen $u(x,0) = sin(\pi x)$, $u(x,1) = sin(\pi x)e^{-\pi}$ und $u(0,y) = u(1,y) = 0$.

Eindimensionale Partitionierung



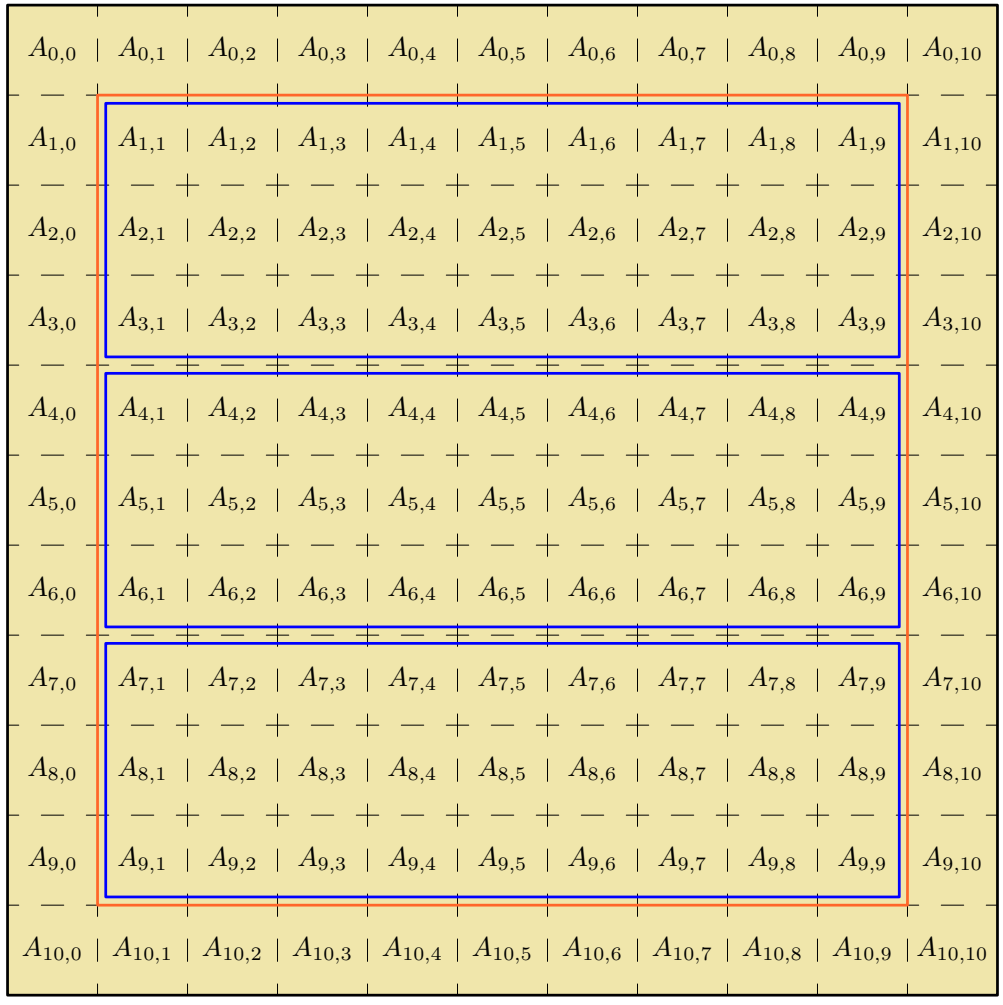
- Gegeben sei eine initialisierte Matrix A.

