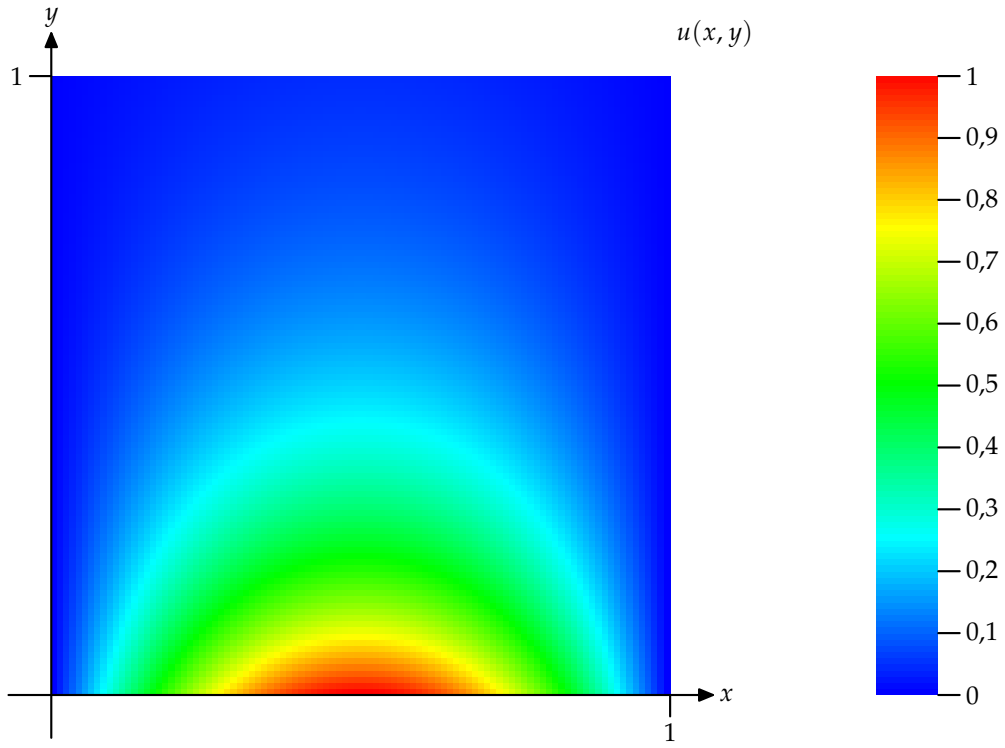
Wie können im Einzelfalle zu lösende Probleme in geeigneter Weise auf n Prozesse aufgeteilt werden?
Problempunkte:

- 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.

np-jacobi.cpp

```
// single Jacobi iteration step
double single_jacobi_iteration(double** A, double** B, int n, int m) {
   for (int i = 1; i <= n; ++i) {
      for (int j = 1; j <= m; ++j) {
         B[i][j] = 0.25 * (A[i-1][j] + A[i][j-1] +
                           A[i][j+1] + A[i+1][j]);
      }
   }
   double maxdiff = 0;
   for (int i = 1; i <= n; ++i) {
      for (int j = 1; j <= m; ++j) {
         double diff = fabs(A[i][j] - B[i][j]);
         if (diff > maxdiff) maxdiff = diff;
         A[i][j] = B[i][j];
      }
   }
   return maxdiff;
}
```

- n entspricht hier $N-2$. A repräsentiert A_k und B die Näherungslösung des folgenden Iterationsschritts A_{k+1}.

np-jacobi.cpp

```
void initialize_A(double& Aij, int i, int j, int N) {
   const static double E_POWER_MINUS_PI = pow(M_E, -M_PI);
   if (j == 0) {
      Aij = sin(M_PI * ((double)i/(N-1)));
   } else if (j == N-1) {
      Aij = sin(M_PI * ((double)i/(N-1))) * E_POWER_MINUS_PI;
   } else {
      Aij = 0;
   }
}
```

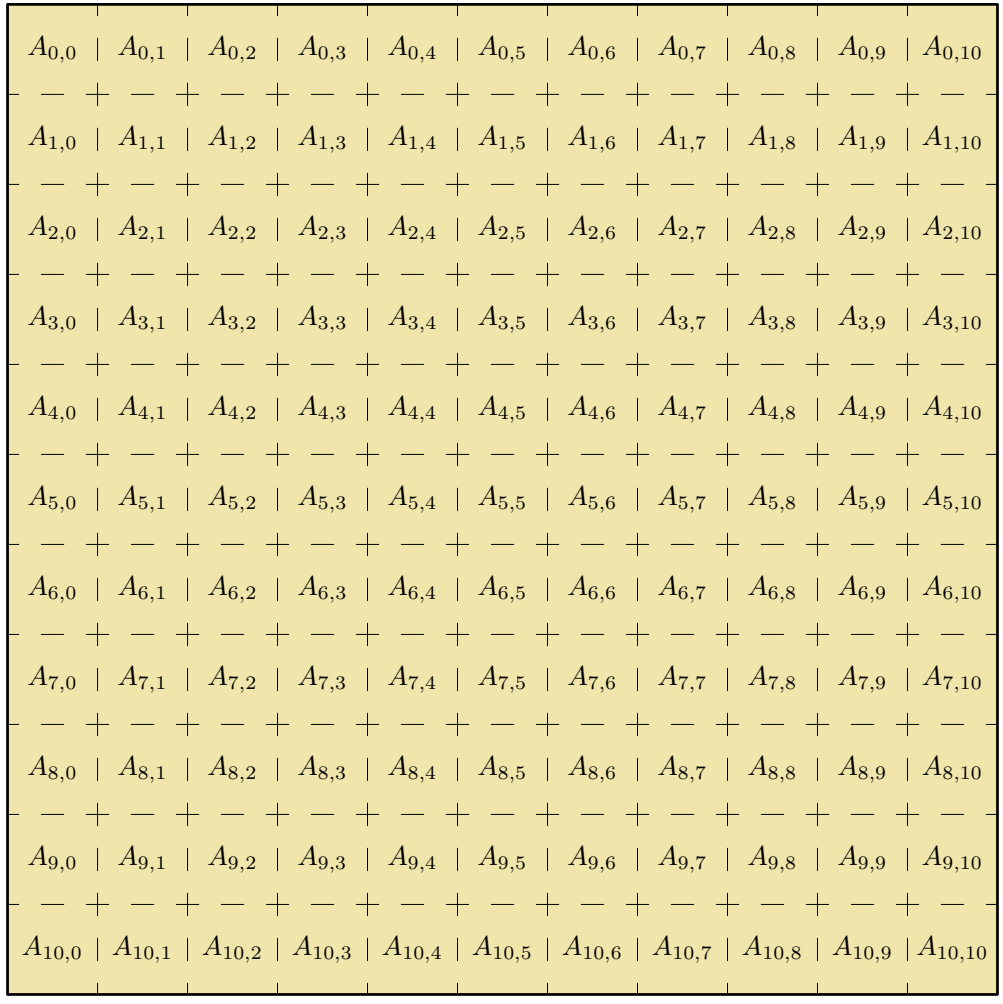
- Der gesamte Innenbereich wird mit 0 initialisiert.
- Für den Rand gelten die Randbedingungen $u(x,0) = sin(\pi x)$, $u(x,1) = sin(\pi x)e^{-\pi}$ und $u(0,y) = u(1,y) = 0$.

