```
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* 299

mpi-deadlock.cpp

```
int main(int argc, char** argv) {
   alarm(1); // terminate ourselves if we need more than 1s
   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;
   constexpr 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) std::cout << "len = " << len << " survived" << std::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*


```
theon$ 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
--------------------------------------------------------------------------
mpirun noticed that process rank 1 with PID 0 on node theon exited on signal 14 (Alarm Clock).
--------------------------------------------------------------------------
theon$
```

- 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.

Die Prozesse P_0 und P_1 wollen jeweils zuerst senden und erst danach empfangen:

$$
\begin{aligned}
P &= P_0 \parallel P_1 \parallel \mathit{Network} \\
P_0 &= (p_0\mathit{SendsMsg} \rightarrow p_0\mathit{ReceivesMsg} \rightarrow P_0) \\
P_1 &= (p_1\mathit{SendsMsg} \rightarrow p_1\mathit{ReceivesMsg} \rightarrow P_1) \\
\mathit{Network} &= (p_0\mathit{SendsMsg} \rightarrow p_1\mathit{ReceivesMsg} \rightarrow \mathit{Network} \mid \\
&\qquad p_1\mathit{SendsMsg} \rightarrow p_0\mathit{ReceivesMsg} \rightarrow \mathit{Network})
\end{aligned}
$$

Das gleiche Szenario, bei dem P_0 und P_1 jeweils zuerst senden und dann empfangen, diesmal aber mit den Puffern $P_0 Buffer$ und $P_1 Buffer$:

$$
\begin{aligned}
P &= P_0 \parallel P_1 \parallel P_0 Buffer \parallel P_1 Buffer \parallel Network \\
P_0 &= (p_0 SendsMsg \rightarrow p_0 ReceivesMsgFromBuffer \rightarrow P_0) \\
P_0 Buffer &= (p_0 ReceivesMsg \rightarrow p_0 ReceivesMsgFromBuffer \rightarrow \\
&\quad P_0 Buffer) \\
P_1 &= (p_1 SendsMsg \rightarrow p_1 ReceivesMsgFromBuffer \rightarrow P_1) \\
P_1 Buffer &= (p_1 ReceivesMsg \rightarrow p_1 ReceivesMsgFromBuffer \rightarrow \\
&\quad P_1 Buffer) \\
Network &= (p_0 SendsMsg \rightarrow p_1 ReceivesMsg \rightarrow Network \mid \\
&\quad p_1 SendsMsg \rightarrow p_0 ReceivesMsg \rightarrow Network)
\end{aligned}
$$

Zu den häufigen Kommunikationsszenarien analog zum Fork-und-Join-Pattern gehören

- die Aufteilung eines Vektors oder einer Matrix an alle beteiligten Prozesse (*scatter*) und
- nach der Durchführung der verteilten Berechnungen das Einsammeln aller Ergebnisse, die wieder zu einem Vektor oder einer Matrix zusammengefasst werden sollen (*gather*).

Dies ließe sich mit *MPI_Send* und *MPI_Recv* erledigen, wobei ein ausgewählter Prozess (typischerweise mit *rank* 0) dann wiederholt *MPI_Send* bzw. *MPI_Recv* aufrufen müsste.
Die wegen der begrenzten Pufferung sich aufaddierenden Latenzzeiten lassen sich reduzieren, wenn das Parallelisierungspotential durch optimierte Operationen ausgenutzt wird.

mpi-gemv-sg.cpp

```
int main(int argc, char** argv) {
   MPI_Init(&argc, &argv);
   int rank; MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   int n; double* A = nullptr; double* x = nullptr; double* y = nullptr;
   if (rank == 0) {
      // process arguments where we expect an input file (in) ...
      if (!read_parameters(in, n, A, x)) {
         std::cerr << "Invalid input!" << std::endl;
         MPI_Abort(MPI_COMM_WORLD, 1);
      }
      y = new double[n];
   }
   mpi_gemv(n, A, x, y); MPI_Finalize();
   if (rank == 0) {
      for (int i = 0; i < n; ++i) {
         std::cout << " " << y[i] << std::endl;
      }
   }
}
```

