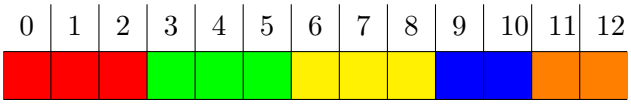
# Blatt 07 Antworten

**Aufgabe 1.**

Wie in Blatt05 ausgearbeitet kann die Formel $c(N, n_{procs}, rank) = \left\lfloor \frac{N}{n_{procs}} \right\rfloor + (rank < (N \bmod n_{procs}))$ verwendet werden um zu bestimmen wieviele element der n thread kriegt. Für $N = 13$ und $n_{procs} = 5$ gibt das $c(13, 5, \{0, 1, 2\}) = 3$ und $c(13, 5, \{4, 5\}) =$




**Aufgabe 2.** 1. Die Richtung ist implizit durch die Zeitachse gegeben: Kommunikation startet in der Vergangenheit, i.e. auf der linken Seite und geht in die Zukunft, i.e. nach rechts. Das heißt der senden Prozess ist der, bei dem die Verbindung links anfängt. Das entspricht meinen Erwartungen, allerdings überrascht es mich das dies zuverlässig funktioniert und nicht unterschiedliche Zeitmessungen häufig Probleme machen.

2.

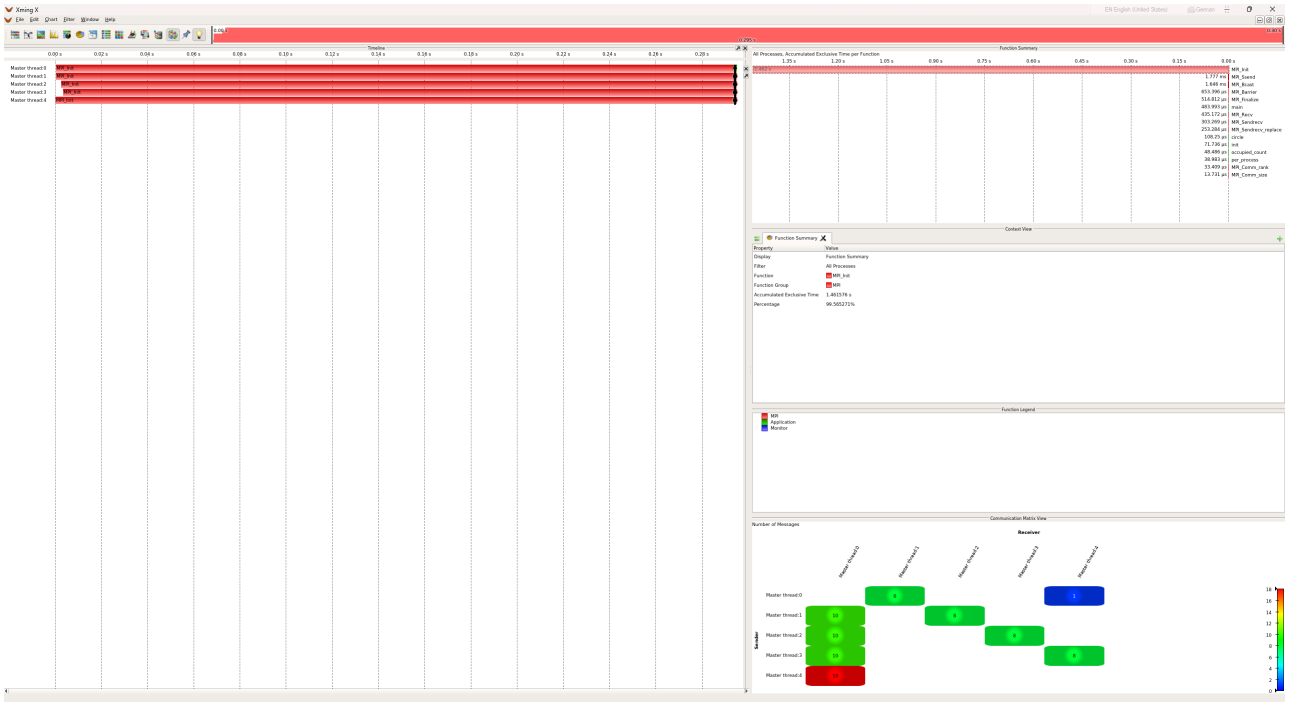


Abbildung 1: Gesamtansicht von Vampir mit der Communication Matrix View unten rechts