Nicht-parallelisierte Umsetzung

np-jacobi.cpp

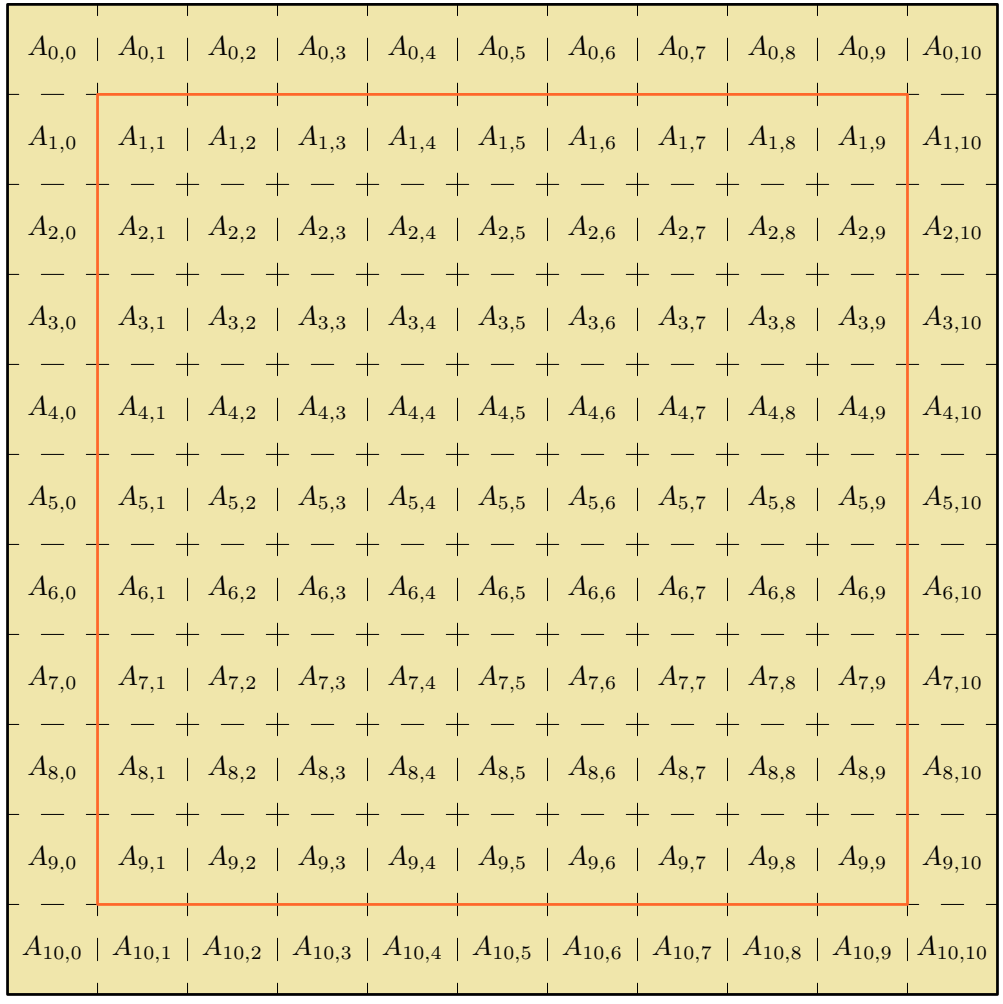
```
double** run_jacobi_iteration(int N, double eps) {
   int n = N-2;
   double** A = new double*[N]; assert(A);
   for (int i = 0; i < N; ++i) {
      A[i] = new double[N]; assert(A[i]);
      for (int j = 0; j < N; ++j) {
         initialize_A(A[i][j], i, j, N);
      }
   }
   double** B = new double*[N-1];
   for (int i = 1; i < N-1; ++i) {
      B[i] = new double[N-1]; assert(B[i]);
   }

   double maxdiff;
   do {
      maxdiff = single_jacobi_iteration(A, B, n, n);
   } while (maxdiff > eps);

   for (int i = 1; i < N-1; ++i) {
      delete[] B[i];
   }
   delete[] B;

   return A;
}
```