- Als Beispiel wird hier wieder die Matrix-Vektor-Multiplikation verwendet, weil das Verfahren gut demonstriert, nicht weil es ansonsten sinnvoll wäre.

mpi-gemv-sg.cpp

```
static void mpi_gemv(int n, double* A, double* x, double* y) {
   /* ... preparation ... */

   /* ... scatter rows of A among all participating processes ... */

   /* ... compute assigned part of the resulting vector ... */

   /* ... gather results ... */

   /* ... clean up ... */
}
```

- Die Funktion *mpi_gemv* wird von allen beteiligten Prozessen gemeinsam aufgerufen.
- Zu beachten ist, dass die Parameter nur beim Prozess 0 sinnvoll gefüllt sind.

Vorbereitung bei Scatter/Gather

mpi-gemv-sg.cpp

```
int rank; MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (!x) {
   assert(rank > 0);
   x = new double[n];
}
MPI_Bcast(x, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

int nofprocesses; MPI_Comm_size(MPI_COMM_WORLD, &nofprocesses);

int nofrows = n / nofprocesses;
int remainder = n % nofprocesses;
if (rank < remainder) {
   ++nofrows;
}
```

- Zunächst müssen n und der Vektor x an alle Prozesse verbreitet werden.
- Da nicht sichergestellt ist, dass n durch die Zahl der Prozesse teilbar ist, können die Zahl der zu bearbeitenden Matrixzeilen für die einzelnen Prozesse unterschiedlich sein.

mpi-gemv-sg.cpp

```
/* scatter rows of A among all participating processes */
int* counts = nullptr; int* displs = nullptr;
if (rank == 0) {
   counts = new int[nofprocesses]; displs = new int[nofprocesses];
   int offset = 0;
   for (int i = 0; i < nofprocesses; ++i) {
      displs[i] = offset; counts[i] = n / nofprocesses;
      if (i < remainder) {
         ++counts[i];
      }
      counts[i] *= n; offset += counts[i];
   }
}
double* myrows = new double[nofrows * n];
MPI_Scatterv(A, counts, displs, MPI_DOUBLE,
   myrows, nofrows * n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

- Wenn einheitlich aufgeteilt wird, können die einfacheren Funktionen *MPI_Scatter* und *MPI_Gather* verwendet werden.
- Hier werden die Arrays *counts* und *displs* verwendet, die festlegen, wieviel Elemente (hier vom Typ *MPI_DOUBLE*) jeder Prozess erhält und ab welchem Offset in A diese zu finden sind.

mpi-gemv-sg.cpp

```
/* compute assigned part of the resulting vector */
double* result = new double[nofrows];
for (unsigned int i = 0; i < nofrows; ++i) {
   double val = 0;
   for (unsigned int j = 0; j < n; ++j) {
      val += myrows[i*n + j] * x[j];
   }
   result[i] = val;
}
```

- Alle Prozesse einschließlich dem verteilenden Prozess 0 arbeiten jetzt mit *myrows*, dem Matrix-Ausschnitt, der mit *MPI_Scatter* verteilt wurde.

mpi-gemv-sg.cpp

```
/* gather results */
if (rank == 0) {
   int offset = 0;
   for (int i = 0; i < nofprocesses; ++i) {
      displs[i] = offset;
      counts[i] = n / nofprocesses;
      if (i < remainder) {
         ++counts[i];
      }
      offset += counts[i];
   }
}
MPI_Gatherv(result, nofrows, MPI_DOUBLE,
   y, counts, displs, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

- *counts* und *displs* müssen neu berechnet werden, da jetzt nur noch ein Vektor zusammengesetzt wird.
- Die Zuordnung der jeweiligen Ausschnitte beim Verteilen und Aufsammeln erfolgt strikt nach der Rangordnung der Prozesse.

