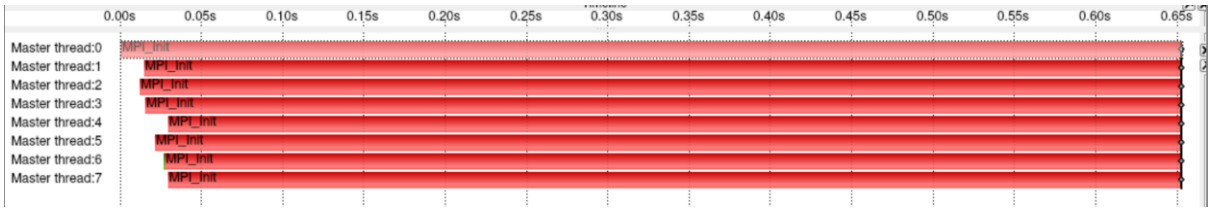
SPP Praktikum 2

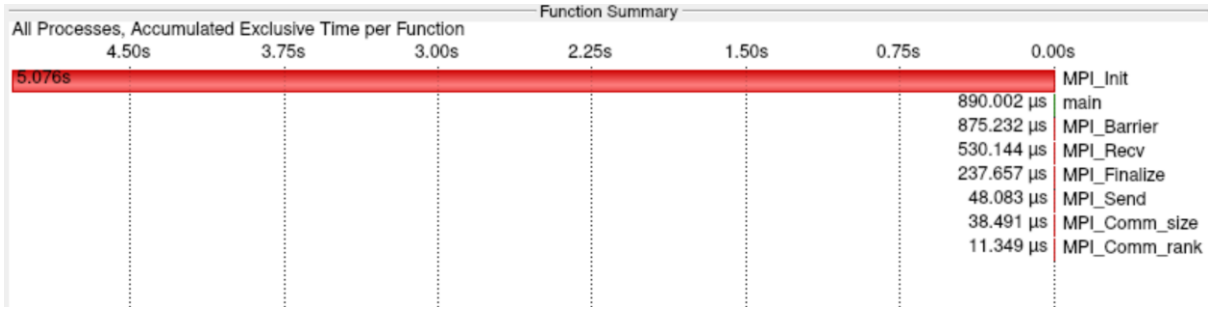
Group 151

Task 2 (Beschreibung wie Teilergebnisse zusammengeführt werden)



Hier oben im Screenshot sieht man alle parallelen Prozesse und in der x-Achse erkennt man, dass die gesamte Laufzeit 0.65s betrug. Dabei sieht man, dass die einzelnen Threads zu unterschiedlichen Zeiten begonnen haben(MPI_Init). Hier beginnt zuerst der thread: 0 in der Sekunde 0, danach thread:2 nach ca.0.015s als nächstes usw.

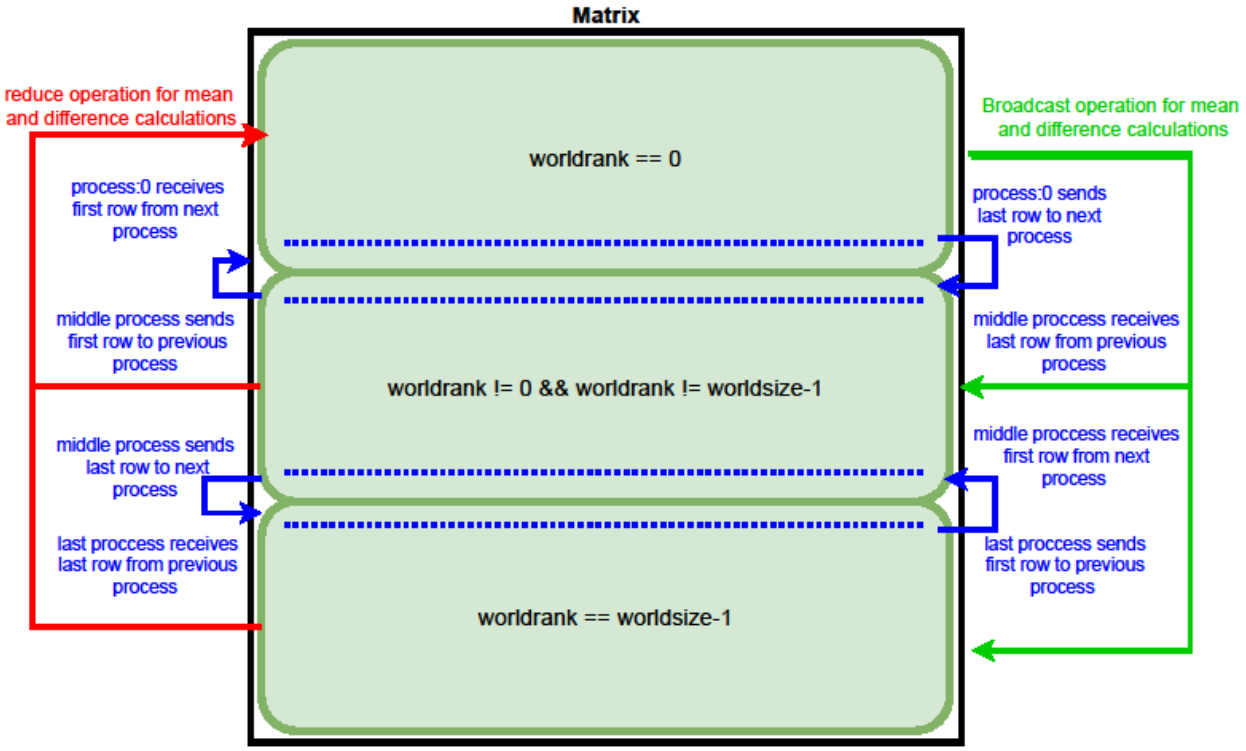
Jedoch werden alle Prozesse gleichzeitig nach 0.65s wegen MPI_Finalize() beendet. Dh. Nach dieser Zeit wurden alle Teilergebnisse zusammengeführt.