Eindimensionale Partitionierung



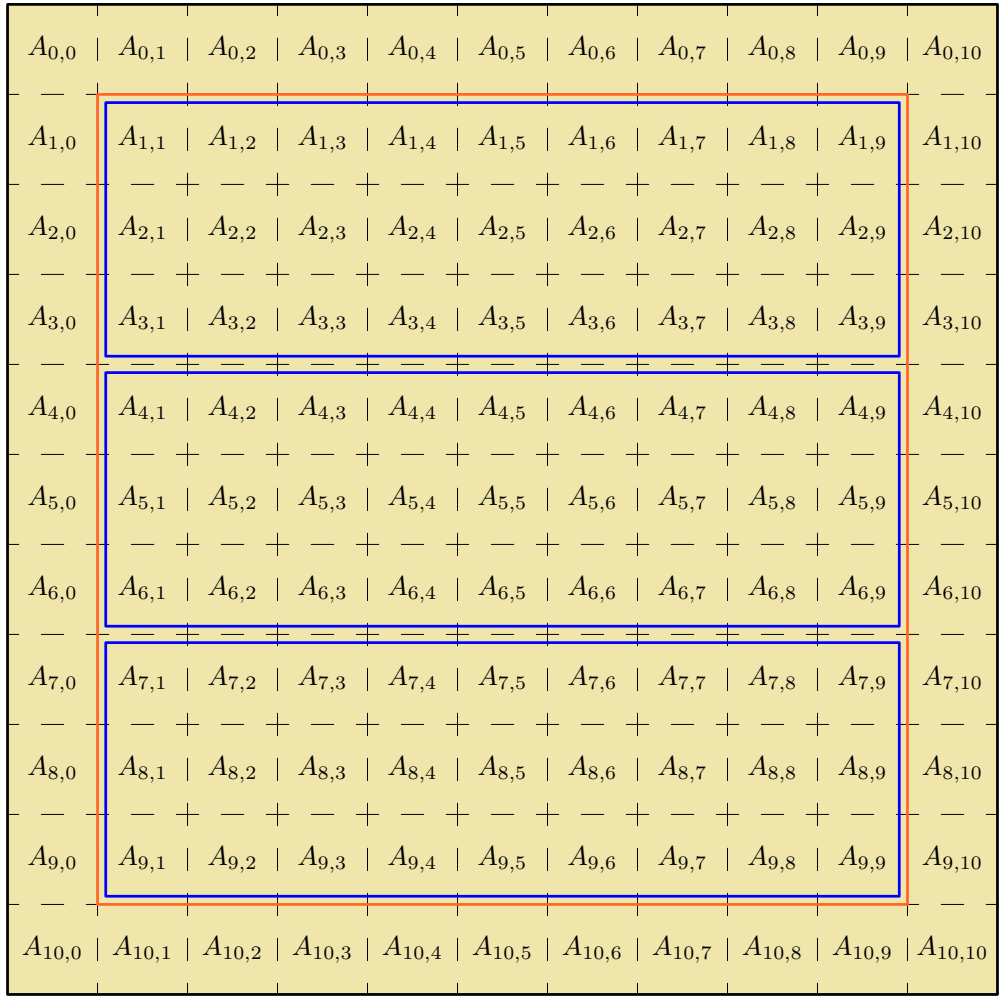
- Gegeben sei eine initialisierte Matrix A.

Eindimensionale Partitionierung



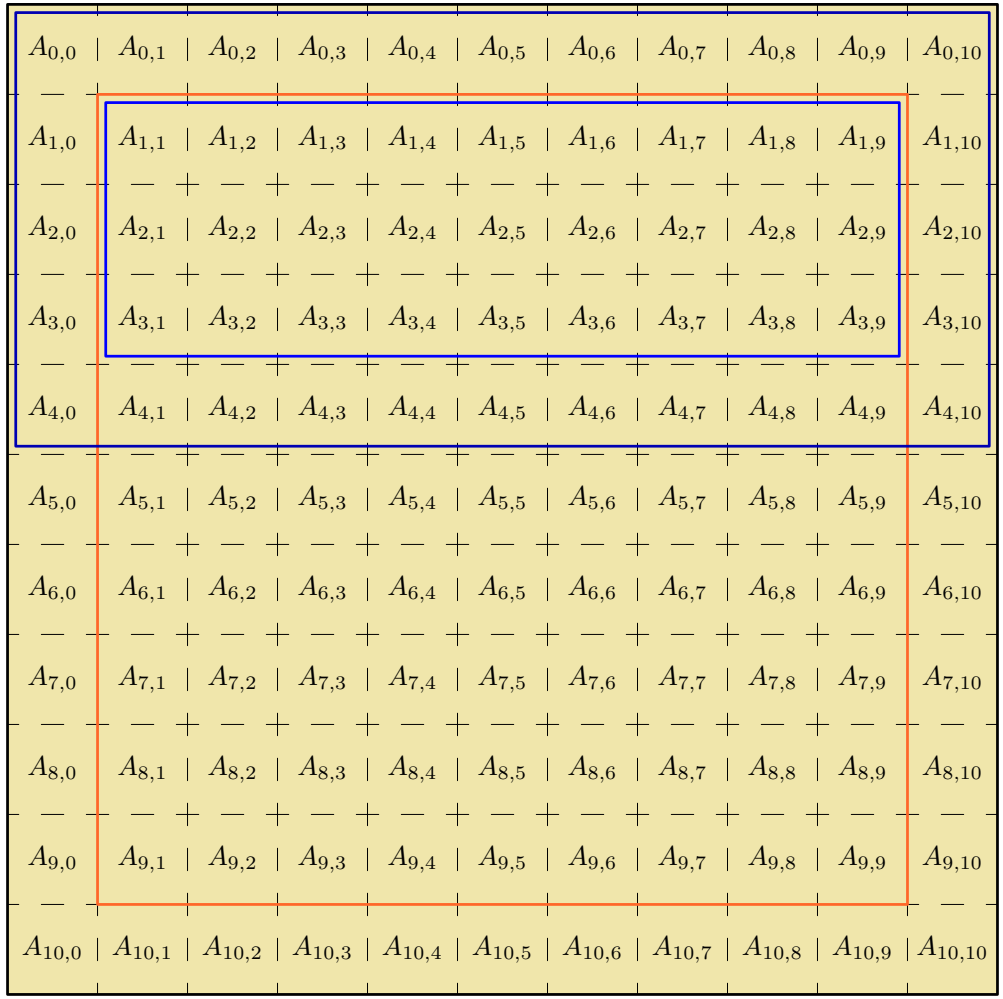
- Der Rand von A ist fest vorgegeben, der innere Teil ist näherungsweise zu bestimmen.