Eindimensionale Partitionierung



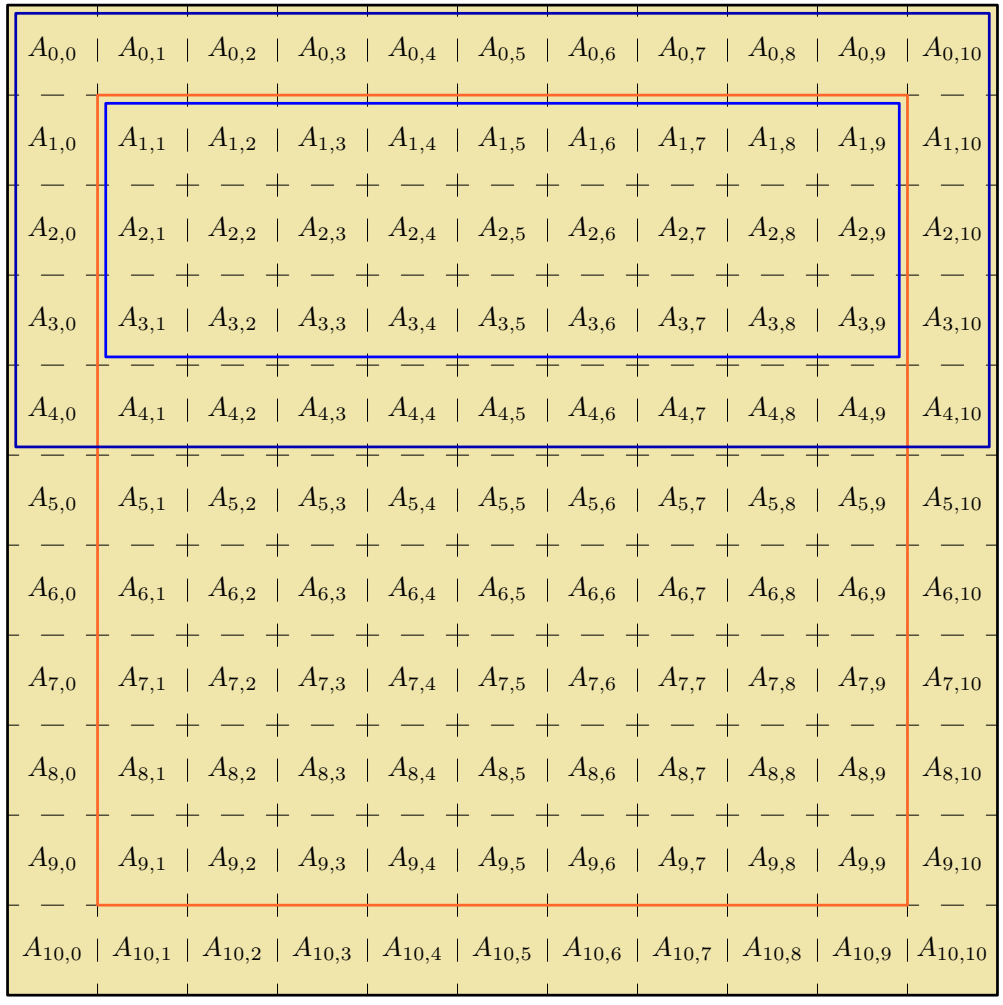
- Der Rand von A ist fest vorgegeben, der innere Teil ist näherungsweise zu bestimmen.

Eindimensionale Partitionierung



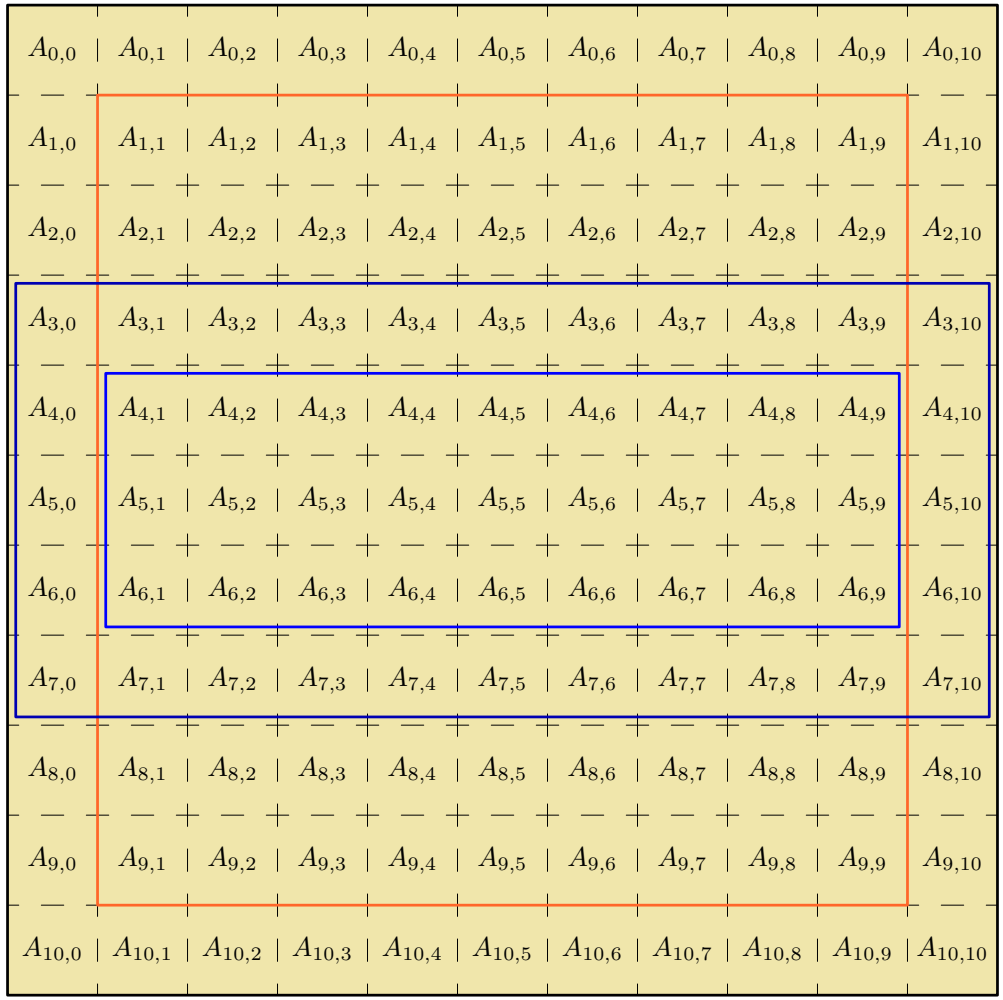
- Der innere Teil von A ist auf m Prozesse (hier $m = 3$) gleichmäßig aufzuteilen.