3. Wie im obigen Screenshot zu sehen dominiert die `MPI_Init`-Phase die Laufzeit. Um den rest klarer zu sehen zoom wir nochmal ran und markieren die 4 Phasen, `MPI_Init` (welche sehr viel größer ist als hier sichtbar), `BEFORE` (wo das array vorher einmal

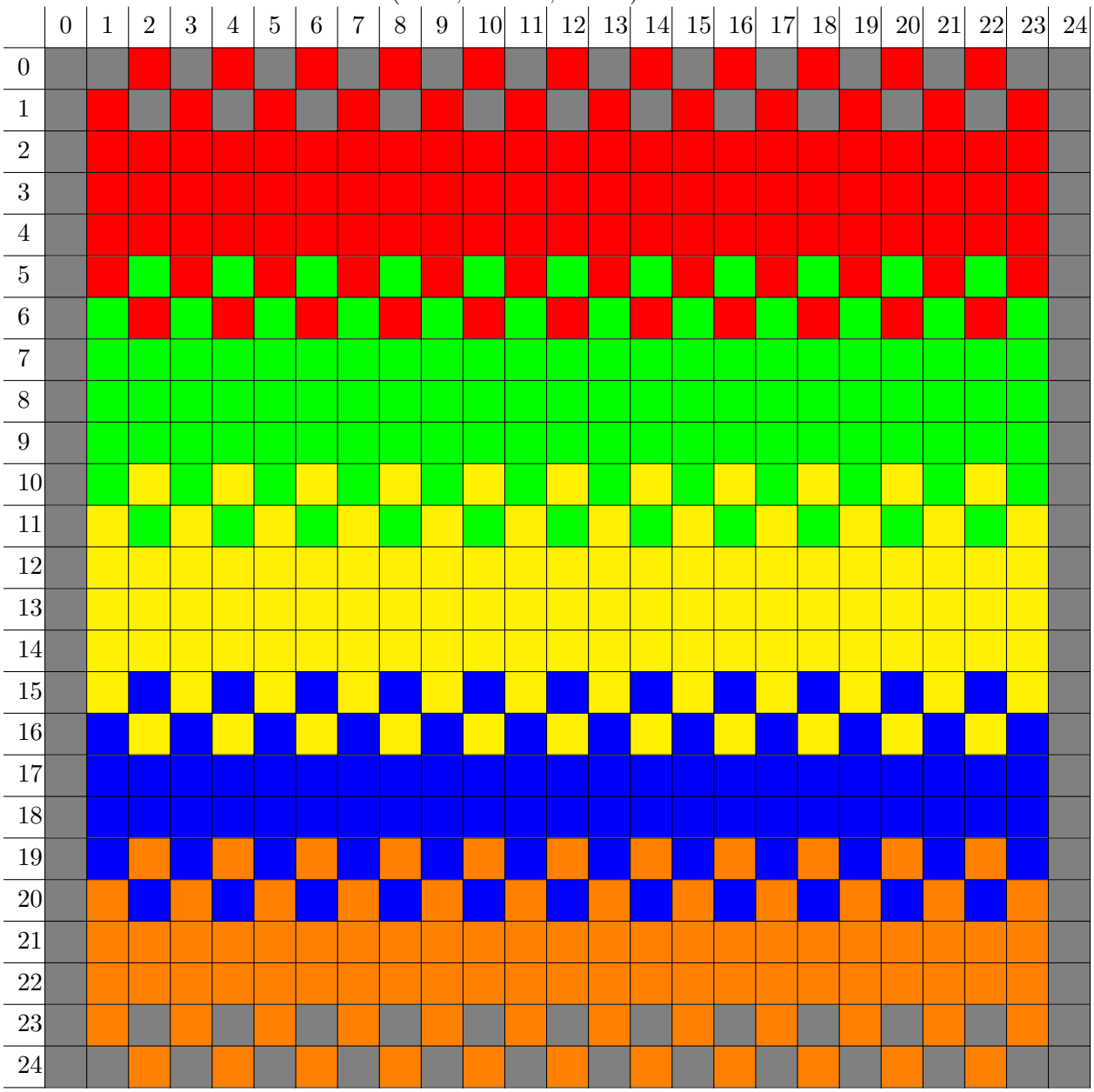
ausgegeben wird), `circle` (wo die Haupt-ärbeit"passiert) und `AFTER` (wo das array nachher nocheinmal ausgegeben wird, sowie `MPI_Finalize` aufgerufen wird).