Eindimensionale Partitionierung



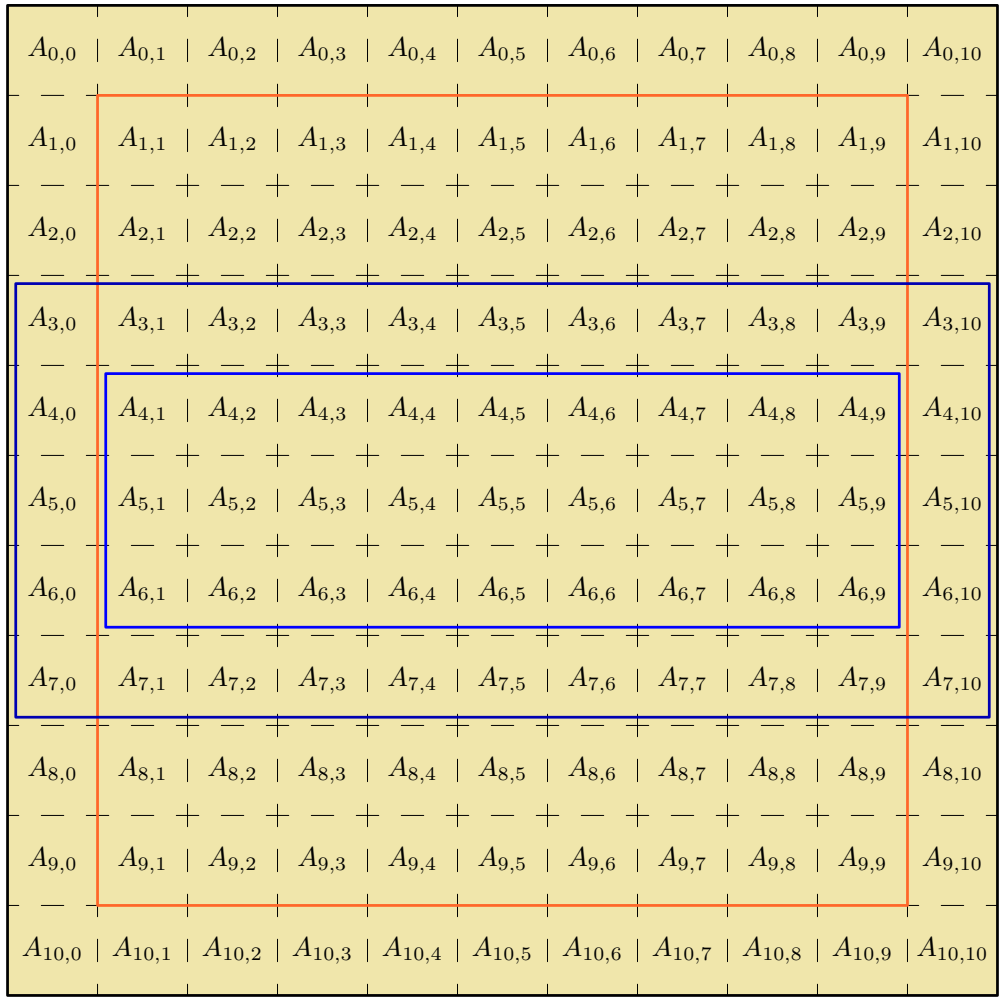
- Der innere Teil von A ist auf m Prozesse (hier $m = 3$) gleichmäßig aufzuteilen.

Eindimensionale Partitionierung

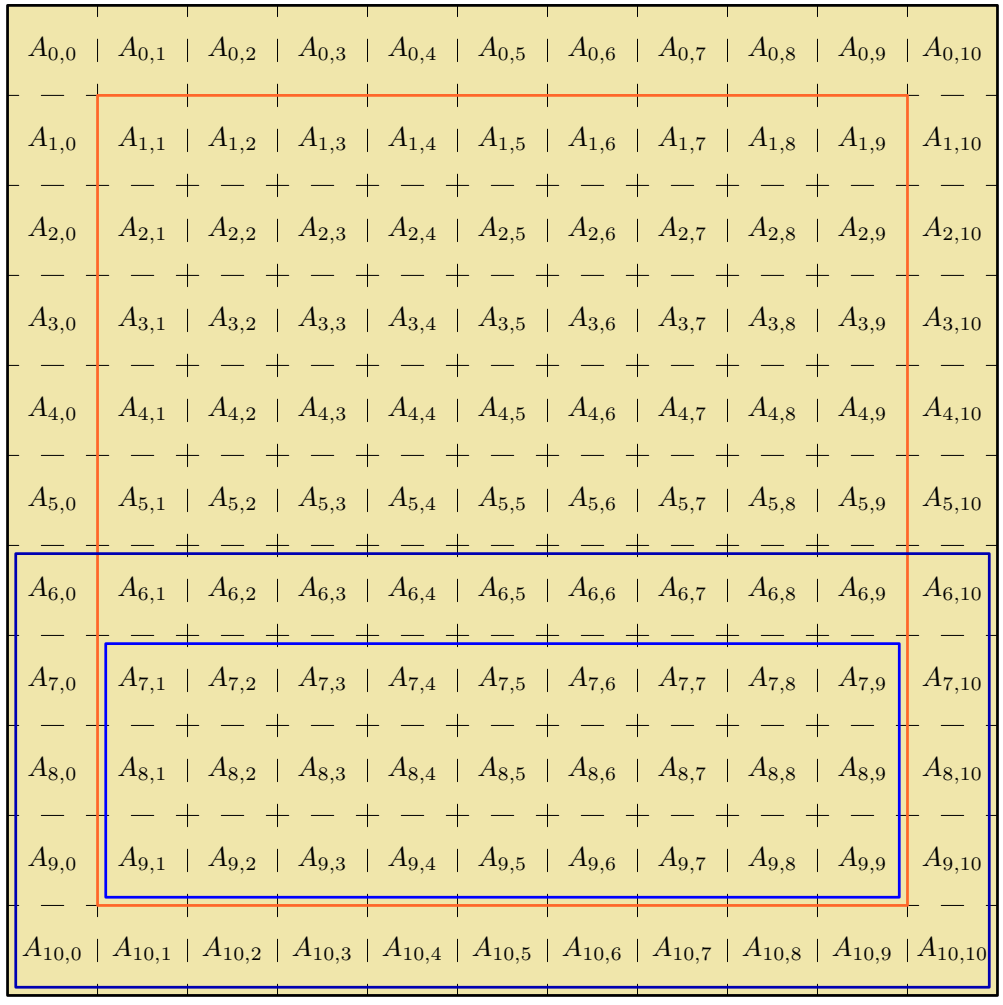


- Jeder der Prozesse benötigt lesenderweise den Rand. Soweit es sich nicht um den äußeren Rand handelt, muss dieser vom jeweiligen Nachbarn nach jedem Iterationsschritt erneut organisiert werden.

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.



- 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.

```
int MPI_Send(void* buf, int count,
             MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm);
```

- MPI-Nachrichten bestehen aus einem Header und der zu versendenden Datenstruktur (*buf*, *count* und *datatype*).
- Der (sichtbare) Header ist ein Tupel bestehend aus der
  - Kommunikationsdomäne (normalerweise *MPI_COMM_WORLD*), dem
  - Absender (*rank* innerhalb der Kommunikationsdomäne) und einer
  - Markierung (*tag*).

```
int MPI_Recv(void* buf, int count,
             MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm,
             MPI_Status* status);
```

Eine mit *MPI_Send* versendete MPI-Nachricht passt zu einem *MPI_Recv* beim Empfänger, falls gilt:

- die Kommunikationsdomänen stimmen überein,
- der Absender stimmt mit *source* überein oder es wurde *MPI_ANY_SOURCE* angegeben,
- die Markierung stimmt mit *tag* überein oder es wurde *MPI_ANY_TAG* angegeben,
- die Datentypen sind identisch und
- die Zahl der Elemente ist kleiner oder gleich der angegebenen Buffergröße.