Eindimensionale Partitionierung



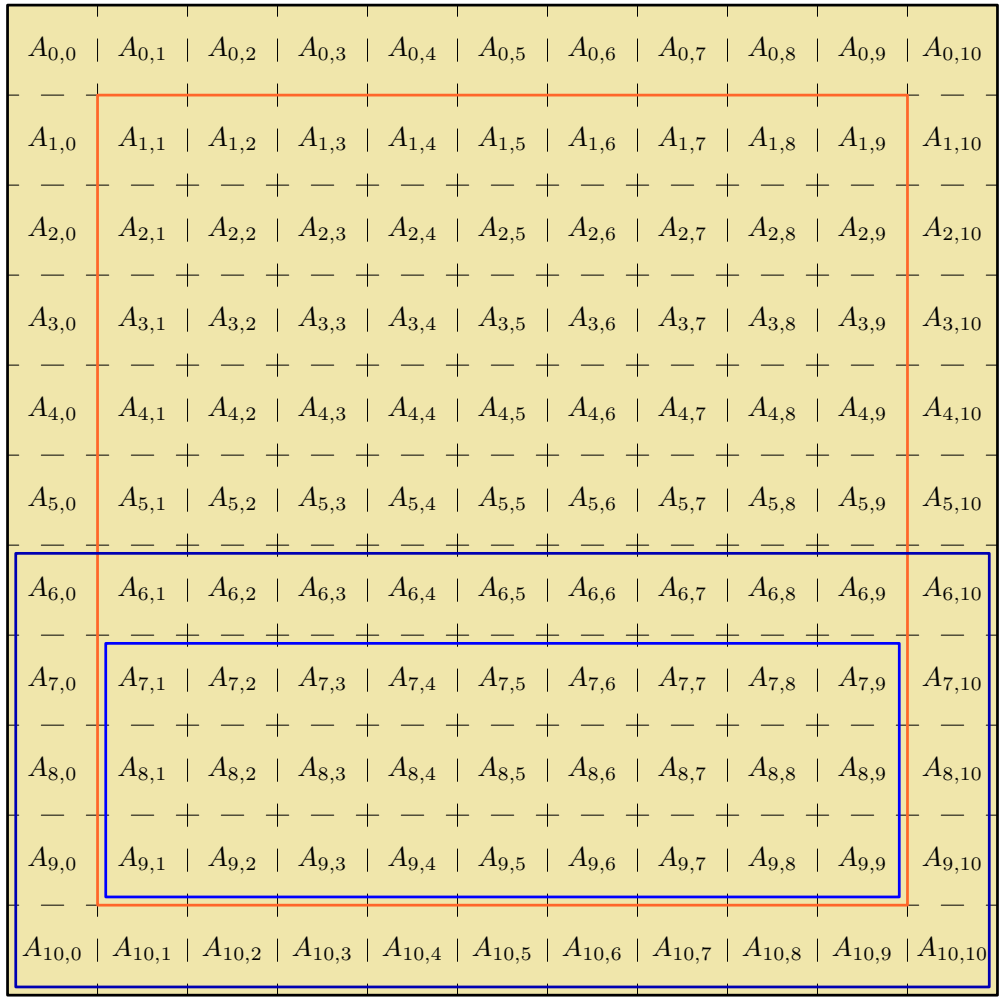
- Jeder der Prozesse benötigt lesenderweise den Rand. Soweit es sich nicht um den äußeren Rand handelt, muss dieser vom jeweiligen Nachbarn nach jedem Iterationsschritt erneut organisiert werden.

Eindimensionale Partitionierung



- Jeder der Prozesse benötigt lesenderweise den Rand. Soweit es sich nicht um den äußeren Rand handelt, muss dieser vom jeweiligen Nachbarn nach jedem Iterationsschritt erneut organisiert werden.

Eindimensionale Partitionierung



- Jeder der Prozesse benötigt lesenderweise den Rand. Soweit es sich nicht um den äußeren Rand handelt, muss dieser vom jeweiligen Nachbarn nach jedem Iterationsschritt erneut organisiert werden.

Kommunikationsaufwand bei einer ein-dimensionalen Partitionierung

- Im vorgestellten Beispiel mit $N = 11$ und $m = 3$ sind folgende Übertragungen nach einem Iterationsschritt notwendig:
  - $P_1 \longrightarrow P_2 : A_{3,1} \ldots A_{3,9}$
  - $P_2 \longrightarrow P_3 : A_{6,1} \ldots A_{6,9}$
  - $P_3 \longrightarrow P_2 : A_{7,1} \ldots A_{7,9}$
  - $P_2 \longrightarrow P_1 : A_{4,1} \ldots A_{4,9}$
- Jede innere Partition erhält und sendet zwei innere Zeilen von A. Die Randpartitionen empfangen und senden jeweils nur eine Zeile.
- Generell müssen $2m - 2$ Datenblöcke mit jeweils $N - 2$ Werten verschickt werden. Dies lässt sich prinzipiell parallelisieren.

