- Bei inhomogenen Rechnerleistungen oder bei einer inhomogenen Stückelung in Einzelaufgaben kann es sinnvoll sein, die Last dynamisch zu verteilen.
- In diesem Falle übernimmt ein Prozess die Koordination, indem er Einzelaufträge vergibt, die Ergebnisse aufsammelt und – sofern noch mehr zu tun ist – weitere Aufträge verschickt.
- Die anderen Prozesse arbeiten alle als Sklaven, die Aufträge entgegennehmen, verarbeiten und das Ergebnis zurücksenden.
- Dies wird aus Gründen der Einfachheit an einem Beispiel der Matrix-Vektor-Multiplikation demonstriert, wobei diese Technik in diesem konkreten Beispiel wegen des Kopieraufwands nichts bringt.

Angenommen, die Matrix habe m Zeilen und uns stehen n Sklaven zur Verfügung. Der Einfachheit halber wird $m > n$ angenommen. Dann sehen die Rollen wie folgt aus:

- Master:
  - Verteile n und den Vektor an alle n Sklaven.
  - Versende jedem der n Sklaven eine Zeile der Matrix.
  - Insgesamt $m - n$ Mal: Empfange von irgendeinem Sklaven einen Wert des Resultatsvektors und schicke in Antwort eine weitere Zeile der Matrix.
  - Insgesamt n Mal: Empfange von irgendeinem Sklaven einen Wert des Resultatsvektors und schicke in Antwort ein Endesignal.
- Sklave:
  - Empfange n und den Vektor.
  - Für jede erhaltene Matrixzeile wird das entsprechende Skalarprodukt berechnet und verschickt.
  - Wenn das Endesignal ankommt, wird die Arbeit beendet.

Seien $n = 2$ und $m = 4$. Dann kann das so in CSP übertragen werden:

$$
\begin{aligned}
P &= \mathit{Master} \;||\; (\mathit{Slave} \;|||\; \mathit{Slave}) \\
\mathit{Master} &= \mathit{broadcast\_parameters} \rightarrow \mathit{broadcast\_parameters} \rightarrow \\
&\quad \mathit{exchange\_row} \rightarrow \mathit{exchange\_row} \rightarrow \\
&\quad \mathit{exchange\_value} \rightarrow \mathit{exchange\_row} \rightarrow \\
&\quad \mathit{exchange\_value} \rightarrow \mathit{exchange\_row} \rightarrow \\
&\quad \mathit{exchange\_value} \rightarrow \mathit{finish} \rightarrow \\
&\quad \mathit{exchange\_value} \rightarrow \mathit{finish} \rightarrow \\
&\quad \mathit{SKIP}_{\alpha \mathit{Master}} \\
\mathit{Slave} &= \mathit{broadcast\_parameters} \rightarrow \mathit{WorkingSlave} \\
\mathit{WorkingSlave} &= \mathit{exchange\_row} \rightarrow \mathit{exchange\_value} \rightarrow \mathit{WorkingSlave} \;| \\
&\quad \mathit{finish} \rightarrow \mathit{SKIP}_{\alpha \mathit{Slave}}
\end{aligned}
$$

MPI und CSP kommen sich in der Ausdrucksform hier sehr nahe:

- Die Datenübertragung erfolgt im einfachsten Falle synchron. Die *exchange_row*- und *exchange_value*-Ereignisse entsprechen jeweils einer Paarung von *MPI_Send* und *MPI_Recv*, die ebenfalls synchron erfolgen sollten.
- Bei *MPI_Send* und *MPI_Recv* wird zusätzlich noch eine Markierung in Form eines ganzzahligen Werts mit übertragen, der die Art der Nachricht charakterisiert (*tag value*). Dieser Wert kann verwendet werden, um auf der Seite des Sklaven das Empfangen einer weiteren Zeile von dem Empfangen des Endesignals unterscheiden zu können. Im Beispiel werden hier die Werte *NEXT_ROW* und *FINISH* verwendet.

mpi-gemv.cpp

```
int main(int argc, char** argv) {
   MPI_Init(&argc, &argv);

   int rank; MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   int nofslaves; MPI_Comm_size(MPI_COMM_WORLD, &nofslaves);
   --nofslaves; assert(nofslaves > 0);

   if (rank == 0) {
      int n; double** A; double* x;
      if (!read_parameters(n, A, x)) {
         cerr << "Invalid input!" << endl;
         MPI_Abort(MPI_COMM_WORLD, 1);
      }
      double* y = new double[n];
      gemv_master(n, A, x, y, nofslaves);
      for (int i = 0; i < n; ++i) {
         cout << " " << y[i] << endl;
      }
   } else {
      gemv_slave();
   }

   MPI_Finalize();
}
```

Rolle eines Sklaven

mpi-gemv.cpp

```
static void gemv_slave() {
   int n;
   MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
   double* x = new double[n];
   MPI_Bcast(x, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   double* row = new double[n];
   // receive tasks and process them
   for(;;) {
      // receive next task
      MPI_Status status;
      MPI_Recv(row, n, MPI_DOUBLE, 0, MPI_ANY_TAG,
         MPI_COMM_WORLD, &status);
      if (status.MPI_TAG == FINISH) break;
      // process it
      double result = 0;
      for (int i = 0; i < n; ++i) {
         result += row[i] * x[i];
      }
      // send result back to master
      MPI_Send(&result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
   }
   // release allocated memory
   delete[] x; delete[] row;
}
```

mpi-gemv.cpp

```
int n;
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
double* x = new double[n];
MPI_Bcast(x, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

- Zu Beginn werden die Größe des Vektors und der Vektor selbst übermittelt.
- Da alle Sklaven den gleichen Vektor (mit unterschiedlichen Zeilen der Matrix) multiplizieren, kann der Vektor ebenfalls gleich zu Beginn mit *Bcast* an alle verteilt werden.

mpi-gemv.cpp