Was passiert bei *MPI_Send*?

- Wenn die Gegenseite bei einem passenden *MPI_Recv* auf ein Paket wartet, werden die Daten direkt übertragen.
- Wenn die Gegenseite noch nicht in einem passenden *MPI_Recv* wartet, **kann** die Nachricht gepuffert werden. In diesem Falle wird „im Hintergrund“ darauf gewartet, dass die Gegenseite eine passende *MPI_Recv*-Operation ausführt.
- Alternativ kann *MPI_Send* solange blockieren, bis die Gegenseite einen passenden *MPI_Recv*-Aufruf absetzt.
- Wird die Nachricht übertragen oder kommt es zu einer Pufferung, so kehrt *MPI_Send* zurück. D.h. nach dem Aufruf von *MPI_Send* kann in jedem Falle der übergebene Puffer andersweitig verwendet werden.
- Die Pufferung ist durch den Kopieraufwand teuer, ermöglicht aber die frühere Fortsetzung des sendenden Prozesses.
- Ob eine Pufferung zur Verfügung steht oder nicht und welche Kapazität sie ggf. besitzt, ist systemabhängig.

Die Gefahr von Deadlocks bei *MPI_Send*

mpi-deadlock.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 = new double[maxsize];
   for (int len = 1; len <= maxsize; len *= 2) {
      MPI_Send(bigbuf, len, MPI_DOUBLE, other, 0, MPI_COMM_WORLD);
      MPI_Status status;
      MPI_Recv(bigbuf, len, MPI_DOUBLE, other, 0, MPI_COMM_WORLD,
         &status);
      if (rank == 0) cout << "len = " << len << " survived" << endl;
   }
   MPI_Finalize();
}
```

- Hier versuchen die beiden Prozesse 0 und 1 sich erst jeweils etwas zuzusenden, bevor sie *MPI_Recv* aufrufen. Das kann nur mit Pufferung gelingen.

Die Gefahr von Deadlocks bei *MPI_Send*


```
dairinis$ mpirun -np 2 mpi-deadlock
len = 1 survived
len = 2 survived
len = 4 survived
len = 8 survived
len = 16 survived
len = 32 survived
len = 64 survived
len = 128 survived
len = 256 survived
^Cmpirun: killing job...

--------------------------------------------------------------------------
mpirun noticed that process rank 0 with PID 28203 on node dairinis exited on signal 0 (UNKNOWN SIGNAL).
--------------------------------------------------------------------------
2 total processes killed (some possibly by mpirun during cleanup)
mpirun: clean termination accomplished

dairinis$
```

- Hier war die Pufferung nicht in der Lage, eine Nachricht mit 512 Werten des Typs **double** aufzunehmen.
- MPI-Anwendungen, die sich auf eine vorhandene Pufferung verlassen, sind unzulässig bzw. deadlock-gefährdet in Abhängigkeit der lokalen Rahmenbedingungen.

- 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* 120

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*.

Rahmen der parallelisierten Fassung

mpi-jacobi.cpp

```
// 1D-partitioned task
double** run_jacobi_iteration(int rank, int nofprocesses, int N, double eps) {
   int n = N-2;
   assert(nofprocesses <= n);
   int nofrows = n / nofprocesses;
   int remainder = n % nofprocesses;
   int first_row = rank * nofrows + 1;
   if (rank < remainder) {
      ++nofrows;
      if (rank > 0) first_row += rank;
   } else {
      first_row += remainder;
   }
   int last_row = first_row + nofrows - 1;

   // ... initialization ...

   for(;;) {
      double maxdiff = single_jacobi_iteration(A, B, nofrows, n);
      double global_max;
      MPI_Reduce(&maxdiff, &global_max, 1, MPI_DOUBLE,
         MPI_MAX, 0, MPI_COMM_WORLD);
      MPI_Bcast(&global_max, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
      if (global_max < eps) break;
      // ... message exchange with neighbors ...
   }

   // ... collect results in process 0 ...
   return Result;
}
```

mpi-jacobi.cpp

```
double** A = new double*[nofrows+2];
for (int i = 0; i <= nofrows+1; ++i) {
   A[i] = new double[N];
   for (int j = 0; j < N; ++j) {
      initialize_A(A[i][j], i + first_row, j, N);
   }
}
double** B = new double*[nofrows+1];
for (int i = 1; i <= nofrows; ++i) {
   B[i] = new double[N-1];
}
```

- Speicher für A und B wird hier nur im benötigten Umfang der entsprechenden Teilmatrizen belegt.
- A enthält auch die Randzonen von den jeweiligen Nachbarn. Bei B entfällt dies. Aus Gründen der Einfachheit werden aber A und B auf gleiche Weise indiziert.

mpi-jacobi.cpp

```
double global_max;
MPI_Reduce(&maxdiff, &global_max, 1, MPI_DOUBLE,
   MPI_MAX, 0, MPI_COMM_WORLD);
MPI_Bcast(&global_max, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
if (global_max < eps) break;
```

- Zuerst wird das globale Maximum festgestellt, indem aus den lokalen Maxima mit *MPI_Reduce* im Prozess 0 das globale Maximum festgestellt wird und dieses danach mit *MPI_Bcast* an alle Prozesse übermittelt wird.
- Sobald die maximale Abweichung klein genug ist, wird die Schleife abgebrochen und auf eine weitere Kommunikation zwischen den Nachbarn verzichtet.

```
MPI_Status status;
// send highest row to the process which is next in rank
if (rank == 0) {
   MPI_Send(A[nofrows] + 1, n, MPI_DOUBLE, rank+1, 0,
      MPI_COMM_WORLD);
} else if (rank == nofprocesses-1) {
   MPI_Recv(A[0] + 1, n, MPI_DOUBLE, rank-1, 0,
      MPI_COMM_WORLD, &status);
} else {
   MPI_Sendrecv(A[nofrows] + 1, n, MPI_DOUBLE, rank+1, 0,
      A[0] + 1, n, MPI_DOUBLE, rank-1, 0,
      MPI_COMM_WORLD, &status);
}
```

- Das ist die Umsetzung der ersten Welle, bei der jeweils eine Zeile zur nächsthöheren Prozessnummer übermittelt wird bzw. eine Zeile von der nächstniedrigeren Prozessnummer entgegenzunehmen ist.