4. `MPI_Init` hat über alle Prozesse akkumuliert 1.462 sekunden gebraucht, viel hiervon ist jedoch natürlich parallel abgelaufen sodass die tatsächliche Echtzeit nur 0.295s ist.

**Aufgabe 3.** 1. Ähnlich wie bisher können wir $c(N,K,p) = \lfloor \frac{N}{K} \rfloor + (p < (N \mod K))$ als Basis verwenden. Dabei ist zu beachten das wir $N = i * 8 + 7$ verwenden um nur die inneren Zeilen zu messen. Die Anzahl der gespeicherten Zeilen ist $c(N,K,p) + 2$. Damit kriegen wir die folgenden Formeln für die Speicher- start- und end- zeile des p-ten Prozesses: $s(N,K,p) = \lfloor \frac{N}{K} \rfloor \times p + \min(p, N \mod K)$ und $s(N,K,p) = \lfloor \frac{N}{K} \rfloor \times (p+1) + \min(p+1, N \mod K) + 2$, beide inklusive.

2. In der Grafik müssen alle Zeilen mit einem Schachbrettmuster zwischen diesen Zwei Prozessen kommuniziert werden, am einfachsten über ein SendRecv. Rot und Orange welche jeweils auf einer Seite mit dem grauen Rand austauschen sollen, würden hierzu mit dem Null Prozess kommunizieren. Dies wird den Code einfacher machen da weniger spezial Cgitasing für die Ränder notwendig ist.

$(i = 2, N = 23, K = 5)$


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

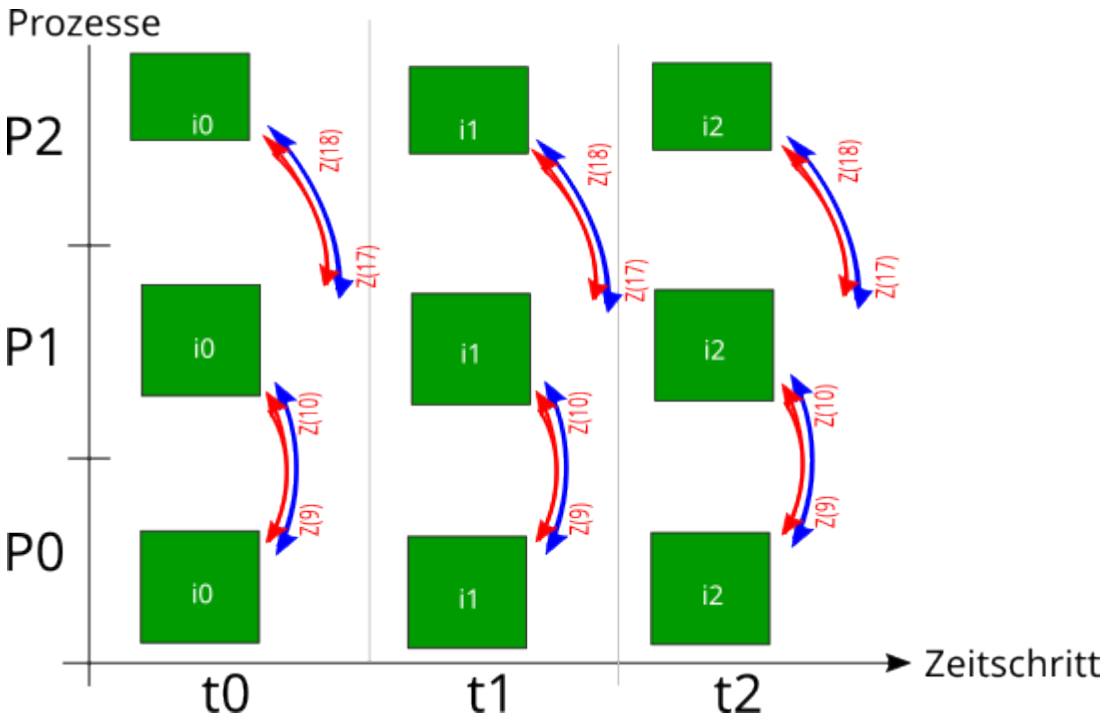



Abbildung 2: Diagramm für Abbruchbedingung iterations bei i=2, K=3. Rot-Blaue Pfeile stellen `MPI_Sendrecv` Operationen dar. Der Text näher bei dem Prozess gibt an was der Prozess sendet.