```
      MPI_Status status;
      MPI_Recv(row, n, MPI_DOUBLE, 0, MPI_ANY_TAG,
         MPI_COMM_WORLD, &status);
      if (status.MPI_TAG == FINISH) break;
```

- Mit *MPI_Recv* wird hier aus der globalen Gruppe eine Nachricht empfangen.
- Die Parameter: Zeiger auf den Datenpuffer, die Zahl der Elemente, der Element-Datentyp, der sendende Prozess, die gewünschte Art der Nachricht (*MPI_ANY_TAG* akzeptiert alles), die Gruppe und der Status, über den Nachrichtenart ermittelt werden kann.
- Nachrichtenarten gibt es hier zwei: *NEXT_ROW* für den nächsten Auftrag und *FINISH*, wenn es keine weiteren Aufträge mehr gibt.

mpi-gemv.cpp

```
MPI_Send(&result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
```

- *MPI_Send* versendet eine individuelle Nachricht synchron, d.h. diese Methode kehrt erst dann zurück, wenn der Empfänger die Nachricht erhalten hat.
- Die Parameter: Zeiger auf den Datenpuffer, die Zahl der Elemente (hier 1), der Element-Datentyp, der Empfänger-Prozess (hier 0) und die Art der Nachricht (0, spielt hier keine Rolle).

mpi-gemv.cpp

```
static void
gemv_master(int n, double** A, double *x, double* y, int nofslaves) {
   // broadcast parameters that are required by all slaves
   MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
   MPI_Bcast(x, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

   // send out initial tasks for all slaves
   int* tasks = new int[nofslaves];
   // ...

   // collect results and send out remaining tasks
   // ...

   // release allocated memory
   delete[] tasks;
}
```

- Zu Beginn werden die beiden Parameter n und x, die für alle Sklaven gleich sind, mit *Bcast* verteilt.
- Danach erhält jeder der Sklaven einen ersten Auftrag.
- Anschließend werden Ergebnisse eingesammelt und – sofern noch etwas zu tun übrig bleibt – die Anschlußaufträge verteilt.

mpi-gemv.cpp

```
// send out initial tasks for all slaves
   // remember the task for each of the slaves
int* tasks = new int[nofslaves];
int next_task = 0;
for (int slave = 1; slave <= nofslaves; ++slave) {
   if (next_task < n) {
      int row = next_task++; // pick next remaining task
      MPI_Send(A[row], n, MPI_DOUBLE, slave, NEXT_ROW,
         MPI_COMM_WORLD);
      // remember which task was sent out to whom
      tasks[slave-1] = row;
   } else {
      // there is no work left for this slave
      MPI_Send(0, 0, MPI_DOUBLE, slave, FINISH, MPI_COMM_WORLD);
   }
}
```

- Die Sklaven erhalten zu Beginn jeweils eine Zeile der Matrix A, die sie dann mit x multiplizieren können.

mpi-gemv.cpp

```
// collect results and send out remaining tasks
int done = 0;
while (done < n) {
   // receive result of a completed task
   double value = 0; // initialize it to get rid of warning
   MPI_Status status;
   MPI_Recv(&value, 1, MPI_DOUBLE,
      MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
   int slave = status.MPI_SOURCE;
   int row = tasks[slave-1];
   y[row] = value;
   ++done;
   // send out next task, if there is one left
   if (next_task < n) {
      row = next_task++;
      MPI_Send(A[row], n, MPI_DOUBLE, slave, NEXT_ROW,
         MPI_COMM_WORLD);
      tasks[slave-1] = row;
   } else {
      // send notification that there is no more work to be done
      MPI_Send(0, 0, MPI_DOUBLE, slave, FINISH, MPI_COMM_WORLD);
   }
}
```

Beachtenswert ist hier, dass bei der Übertragung eines Arrays die Länge dynamisch gewählt weden kann:

- Beim Versenden der nächsten Matrixzeile werden nebem dem *tag value* noch *n* Werte übermittelt:

```
MPI_Send(A[row], n, MPI_DOUBLE, slave, NEXT_ROW, MPI_COMM_WORLD);
```

- Beim Übermitteln des Endesignals wird als Array-Länge die 0 angegeben, d.h. es wird nur *FINISH* übertragen:

```
MPI_Send(0, 0, MPI_DOUBLE, slave, FINISH, MPI_COMM_WORLD);
```

Die Kombination von ganzzahligen Paketarten (hier *NEXT_ROW* oder *FINISH*) mit dynamischen Arrays vermeidet die Aufsplittung solcher Pakete in getrennte Header- und Datenpakete, die die Latenzzeiten erhöhen würden.

mpi-gemv.cpp

```
      MPI_Status status;
      MPI_Recv(&value, 1, MPI_DOUBLE,
         MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      int slave = status.MPI_SOURCE;
```

- Mit *MPI_ANY_SOURCE* wird angegeben, dass ein beliebiger Sender akzeptiert wird.
- Hier ist die Identifikation des Sklaven wichtig, damit das Ergebnis korrekt in y eingetragen werden kann. Dies erfolgt hier durch das Auslesen von *status.MPI_SOURCE*.

```
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.

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.

```
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.

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

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

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 $P_1 Buffer$:

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