```
// send lowest row to the process which is previous in rank
if (rank == 0) {
   MPI_Recv(A[nofrows+1] + 1, n, MPI_DOUBLE, rank+1, 0,
      MPI_COMM_WORLD, &status);
} else if (rank == nofprocesses-1) {
   MPI_Send(A[1] + 1, n, MPI_DOUBLE, rank-1, 0, MPI_COMM_WORLD);
} else {
   MPI_Sendrecv(A[1] + 1, n, MPI_DOUBLE, rank-1, 0,
      A[nofrows+1] + 1, n, MPI_DOUBLE, rank+1, 0,
      MPI_COMM_WORLD, &status);
}
```

- Danach folgt die Rückwelle, bei der jeweils eine Zeile zur nächstniedrigeren Prozessnummer übermittelt wird bzw. eine Zeile von der nächsthöheren Prozessnummer entgegenzunehmen ist.

Vereinfachung mit Nullprozessen

mpi-jacobi.cpp

```
int previous = rank == 0? MPI_PROC_NULL: rank-1;
int next = rank == nofprocesses-1? MPI_PROC_NULL: rank+1;
for(;;) {
   double maxdiff = single_jacobi_iteration(A, B, nofrows, n);
   double global_max;
   MPI_Reduce(&maxdiff, &global_max, 1, MPI_DOUBLE,
      MPI_MAX, 0, MPI_COMM_WORLD);
   MPI_Bcast(&global_max, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   if (global_max < eps) break;
   MPI_Status status;
   // send highest row to the process which is next in rank
   MPI_Sendrecv(A[nofrows] + 1, n, MPI_DOUBLE, next, 0,
      A[0] + 1, n, MPI_DOUBLE, previous, 0, MPI_COMM_WORLD, &status);
   // send lowest row to the process which is previous in rank
   MPI_Sendrecv(A[1] + 1, n, MPI_DOUBLE, previous, 0,
      A[nofrows+1] + 1, n, MPI_DOUBLE, next, 0, MPI_COMM_WORLD, &status);
}
```

- Die Behandlung der Spezialfälle am Rand lässt sich durch den Einsatz von sogenannten Nullprozessen vermeiden.
- Eine Kommunikation mit dem Prozess *MPI_PROC_NULL* findet nicht statt.

Einsammeln der Ergebnisse

mpi-jacobi.cpp

```
// collect results in process 0
double** Result = 0;
if (rank == 0) {
   Result = new double*[N]; assert(Result);
   for (int i = 0; i < N; ++i) {
      Result[i] = new double[N]; assert(Result[i]);
      for (int j = 0; j < N; ++j) {
         initialize_A(Result[i][j], i, j, N);
      }
   }
   for (int i = 1; i <= last_row; ++i) {
      memcpy(Result[i] + 1, A[i] + 1, n * sizeof(double));
   }
   for (int i = last_row+1; i <= n; ++i) {
      MPI_Status status;
      MPI_Recv(Result[i] + 1, n, MPI_DOUBLE,
         MPI_ANY_SOURCE, i, MPI_COMM_WORLD, &status);
   }
} else {
   for (int i = 1; i <= nofrows; ++i) {
      MPI_Send(A[i] + 1, n, MPI_DOUBLE, 0, first_row + i - 1,
         MPI_COMM_WORLD);
   }
}
return Result;
```

- 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.

mpi-jacobi-nb.cpp

```
// compute border zone
for (int j = 1; j <= n; ++j) {
   B[1][j] = 0.25 * (A[0][j] + A[1][j-1] + A[1][j+1] + A[2][j]);
   B[nofrows][j] = 0.25 * (A[nofrows-1][j] + A[nofrows][j-1] +
                           A[nofrows][j+1] + A[nofrows+1][j]);
}
// initiate non-blocking communication
MPI_Request req[4];
MPI_Irecv(A[0] + 1, n, MPI_DOUBLE, previous, 0, MPI_COMM_WORLD, &req[0]);
MPI_Irecv(A[nofrows+1] + 1, n, MPI_DOUBLE, next, 0, MPI_COMM_WORLD, &req[1]);
MPI_Isend(B[1] + 1, n, MPI_DOUBLE, previous, 0, MPI_COMM_WORLD, &req[2]);
MPI_Isend(B[nofrows] + 1, n, MPI_DOUBLE, next, 0, MPI_COMM_WORLD, &req[3]);
// computer inner zone
for (int i = 2; i < nofrows; ++i) {
   for (int j = 1; j <= n; ++j) {
      B[i][j] = 0.25 * (A[i-1][j] + A[i][j-1] + A[i][j+1] + A[i+1][j]);
   }
}
// prepare next iteration and compute maxdiff
double maxdiff = 0;
for (int i = 1; i <= nofrows; ++i) {
   for (int j = 1; j <= n; ++j) {
      double diff = fabs(A[i][j] - B[i][j]);
      if (diff > maxdiff) maxdiff = diff;
      A[i][j] = B[i][j];
   }
}
// block until initiated communication is finished
MPI_Status status; for (int i = 0; i < 4; ++i) MPI_Wait(&req[i], &status);
```

mpi-jacobi-nb.cpp

```
MPI_Request req[4];
MPI_Irecv(A[0] + 1, n, MPI_DOUBLE, previous, 0,
   MPI_COMM_WORLD, &req[0]);
MPI_Irecv(A[nofrows+1] + 1, n, MPI_DOUBLE, next, 0,
   MPI_COMM_WORLD, &req[1]);
MPI_Isend(B[1] + 1, n, MPI_DOUBLE, previous, 0,
   MPI_COMM_WORLD, &req[2]);
MPI_Isend(B[nofrows] + 1, n, MPI_DOUBLE, next, 0,
   MPI_COMM_WORLD, &req[3]);
```

- 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 131

mpi-jacobi-nb.cpp

```
// block until initiated communication is finished
for (int i = 0; i < 4; ++i) {
   MPI_Status status;
   MPI_Wait(&req[i], &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.