Im folgenden wird unsere Vorlesungsbibliothek aus HPC I von Michael Lehn und mir verwendet, die insbesondere Matrizen und Vektoren unterstützt:

- Sie steht auf unseren Maschinen unter */home/numerik/pub/pp/ss19/lib* zur Verfügung.
- Sie lässt sich herunterladen:
  `http://www.mathematik.uni-ulm.de/numerik/pp/ss19/hpc.tar.gz`
- Sie besteht nur aus Headern unterhalb eines Verzeichnisses mit dem Namen *hpc*.
- Beim Übersetzen ist dann nur „-I“ anzugeben mit dem Verzeichnis, worunter das *hpc*-Verzeichnis liegt.

Zusätzlich wird *fmt*::*printf* verwendet, das auf unseren Maschinen installiert ist und unter `https://github.com/afborchert/fmt` zur Verfügung steht.

In der HPC-Bibliothek belegen Vektoren und Matrizen nicht mehr notwendigerweise eine zusammenhängende Speicherfläche. Stattdessen wird mit relativen Abständen gearbeitet, um zum nächsten Element in einer Dimension zu gelangen.
Beispiel: Sei

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

und folgende Deklaration gegeben:

```
using namespace hpc::matvec;
using T = double;
using Matrix = GeMatrix<T>;
Matrix A(2, 3, StorageOrder::RowMajor);
for (auto [i, j, Aij]: A) {
   Aij = 1 + i * A.numCols() + j;
});
```

Dann beträgt der Abstand zwischen $A_{i,j}$ und $A_{i,j+1}$ 1 (*incCol*) und der Abstand zwischen $A_{i,j}$ und $A_{i+1,j}$ hat den Wert 3 (*incRow*).

Wenn die mittlere Spalte ausgewählt wird mit

```
auto column = A.col(1);
```

dann beträgt der Abstand zwischen aufeinanderfolgenden Elementen 3.

Oder wenn eine Teilmatrix ausgewählt wird (mit der zweiten und dritten Spalte) mittels

```
auto A_ = A.block(0, 1).dim(2, 2);
```

(Parameter: Indizes des ersten Elements bei *block*, gefolgt bei *dim* von der Zahl der Zeilen und Spalten), dann bleiben *incCol* bei 1 bzw. *incRow* bei 3. Anders als zuvor liegt die Teilmatrix nicht mehr zusammenhängend im Speicher, da sie nur Teil einer größeren Matrix ist.

Das bedeutet, dass einfache *MPI_Send* bzw. *MPI_Recv*-Operationen mit einem der Elementartypen (wie etwa *MPI_DOUBLE*) einen Vektor oder eine Matrix im allgemeinen Fall nicht übertragen können. Entsprechend müssen neue MPI-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 $o_i \in \mathbb{Z}$ 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

Bei der Übertragung muss bezüglich der Reihenfolge einheitlich festgelegt werden, ob wir mit *row major* oder *col major* operieren. Im folgenden gehen wir von *row major* aus.

Für den Sonderfall, dass *incCol* den Wert 1 hat, d.h. dass innerhalb einer Zeile die Elemente unmittelbar hintereinander im Speicher liegen, lässt sich der Datentyp mit einem einzigen Aufruf von *MPI_Type_vector* erzeugen:

```
MPI_Datatype datatype;
MPI_Type_vector(
   /* count = */ A.numRows(),
   /* blocklength = */ A.numCols(),
   /* stride = */ A.incRow(),
   /* element type = */ get_type(A(0, 0)),
   /* newly created type = */ &datatype);
```

- Ein Matrix-Typ kann nicht einfach als Array von Vektoren definiert werden, wenn unklar ist, wie das funktioniert.
- Wenn bei einer Matrix *incRow* den Wert 1 hat (*col major*) und die Matrix in *row major* übertragen wird, dann überlappen sich die Speicherbereiche der Zeilen.