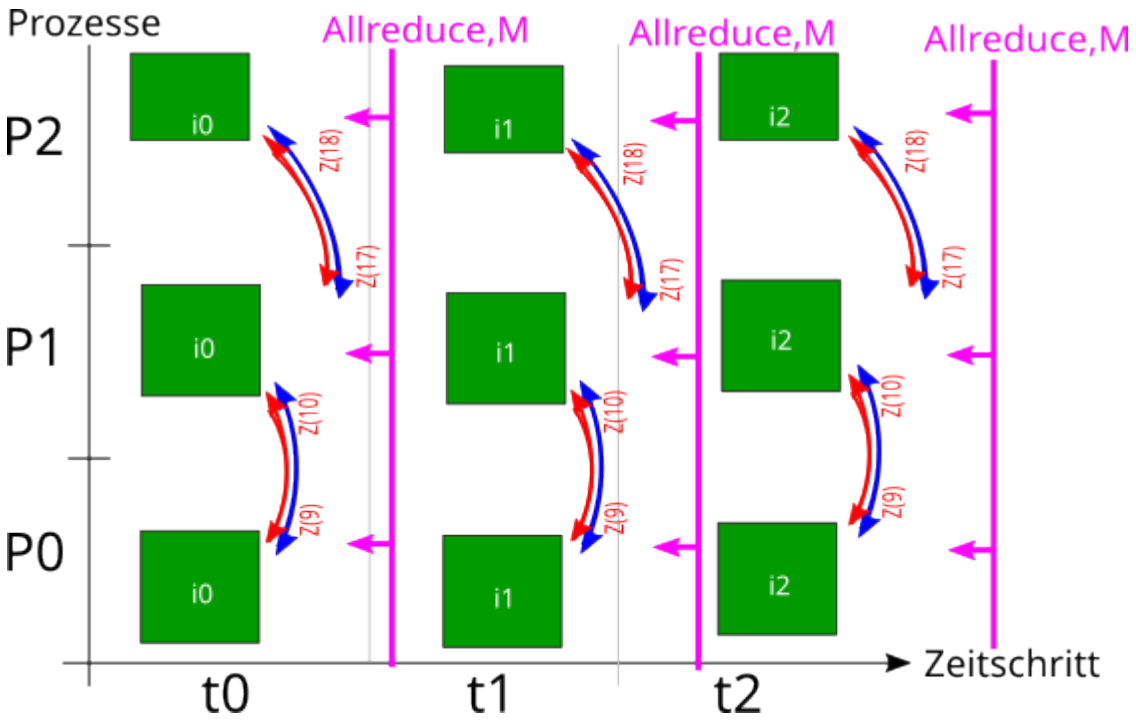



Abbildung 3: Diagramm für Abbruchbedingung precision bei i=2, K=3. Rot-Blaue Pfeile stellen `MPI_Sendrecv` Operationen dar. Der Text näher bei dem Prozess gibt an was der Prozess sendet. Die pinken Barrieren sind `MPI_Allreduce` Operationen, welche dafür sorgen dass keine weitere Synchronisation notwendig ist, um zu entscheiden, ob weiter iteriert werden soll.

3. Knifflige stellen: Wenn nicht `MPI_Sendrecv` verwendet wird, muss sehr darauf geachtet werden, dass keine deadlocks passieren können, wo zwei threads gleichzeitig darauf warten, dass der jeweilige andere `MPI_Recv`/`MPI_Send` aufruft. Bei Abbruch nach Präzision muss auch darauf geachtet werden, dass alle threads wissen, das abgebrochen werden soll. Dies kann entweder nach einem `MPI_Reduce` mithilfe von `MPI_Bcast` passieren oder mit `MPI_Allreduce`. Bei der Implementierung muss auch darauf geachtet werden, das die zwei Randprozesse nicht aus versehen mit invaliden Prozesshandles kommunizieren.