- 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

```
Matrix*
run_jacobi_iteration(int rank, int nofprocesses, int N, double eps) {
   int n = N - 2; // without the surrounding border
   // create two-dimensional Cartesian grid
   int dims[2] = {0, 0}; int periods[2] = {false, false};
   MPI_Dims_create(nofprocesses, 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(MPI_COMM_WORLD, &rank); // update rank
   // ...
}
```

- 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};
MPI_Dims_create(nofprocesses, 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 periods[2] = {false, false};
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

```
// locate our own submatrix
int first_row, nof_rows, first_col, nof_cols;
get_submatrix(grid, dims, n, rank,
              first_row, first_col, nof_rows, nof_cols);

Matrix A(nof_rows + 2, nof_cols + 2, first_row, first_col);
Matrix B(nof_rows, nof_cols, first_row + 1, first_col + 1);
for (int i = A.firstRow(); i <= A.lastRow(); ++i) {
   for (int j = A.firstCol(); j <= A.lastCol(); ++j) {
      initialize_A(A(i, j), i, j, N);
   }
}
```

- *Matrix* ist eine Instantiierung der *GeMatrix*-Template-Klasse aus dem FLENS-Paket:
  typedef *flens*::*GeMatrix*<*flens*::*FullStorage*<**double**, *cxxblas*::*RowMajor*> > *Matrix*;
- *get_submatrix* ermittelt die Koordinaten der eigenen Teilmatrix, so dass dann A und B entsprechend deklariert werden können.

mpi-jacobi-2d.cpp

```
void get_submatrix(const MPI_Comm& grid, int* dims,
                   int n, int rank,
                   int& first_row, int& first_col,
                   int& nof_rows, int& nof_cols) {
   int coords[2];
   MPI_Cart_coords(grid, rank, 2, coords); // retrieve our position
   get_partition(n, dims[0], coords[0], first_row, nof_rows);
   get_partition(n, dims[1], coords[1], first_col, nof_cols);
}
```

- Die Funktion *MPI_Cart_coords* liefert die Gitterkoordinaten (in *coords*) für einen Prozess (*rank*).
- Der dritte Parameter gibt die Zahl der Dimensionen an, die niedriger als die Zahl der Dimensionen des Gitters sein kann.

mpi-jacobi-2d.cpp

```
void get_partition(int len, int nofprocesses, int rank,
                   int& start, int& locallen) {
   locallen = (len - rank - 1) / nofprocesses + 1;
   int share = len / nofprocesses;
   int remainder = len % nofprocesses;
   start = rank * share + (rank < remainder? rank: remainder);
}
```

- Diese Hilfsfunktion ermittelt in üblicher Weise das Teilintervall $[start, start + locallen - 1]$ aus dem Gesamtintervall $[0, len - 1]$ für den Prozess *rank*.

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

```
MPI_Datatype vector_type(int len, int stride) {
   MPI_Datatype datatype;
   MPI_Type_vector(
      /* count = */ len,
      /* blocklength = */ 1,
      /* stride = */ stride,
      /* element type = */ MPI_DOUBLE,
      /* newly created type = */ &datatype);
   MPI_Type_commit(&datatype);
   return datatype;
}
```

- Da dem linken und rechten Nachbarn in einem zweidimensionalen Gitter jeweils Spaltenvektoren zu übermitteln sind, lässt sich dies nicht mehr mit dem vorgegebenen Datentyp *MPI_DOUBLE* und einer Anfangsadresse erreichen.
- Um solche Probleme zu lösen, können mit Hilfe einiger Typkonstruktoren eigene Datentypen definiert werden.

- Es gibt die Menge der Basistypen BT in MPI, der beispielsweise *MPI_DOUBLE* oder *MPI_INT* angehören.
- Ein Datentyp T mit der Kardinalität n ist in der MPI-Bibliothek eine Sequenz von Tupeln $\{(bt_1, o_1), (bt_2, o_2), \ldots, (bt_n, o_n)\}$, mit $bt_i \in BT$ und den zugehörigen Offsets $o_i \in \mathbb{N}_0$ für $i = 1, \ldots, n$.
- Die Offsets geben die relative Position der jeweiligen Basiskomponenten zur Anfangsadresse an.
- Bezüglich der Kompatibilität bei *MPI_Send* und *MPI_Recv* sind zwei Datentypen T_1 und T_2 genau dann kompatibel, falls die beiden Kardinalitäten n_1 und n_2 gleich sind und $bt_{1_i} = bt_{2_i}$ für alle $i = 1, \ldots, n_1$ gilt.
- Bei *MPI_Send* sind Überlappungen zulässig, bei *MPI_Recv* haben sie einen undefinierten Effekt.
- Alle Datentypobjekte haben in der MPI-Bibliothek den Typ *MPI_Datatype*.

- Ein Zeilenvektor des Basistyps *MPI_DOUBLE* (8 Bytes) der Länge 4 hat den Datenyp $\{(DOUBLE, 0), (DOUBLE, 8), (DOUBLE, 16), (DOUBLE, 24)\}$.
- Ein Spaltenvektor der Länge 3 aus einer 5×5-Matrix hat den Datentyp $\{(DOUBLE, 0), (DOUBLE, 40), (DOUBLE, 80)\}$.
- Die Spur einer 3×3-Matrix hat den Datentyp $\{(DOUBLE, 0), (DOUBLE, 32), (DOUBLE, 64)\}$.
- Die obere Dreiecks-Matrix einer 3×3-Matrix: $\{(DOUBLE, 0), (DOUBLE, 8), (DOUBLE, 16),$ $(DOUBLE, 32), (DOUBLE, 40), (DOUBLE, 64)\}$

Alle Konstruktoren sind Funktionen, die als letzte Parameter den zu verwenden Elementtyp und einen Zeiger auf den zurückzuliefernden Typ erhalten:

MPI_Type_contiguous(*count*, *elemtype*, *newtype*)
: zusammenhängender Vektor aus *count* Elementen

MPI_Type_vector(*count*, *blocklength*, *stride*, *elemtype*, *newtype*)
: *count* Blöcke mit jeweils *blocklength* Elementen, deren Anfänge jeweils *stride* Elemente voneinander entfernt sind

MPI_Type_indexed(*count*, *blocklengths*, *offsets*, *elemtype*, *newtype*)
: *count* Blöcke mit jeweils individuellen Längen und Offsets