Sei T ein Datentyp mit $T = \{(et_1, o_1), (et_2, o_2), \ldots, (et_n, o_n)\}$:

- Für jeden Datentyp T sei der Umfang extent(T) definiert. Bei elementaren Datentypen wird das Resultat von **sizeof** verwendet. Beispiel: extent(MPI_DOUBLE) = 8.
- Die untere Schranke lb sei wie folgt definiert:

$$\text{lb}(T) = \min_{1 \le i \le n} \{o_i\}$$

- Die obere Schranke berücksichtigt den Umfang:

$$\text{ub}(T) = \max_{1 \le i \le n} \{o_i + \text{extent}(et_i)\} + \epsilon$$

  Hierbei ist ϵ als notwendige Größe zum Aufrunden auf die nächste Alignment-Kante anzusehen.

- Dann lässt sich der Umfang allgemein definieren:

$$\text{extent}(T) = \text{ub}(T) - \text{lb}(T)$$

Wenn beispielsweise durch *MPI_Type_vector* mehrere Elemente des Typs *elemtype* hintereinander gelegt werden, dann legt extent(*elemtype*) den relativen Abstand der nachfolgenden Elemente fest.

Problem: Angenommen wir haben

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

und diese Matrix ist spaltenweise im Speicher, d.h. in der Sequenz 1, 4, 2, 5, 3, 6 mit $incRow = 1$ und $incCol = 2$. Dann lässt sich der Typ für eine Zeile mit *MPI_Type_vector*(1, 1, 2, *MPI_DOUBLE*, &*row_type*) erzeugen. Dann gilt für $T = row\_type =$ $\{(0, \text{MPI\_DOUBLE}), (16, \text{MPI\_DOUBLE}), (32, \text{MPI\_DOUBLE})\}$: $\text{lb}(T) = 0$, $\text{ub}(T) = 40$ und entsprechend $\text{extent}(T) = \text{ub}(T) - \text{lb}(T) = 40$.

Aus solchen Zeilentypen können wir nicht den korrekten Typ für die Matrix konstruieren, da der korrekte relative Offset bei aufeinanderfolgenden Zeilen 8 beträgt und nicht 40.

Datentyp für die Zeile einer Matrix

hpc/mpi/matrix.hpp

```
template<typename T, template<typename> class Matrix,
   Require<Ge<Matrix<T>>> = true>
MPI_Datatype get_row_type(const Matrix<T>& A) {
   MPI_Datatype rowtype;
   MPI_Type_vector(
      /* count = */ A.numCols(),
      /* blocklength = */ 1,
      /* stride = */ A.incCol(),
      /* element type = */ get_type(A(0, 0)),
      /* newly created type = */ &rowtype);

   /* in case of row-major we are finished */
   if (A.incRow() == A.numCols()) {
      MPI_Type_commit(&rowtype);
      return rowtype;
   }

   /* the extent of the MPI data type does not match
      the offset of subsequent rows -- this is a problem
      whenever we want to handle more than one row;
      to fix this we need to use the resize function
      which allows us to adapt the extent to A.incRow() */
   MPI_Datatype resized_rowtype;
   MPI_Type_create_resized(rowtype, 0, /* lb remains unchanged */
      A.incRow() * sizeof(T), &resized_rowtype);
   MPI_Type_commit(&resized_rowtype);
   MPI_Type_free(&rowtype);
   return resized_rowtype;
}
```

Datentyp für eine Matrix

hpc/mpi/matrix.hpp

```
template<typename T, template<typename> class Matrix,
   Require<Ge<Matrix<T>>> = true>
MPI_Datatype get_type(const Matrix<T>& A) {
   MPI_Datatype datatype;
   if (A.incCol() == 1) {
      MPI_Type_vector(
         /* count = */ A.numRows(),
         /* blocklength = */ A.numCols(),
         /* stride = */ A.incRow(),
         /* element type = */ get_type(A(0, 0)),
         /* newly created type = */ &datatype);
   } else {
      /* vector of row vectors */
      MPI_Datatype rowtype = get_row_type(A);
      MPI_Type_contiguous(A.numRows(), rowtype, &datatype);
      MPI_Type_free(&rowtype);
   }
   MPI_Type_commit(&datatype);
   return datatype;
}
```