- Ein Ansatz wäre eine Paarbildung, d.h. zuerst kommunizieren die Prozesspaare $(0,1)$, $(2,3)$ usw. untereinander. Danach werden die Paare $(1,2)$, $(3,4)$ usw. gebildet. Bei jedem Paar würde zuerst der Prozess mit der niedrigeren Nummer senden und der mit der höheren Nummer empfangen und danach würden die Rollen jeweils vertauscht werden.
- Alternativ bietet sich auch die Verwendung der MPI-Operation *MPI_Sendrecv* an, die parallel eine *MPI_Send*- und eine *MPI_Recv*-Operation gleichzeitig verfolgt.
- Dann könnte der Austausch in zwei Wellen erfolgen, zuerst aufwärts von 0 nach 1, 1 nach 2 usw. und danach abwärts von m nach $m-1$, $m-1$ nach $m-2$ usw.

Vermeidung des Deadlocks mit *MPI_Sendrecv* 331

mpi-sendrecv.cpp

```
int main(int argc, char** argv) {
   MPI_Init(&argc, &argv);
   int nofprocesses; MPI_Comm_size(MPI_COMM_WORLD, &nofprocesses);
   int rank; MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   assert(nofprocesses == 2); const int other = 1 - rank;
   const unsigned int maxsize = 8192;
   double* bigbuf[2] = {new double[maxsize], new double[maxsize]};
   for (int len = 1; len <= maxsize; len *= 2) {
      MPI_Status status;
      MPI_Sendrecv(
         bigbuf[rank], len, MPI::DOUBLE, other, 0,
         bigbuf[other], len, MPI::DOUBLE, other, 0,
         MPI_COMM_WORLD, &status);
      if (rank == 0) cout << "len = " << len << " survived" << endl;
   }
   MPI_Finalize();
}
```

- Bei *MPI_Sendrecv* werden zuerst die Parameter für *MPI_Send* angegeben, dann die für *MPI_Recv*.

mpi-jacobi.cpp

```
template<typename Matrix>
void exchange_with_neighbors(Matrix& A,
      int previous, int next, MPI_Datatype rowtype) {
   MPI_Status status;
   // upward
   MPI_Sendrecv(&A(1, 0), 1, rowtype, previous, 0,
      &A(A.numRows()-1, 0), 1, rowtype, next, 0,
      MPI_COMM_WORLD, &status);
   // downward
   MPI_Sendrecv(&A(A.numRows()-2, 0), 1, rowtype, next, 0,
      &A(0, 0), 1, rowtype, previous, 0,
      MPI_COMM_WORLD, &status);
}
```

- Der Austausch wird in zwei Wellen organisiert: Zuerst werden Zeilen von höheren Ranks zu niedrigeren übermittelt, dann umgekehrt.

mpi-jacobi.cpp

```
T eps = 1e-6; unsigned int iterations;
for (iterations = 0; ; ++iterations) {
   T maxdiff = jacobi_iteration(B1, B2);
   exchange_with_neighbors(B2, previous, next, rowtype);
   maxdiff = jacobi_iteration(B2, B1);
   if (iterations % 10 == 0) {
      T global_max;
      MPI_Reduce(&maxdiff, &global_max, 1, get_type(maxdiff),
         MPI_MAX, 0, MPI_COMM_WORLD);
      MPI_Bcast(&global_max, 1, get_type(maxdiff), 0, MPI_COMM_WORLD);
      if (global_max < eps) break;
   }
   exchange_with_neighbors(B1, previous, next, rowtype);
}
if (rank == 0) fmt::printf("%d iterations\n", iterations);
```

- Die zentrale Schleife läuft, bis das Abbruchkriterium $\max_{i,j=1\ldots N-1} \left|A_{k+1_{i,j}} - A_{k_{i,j}}\right| \leq \epsilon$ erfüllt ist.
- Die Matrizen B_1 und B_2 umfassen jeweils nur die lokal zu betrachtenden Zeilen und den Rand. Es werden immer zwei Schritte durchgeführt: $B_1 \to B_2$ gefolgt von $B_2 \to B_1$.

Initialisierung von B_1 und B_2

mpi-jacobi.cpp

```
UniformSlices<std::size_t> slices(nof_processes, A.numRows() - 2);
Matrix B1(slices.size(rank) + 2, A.numCols(), Order::RowMajor);
for (auto [i, j, Bij]: B1) {
   (void) i; (void) j; // suppress g++ warning
   Bij = 0;
}
auto B = B1.block(1, 0).dim(B1.numRows() - 2, B1.numCols());
MPI_Datatype rowtype = get_row_type(B);
```

- Mit *UniformSlices* wird festgelegt, wie die Zeilen der Gesamtmatrix A auf die einzelnen Prozesse aufgeteilt wird.
- B ist eine Teilmatrix ohne den linken und rechten Rand, der sich (nach der anfänglichen Initialisierung) nicht mehr verändert.
- Nur der innere Teil der Zeilen ist jeweils zu übermitteln.

mpi-jacobi.cpp

```
/* distribute main body of A */
auto A_ = A.block(1, 0).dim(A.numRows() - 2, A.numCols());
scatter_by_row(A_, B, 0, MPI_COMM_WORLD);
```

- $A\_$ ist der Innenteil der Matrix A ohne den Rand.
- Dann wird mit *scatter_by_row* der Innenteil der Gesamtmatrix A auf die Innenteile der individuellen Teilmatrizen B verteilt.