MPI_Type_create_struct(*count*, *blocklengths*, *offsets*, *elemtypes*, *newtype*)
: analog zu *MPI_Type_indexed*, aber jeweils mit individuellen Typen

mpi-jacobi-2d.cpp

```
struct buffer {
   double* buf;
   MPI_Datatype type;
};
struct buffer in_vectors[] = {
   {&A(A.firstRow(), A.firstCol() + 1), vector_type(nof_cols, 1)},
   {&A(A.lastRow(), A.firstCol() + 1), vector_type(nof_cols, 1)},
   {&A(A.firstRow() + 1, A.firstCol()),
      vector_type(nof_rows, nof_cols + 2)},
   {&A(A.firstRow() + 1, A.lastCol()),
      vector_type(nof_rows, nof_cols + 2)}
};
struct buffer out_vectors[] = {
   {&B(B.lastRow(), B.firstCol()), vector_type(nof_cols, 1)},
   {&B(B.firstRow(), B.firstCol()), vector_type(nof_cols, 1)},
   {&B(B.firstRow(), B.lastCol()), vector_type(nof_rows, nof_cols)},
   {&B(B.firstRow(), B.firstCol()), vector_type(nof_rows, nof_cols)}
};
```

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

mpi-jacobi-2d.cpp

```
int in_neighbor[] = {upper, lower, left, right};
int out_neighbor[] = {lower, upper, right, left};
```

- Passend zu den Austauschvektoren werden die zugehörigen Prozessnummern der Nachbarn spezifiziert.
- Dabei ist zu beachten, dass die Austauschvektoren bzw. die zugehörigen Prozessnummern jeweils paarweise zusammenpassen. (Darauf kann nur verzichtet werden, wenn die gesamte Kommunikation nicht-blockierend ist.)

mpi-jacobi-2d.cpp

```
for(;;) {
   double maxdiff = single_jacobi_iteration(A, B);
   double global_max;
   MPI_Reduce(&maxdiff, &global_max, 1, MPI_DOUBLE,
      MPI_MAX, 0, MPI_COMM_WORLD);
   MPI_Bcast(&global_max, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   if (global_max < eps) break;

   // exchange borders with our neighbors
   for (int dir = 0; dir < 4; ++dir) {
      MPI_Status status;
      MPI_Sendrecv(
         out_vectors[dir].buf, 1, out_vectors[dir].type,
            out_neighbor[dir], 0,
         in_vectors[dir].buf, 1, in_vectors[dir].type,
            in_neighbor[dir], 0,
         MPI_COMM_WORLD, &status
      );
   }
}
```

mpi-jacobi-2d-nb.cpp

```
double global_max;
do {
   // compute border zones
   // ...
   // exchange borders with our neighbors
   // ...
   // computer inner region
   // ...
   // block until initiated communication is finished
   // ...
   // check remaining error
   // ...
} while (global_max > eps);
```

- 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-nb.cpp

```
// compute border zones
for (int j = B.firstCol(); j <= B.lastCol(); ++j) {
   int i = B.firstRow();
   B(i,j) = 0.25 * (A(i-1,j) + A(i,j-1) + A(i,j+1) + A(i+1,j));
   i = B.lastRow();
   B(i,j) = 0.25 * (A(i-1,j) + A(i,j-1) + A(i,j+1) + A(i+1,j));
}
for (int i = B.firstRow(); i <= B.lastRow(); ++i) {
   int j = B.firstCol();
   B(i,j) = 0.25 * (A(i-1,j) + A(i,j-1) + A(i,j+1) + A(i+1,j));
   j = B.lastCol();
   B(i,j) = 0.25 * (A(i-1,j) + A(i,j-1) + A(i,j+1) + A(i+1,j));
}
```

mpi-jacobi-2d-nb.cpp

```
// exchange borders with our neighbors
MPI_Request req[8];
for (int dir = 0; dir < 4; ++dir) {
   MPI_Isend(out_vectors[dir].buf, 1, out_vectors[dir].type,
         out_neighbor[dir], 0, MPI_COMM_WORLD, &req[dir*2]);
   MPI_Irecv(in_vectors[dir].buf, 1, in_vectors[dir].type,
         in_neighbor[dir], 0, MPI_COMM_WORLD, &req[dir*2+1]);
}
// compute inner region
for (int i = B.firstRow() + 1; i < B.lastRow(); ++i) {
   for (int j = B.firstCol() + 1; j < B.lastCol(); ++j) {
      B(i,j) = 0.25 * (A(i-1,j) + A(i,j-1) + A(i,j+1) + A(i+1,j));
   }
}
```

mpi-jacobi-2d.cpp

```
MPI_Datatype matrix_type(const Matrix::View& submatrix) {
   MPI_Datatype datatype; MPI_Type_vector(
      /* count = */ submatrix.numRows(),
      /* blocklength = */ submatrix.numCols(),
      /* stride = */ submatrix.engine().leadingDimension(),
      /* element type = */ MPI_DOUBLE, &datatype);
   MPI_Type_commit(&datatype); return datatype;
}
```

- Am Ende werden die einzelnen Teilmatrizen vom Prozess 0 eingesammelt.
- Hierzu werden vom Prozess 0 jeweils passende Sichten erzeugt (*Matrix*::*View*), für die *matrix_type* jeweils den passenden Datentyp generiert.
- Die Methode *leadingDimension* liefert hier den Abstand zwischen zwei aufeinanderfolgenden Zeilen. Da es sich hier um eine Teilmatrix handelt, kann dieser Abstand deutlich größer als *submatrix.numCols*() sein.

Einsammeln der Ergebnisse

mpi-jacobi-2d.cpp

```
Matrix* Result = 0;
if (rank == 0) {
   Result = new Matrix(N, N, 0, 0); assert(Result);
   for (int i = 0; i < N; ++i) {
      for (int j = 0; j < N; ++j) {
         initialize_A((*Result)(i,j), i, j, N);
      }
   }
   for (int p = 0; p < nofprocesses; ++p) {
      int first_row, first_col, nof_rows, nof_cols;
      get_submatrix(grid, dims, n, p,
         first_row, first_col, nof_rows, nof_cols);
      ++first_row; ++first_col;
      Matrix::View submatrix(Result->engine().view(first_row, first_col,
                             first_row + nof_rows - 1,
                             first_col + nof_cols - 1,
                             first_row, first_col));
      if (p == 0) {
         submatrix = B;
      } else {
         MPI_Status status;
         MPI_Recv(&submatrix(submatrix.firstRow(), submatrix.firstCol()),
            1, matrix_type(submatrix), p, 0,
            MPI_COMM_WORLD, &status);
      }
   }
} else {
   MPI_Send(&B(B.firstRow(), B.firstCol()),
      B.numRows() * B.numCols(), MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
}
return Result;
```