Übermittlung des Rands

mpi-jacobi.cpp

```
/* distribute first and last row of A */
if (rank == 0) {
   copy(A.block(0, 0).dim(1, A.numCols()),
      B1.block(0, 0).dim(1, B1.numCols()));
}
if (nof_processes == 1) {
   copy(A.block(A.numRows()-1, 0).dim(1, A.numCols()),
      B1.block(B.numRows()-1, 0).dim(1, B1.numCols()));
} else if (rank == 0) {
   MPI_Send(&A(A.numRows()-1, 0), 1, rowtype, nof_processes-1, 0,
      MPI_COMM_WORLD);
} else if (rank == nof_processes - 1) {
   MPI_Status status;
   MPI_Recv(&B1(B.numRows()-1, 0), 1, rowtype,
      0, 0, MPI_COMM_WORLD, &status);
}

Matrix B2(B1.numRows(), B1.numCols(), Order::RowMajor);
copy(B1, B2); /* actually just the border needs to be copied */
```

mpi-jacobi.cpp

```
int previous = rank == 0? MPI_PROC_NULL: rank-1;
int next = rank == nof_processes-1? MPI_PROC_NULL: rank+1;
```

- Damit die Sonderfälle am Rand nicht unnötig kompliziert werden, gibt es in MPI den sogenannten Nullprozess: *MPI_PROC_NULL*.
- Wenn bei einer der Kommunikationsoperationen der Nullprozess angegeben wird, dann findet eben keine Kommunikation statt.

mpi-jacobi.cpp

```
gather_by_row(B, A_, 0, MPI_COMM_WORLD);

MPI_Finalize();

if (rank == 0) {
   auto pixbuf = create_pixbuf(A, [](T val) -> HSVColor<float> {
      return HSVColor<float>((1-val) * 240, 1, 1);
   }, 8);
   gdk_pixbuf_save(pixbuf, "jacobi.jpg", "jpeg", nullptr,
      "quality", "100", nullptr);
}
```

- Mit *gather_by_row* wird beim Prozess mit dem Rank 0 die gesamte Matrix zusammengestellt.
- Sobald das fertig ist, kann das MPI-Kommunikationsnetzwerk herunterfahren werden.
- Mit *create_pixbuf* wird die Matrix in einen Pixelbuffer der GDK-Pixbuf-Bibliothek konvertiert und dieser danach in eine JPEG-Datei abgesichert.

- Mit *MPI_Isend* und *MPI_Irecv* bietet die MPI-Schnittstelle eine asynchrone Kommunikationsschnittstelle.
- Die Aufrufe blockieren nicht und entsprechend wird die notwendige Kommunikation parallel zum übrigen Geschehen abgewickelt.
- Wenn es geschickt aufgesetzt wird, können einige Berechnungen parallel zur Kommunikation ablaufen.
- Das ist sinnvoll, weil sonst die CPU-Ressourcen während der Latenzzeiten ungenutzt bleiben.
- Die Benutzung folgt dem Fork-And-Join-Pattern, d.h. mit *MPI_Isend* und *MPI_Irecv* läuft die Kommunikation parallel zum Programmtext nach den Aufrufen ab und mit *MPI_Wait* ist eine Synchronisierung wieder möglich.
- Zu beachten ist, dass die angegebenen Puffer während der laufenden Kommunikation nicht benutzt werden dürfen.

mpi-jacobi-nb.cpp

```
template<typename T, template<typename> class Matrix,
   Require<Ge<Matrix<T>>> = true>
void exchange_with_neighbors(Matrix<T>& A,
      /* ranks of the neighbors */
      int previous, int next,
      /* data type for an inner row, i.e. without the border */
      MPI_Datatype rowtype) {
   MPI_Request requests[4]; int request_index = 0;
   MPI_Irecv(&A(0, 1), 1, rowtype, previous, 0,
      MPI_COMM_WORLD, &requests[request_index++]);
   MPI_Irecv(&A(A.numRows()-1, 1), 1, rowtype, next, 0,
      MPI_COMM_WORLD, &requests[request_index++]);
   MPI_Isend(&A(1, 1), 1, rowtype, previous, 0,
      MPI_COMM_WORLD, &requests[request_index++]);
   MPI_Isend(&A(A.numRows()-2, 1), 1, rowtype, next, 0,
      MPI_COMM_WORLD, &requests[request_index++]);

   for (auto& request: requests) {
      MPI_Status status;
      MPI_Wait(&request, &status);
   }
}
```

mpi-jacobi-nb.cpp

```
MPI_Request requests[4]; int request_index = 0;
MPI_Irecv(&A(0, 1), 1, rowtype, previous, 0,
   MPI_COMM_WORLD, &requests[request_index++]);
MPI_Irecv(&A(A.numRows()-1, 1), 1, rowtype, next, 0,
   MPI_COMM_WORLD, &requests[request_index++]);
MPI_Isend(&A(1, 1), 1, rowtype, previous, 0,
   MPI_COMM_WORLD, &requests[request_index++]);
MPI_Isend(&A(A.numRows()-2, 1), 1, rowtype, next, 0,
   MPI_COMM_WORLD, &requests[request_index++]);
```

- Die Methoden *MPI_Isend* und *MPI_Irecv* werden analog zu *MPI_Send* und *MPI_Recv* aufgerufen, liefern aber ein *MPI_Request*-Objekt zurück.
- Die nicht-blockierenden Operationen initiieren jeweils nur die entsprechende Kommunikation. Der übergebene Datenbereich darf andersweitig nicht verwendet werden, bis die jeweilige Operation abgeschlossen ist.
- Dies kann durch die dadurch gewonnene Parallelisierung etwas bringen. Allerdings wird das durch zusätzlichen Overhead (mehr lokale Threads) bezahlt.

Abschluss einer nichtblockierenden Kommunikation 342

mpi-jacobi-nb.cpp

```
for (auto& request: requests) {
   MPI_Status status;
   MPI_Wait(&request, &status);
}
```

- Für Objekte des Typs *MPI_Request* stehen die Funktionen *MPI_Wait* und *MPI_Test* zur Verfügung.
- Mit *MPI_Wait* kann auf den Abschluss gewartet werden; mit *MPI_Test* ist die nicht-blockierende Überprüfung möglich, ob die Operation bereits abgeschlossen ist.

- Die Partitionierung eines Problems auf einzelne Prozesse und deren Kommunikationsbeziehungen kann als Graph repräsentiert werden, wobei die Prozesse die Knoten und die Kommunikationsbeziehungen die Kanten repräsentieren.
- Der Graph ist normalerweise ungerichtet, weil zumindest die zugrundeliegenden Kommunikationsarchitekturen und das Protokoll bidirektional sind.

Abbildung des Graphen auf eine physische Netzwerkstruktur

- Da die Bandbreiten und Latenzzeiten zwischen einzelnen rechnenden Knoten nicht überall gleich sind, ist es sinnvoll, die Aufteilung der Prozesse auf Knoten so zu organisieren, dass die Kanten möglichst weitgehend auf günstigere Kommunikationsverbindungen gelegt werden.
- Bei Infiniband spielt die Organisation kaum eine Rolle, es sei denn, es liegt eine Zwei-Ebenen-Architektur vor wie beispielsweise bei *SuperMUC* in München.
- Bei MP-Systemen mit gemeinsamen Speicher ist es günstiger, wenn miteinander kommunizierende Prozesse auf Kernen des gleichen Prozessors laufen, da diese typischerweise einen Cache gemeinsam nutzen können und somit der Umweg über den langsamen Hauptspeicher vermieden wird.
- Bei Titan und anderen Installationen, die in einem dreidimensionalen Torus organisiert sind, spielt Nachbarschaft eine wichtige Rolle.

- MPI bietet die Möglichkeit, beliebige Kommunikationsgraphen zu deklarieren.
- Zusätzlich unterstützt bzw. vereinfacht MPI die Deklarationen n-dimensionaler Gitterstrukturen, die in jeder Dimension mit oder ohne Ringstrukturen konfiguriert werden können. Entsprechend sind im eindimensionalen einfache Ketten oder Ringe möglich und im zweidimensionalen Fall Matrizen, Zylinder oder Tori.
- Dies eröffnet MPI die Möglichkeit, eine geeignete Zuordnung von Prozessen auf Prozessoren vorzunehmen.
- Ferner lassen sich über entsprechende MPI-Funktionen die Kommunikationsnachbarn eines Prozesses ermitteln.
- Grundsätzlich ist eine Kommunikation abseits des vorgegebenen Kommunikationsgraphen möglich. Nur bietet diese möglicherweise höhere Latenzzeiten und/oder niedrigere Bandbreiten.

mpi-jacobi-2d.cpp

```
/* create two-dimensional Cartesian grid for our prcesses */
int dims[2] = {0, 0}; int periods[2] = {false, false};
MPI_Dims_create(nof_processes, 2, dims);
MPI_Comm grid;
MPI_Cart_create(MPI_COMM_WORLD,
   2,         // number of dimensions
   dims,      // actual dimensions
   periods,   // both dimensions are non-periodical
   true,      // reorder is permitted
   &grid      // newly created communication domain
);
MPI_Comm_rank(grid, &rank); // update rank (could have changed)
```

- Mit *MPI_Dims_create* lässt sich die Anzahl der Prozesse auf ein Gitter aufteilen.
- Die Funktion *MPI_Cart_create* erzeugt dann das Gitter und teilt ggf. die Prozesse erneut auf, d.h. *rank* könnte sich dadurch ändern.

mpi-jacobi-2d.cpp

```
int dims[2] = {0, 0}; int periods[2] = {false, false};
MPI_Dims_create(nof_processes, 2, dims);
```

- Die Prozesse sind so auf ein zweidimensionales Gitter aufzuteilen, dass *dims*[0]∗*dims*[1] == *nofprocesses* gilt.
- *MPI_Dims_create* erwartet die Zahl der Prozesse, die Zahl der Dimensionen und ein entsprechend dimensioniertes Dimensions-Array.
- Die Funktion ermittelt die Teiler von *nofprocesses* und sucht nach einer in allen Dimensionen möglichst gleichmäßigen Aufteilung. Wenn *nofprocesses* prim ist, entsteht dabei die ungünstige Aufteilung $1 \times$ *nofprocesses*.
- Das Dimensions-Array *dims* muss zuvor initialisiert werden. Bei Nullen darf *Compute_dims* einen Teiler von *nofprocesses* frei wählen; andere Werte müssen Teiler sein und sind dann zwingende Vorgaben.

mpi-jacobi-2d.cpp

```
int dims[2] = {0, 0}; int periods[2] = {false, false};
MPI_Dims_create(nof_processes, 2, dims);
MPI_Comm grid;
MPI_Cart_create(MPI_COMM_WORLD,
   2,         // number of dimensions
   dims,      // actual dimensions
   periods,   // both dimensions are non-periodical
   true,      // reorder is permitted
   &grid      // newly created communication domain
);
```

- *MPI_Cart_create* erwartet im zweiten und dritten Parameter die Zahl der Dimensionen und das entsprechende Dimensionsfeld.
- Der vierte Parameter legt über ein **int**-Array fest, welche Dimensionen ring- bzw. torusförmig angelegt sind. Hier liegt eine einfache Matrixstruktur vor und entsprechend sind beide Werte **false**.
- Der vierte und letzte Parameter erklärt, ob eine Neuzuordnung zulässig ist, um die einzelnen Prozesse und deren Kommunikationsstruktur möglichst gut auf die vorhandene Netzwerkstruktur abzubilden. Hier sollte normalerweise **true** gegeben werden. Allerdings ändert sich dann möglicherweise der *rank*, so dass dieser erneut abzufragen ist.

mpi-jacobi-2d.cpp

```
/* get our position within the grid */
int coords[2];
MPI_Cart_coords(grid, rank, 2, coords);
```

- Mit *MPI_Cart_coords* lässt sich der Rank einer Kommunikationsdomäne in Koordinaten konvertieren.
- Das nutzen wir hier, um die eigenen Koordinaten zu ermitteln.

Anlage der Blockmatrizen B_1 und B_2

mpi-jacobi-2d.cpp

```
/* create matrices B1, B2 for our subarea */
int overlap = 1;
UniformSlices<int> rows(dims[0], A.numRows() - 2*overlap);
UniformSlices<int> columns(dims[1], A.numCols() - 2*overlap);

Matrix B1(rows.size(coords[0]) + 2*overlap,
          columns.size(coords[1]) + 2*overlap,
          Order::RowMajor);
Matrix B2(rows.size(coords[0]) + 2*overlap,
          columns.size(coords[1]) + 2*overlap,
          Order::RowMajor);
```

- Mit Hilfe von *UniformSlices* wird der Innenteil von A zweidimensional möglichst gleichmäßig aufgeteilt.
- Entsprechend werden B_1 und B_2 angelegt.

mpi-jacobi-2d.cpp

```
/* distribute main body of A including left and right border */
scatter_by_block(A, B1, 0, grid, overlap);

copy(B1, B2); /* actually just the border needs to be copied */
```

- Mit *scatter_by_block* wird die gesamte Matrix blockweise aufgeteilt, wobei die sich überlappenden Teile (mit *overlap* = 1) mit berücksichtigt werden.
- Somit werden auch die Ränder mitkopiert, wobei wir darauf achten müssen, dass auch B_2 eine Kopie des Rands erhält.

mpi-jacobi-2d.cpp

```
/* get the process numbers of our neighbors */
int left, right, upper, lower;
MPI_Cart_shift(grid, 0, 1, &upper, &lower);
MPI_Cart_shift(grid, 1, 1, &left, &right);
```

- Die Funktion *MPI_Cart_shift* liefert die Nachbarn in einer der Dimensionen.
- Der zweite Parameter ist die Dimension, der dritte Parameter der Abstand (hier 1 für den unmittelbaren Nachbarn).
- Die Prozessnummern der so definierten Nachbarn werden in den beiden folgenden Variablen abgelegt.
- Wenn in einer Richtung kein Nachbar existiert (z.B. am Rande einer Matrix), wird *MPI_PROC_NULL* zurückgeliefert.

mpi-jacobi-2d.cpp

```
/* used to specify an I/O transfer with one
   of our neighbors;
   each process has four input and four output
   operations per iteration
*/
struct IOTransfer {
   int rank; /* rank of neighbor */
   void* addr; /* start address */
   MPI_Datatype datatype; /* either a column or row */
};
```

- Jeder Prozess kommuniziert in jedem Schritt mit vier anderen Prozessen bidirektional.
- Es ist sinnvoll, das mit Hilfe einer Datenstruktur zu organisieren.

mpi-jacobi-2d.cpp

```
/* compute type for inner rows and cols without the border */
auto B_inner_row = B1.block(0, 1).
   dim(overlap, B1.numCols() - 2*overlap);
MPI_Datatype rowtype = get_type(B_inner_row);
auto B_inner_col = B1.block(0, 1).
   dim(B1.numRows() - 2*overlap, overlap);
MPI_Datatype coltype = get_type(B_inner_col);

IOTransfer B1_in_vectors[] = {
   {left, &B1(1, 0), coltype}, {upper, &B1(0, 1), rowtype},
   {right, &B1(1, B1.numCols()-overlap), coltype},
   {lower, &B1(B1.numRows()-overlap, 1), rowtype},
};
IOTransfer B1_out_vectors[] = {
   {left, &B1(1, 1), coltype}, {upper, &B1(1, 1), rowtype},
   {right, &B1(1, B1.numCols()-2*overlap), coltype},
   {lower, &B1(B1.numRows()-2*overlap, 1), rowtype},
};
```

- Die vier Ein- und vier Ausgabevektoren werden hier zusammen mit den passenden Datentypen in Arrays zusammengestellt. Analog auch für B_2.

mpi-jacobi-2d.cpp

```
template<typename T, template<typename> typename Matrix,
   Require<Ge<Matrix<T>>> = true>
T jacobi_iteration(const Matrix<T>& A, Matrix<T>& B,
       const IOTransfer (&in_vectors)[4],
       const IOTransfer (&out_vectors)[4], MPI_Comm grid) {
   T maxdiff = 0;
   // compute border zones
   // ...
   // exchange borders with our neighbors
   // ...
   // compute inner block in parallel to the ongoing communication
   // ...
   // wait until the communication is finished
   // ...
   return maxdiff;
}
```

- Der generelle Aufbau der Iterationsschleife ist im Vergleich zur eindimensionalen Partitionierung gleich geblieben, abgesehen davon, dass
  - alle vier Ränder zu Beginn zu berechnen sind und
  - insgesamt acht einzelne Ein- und Ausgabe-Operationen parallel abzuwickeln sind.

mpi-jacobi-2d.cpp

```
auto jacobi = [&](std::size_t i, std::size_t j) {
   B(i, j) = 0.25 *
      (A(i - 1, j) + A(i + 1, j) + A(i, j - 1) + A(i, j + 1));
   T diff = std::fabs(A(i, j) - B(i, j));
   if (diff > maxdiff) maxdiff = diff;
};

/* compute the border first which is sent in advance
   to our neighbors */
for (std::size_t i = 1; i + 1 < B.numRows(); ++i) {
   jacobi(i, 1);
   jacobi(i, B.numCols()-2);
}
for (std::size_t j = 2; j + 2 < B.numCols(); ++j) {
   jacobi(1, j);
   jacobi(B.numRows()-2, j);
}
```

mpi-jacobi-2d.cpp

```
/* send border to our neighbors and
   initiate the receive operations */
MPI_Request requests[8]; int ri = 0;
for (auto& in_vector: in_vectors) {
   MPI_Irecv(in_vector.addr, 1, in_vector.datatype,
      in_vector.rank, 0, grid, &requests[ri++]);
}
for (auto& out_vector: out_vectors) {
   MPI_Isend(out_vector.addr, 1, out_vector.datatype,
      out_vector.rank, 0, grid, &requests[ri++]);
}

/* compute the inner block in parallel
   to the initiated communication */
for (std::size_t i = 2; i + 2 < B.numRows(); ++i) {
   for (std::size_t j = 2; j + 2 < B.numCols(); ++j) {
      jacobi(i, j);
   }
}
```

mpi-jacobi-2d.cpp

```
/* main loop for the Jacobi iterations */
T eps = 1e-6; unsigned int iterations;
for (iterations = 0; ; ++iterations) {
   T maxdiff = jacobi_iteration(B1, B2, B2_in_vectors, B2_out_vectors,
      grid);
   maxdiff = jacobi_iteration(B2, B1, B1_in_vectors, B1_out_vectors,
      grid);
   if (iterations % 10 == 0) {
      T global_max;
      MPI_Reduce(&maxdiff, &global_max, 1, get_type(maxdiff),
         MPI_MAX, 0, grid);
      MPI_Bcast(&global_max, 1, get_type(maxdiff), 0, grid);
      if (global_max < eps) break;
   }
}
if (rank == 0) fmt::printf("%d iterations\n", iterations);
```

mpi-jacobi-2d.cpp

```
/* collect results */
gather_by_block(B1, A, 0, grid, overlap);
```

- Am Ende werden die Teilmatrizen mit *gather_by_block* zu einer Gesamtmatrix zusammengefügt, die dann wiederum in einen GDK-Pixbuf konvertiert und ausgegeben wird.