Hier im zweiten Screenshot sieht man die zeitliche Unterteilung der einzelnen MPI Methoden des thread:0. Alle diese Methoden befinden sich innerhalb der main-Methode und deren einzelnen Laufzeiten werden hier dargestellt. Hier erkennt man, dass die MPI_Init Methode mit sehr großem Abstand am meisten Zeit beansprucht.

Task 3

a) Kommunikationsschema



c)

```
if (world_rank == 0)    //Sends their last row to next process, receives last row from next process
{
  //double next_row[N];
  // Sending last row to the 'next' process (world_rank + 1)
  //std::cout << "sending row(master)" << std::endl;

  MPI_Send(&u[subRows-1][0], N, MPI_DOUBLE, world_rank + 1, 0, MPI_COMM_WORLD);
  //std::cout << "sent row(master)" << std::endl;
  // Receiving first row from the 'next' process
  //std::cout << "receiving row(master)" << std::endl;

  MPI_Recv(&next_row[0], N, MPI_DOUBLE, world_rank+1, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  //std::cout << "received row(master) " << next_row[0] << " " << next_row[1] << std::endl;
```

Ein Beispiel aus unserer Aufgabe, bei der blockierende MPI-Kommunikation die Performance beeinflusst hat, ist die Berechnung der „inneren“ Werte der Heated Plate. Bei der blockierenden Kommunikation kann die Berechnung erst beginnen, wenn der Prozess die Werte vom folgenden/vorhergehenden Teil der Matrix erhalten hat, während bei nicht-blockierender Kommunikation, die inneren Werte bereits berechnet werden und zeitgleich die Werte für den Übergang zur nächsten/vorhergehenden Teilmatrix empfangen werden können.

Zudem erlaubt die Verwendung von nicht-blockierender Kommunikation die gewünschte „Plattengröße“ zu erhöhen ohne signifikante Einbußen in der Performance in Kauf zu nehmen. Die blockierende Kommunikation würde wiederum bei größeren „Platten“ zu einem erheblich Bottleneck werden, welcher die Skalierbarkeit und Performance einschränkt.

Task 4

- Hierbei wird ein Bild generiert, das aus einer festen Anzahl an Pixel besteht. Dabei werden alle Pixel (wenn möglich gleichmäßig) zeilenweise auf alle Prozesse aufgeteilt. Hierfür werden die Anzahl der Zeilen durch die Anzahl der Prozesse geteilt (‚own_height'). Das ist vergleichbar mit dem static scheduling von OpenMP.

- In der Implementation 2 werden die Pixel auch an alle Prozesse aufgeteilt, aber nicht gleichmäßig und nicht vor der Berechnung. Hierbei werden die Prozesse in einen master und viele worker Prozesse unterteilt. Der master verteilt dynamisch die Pixel Zeilen einzeln an die worker prozesse. Das ist vergleichbar mit dem dynamic scheduling von OpenMP.

- In Mandelbrot 1 erstellt jeder Prozess sein eigenes „Teilbild" bestehend aus mehreren Zeilen. Diese wird basierend auf ihren Zeilenpositionen an die richtigen Stellen des Gesamtbildes geschrieben. In Mandelbrot 2 hingegen schreiben alle Prozessen gleichzeitig ihre fertigerstellten Zeilen direkt ins Bild. Wegen des parallelen Schreibvorgang ist es schneller.

- In Mandelbrot 1 findet keine Kommunikation statt, da zu Beginn jedem Prozess die „Teilbilder" zugeordnet wurden. In Mandelbrot 2 werden mittels MPI_Recv and MPI_Send sehr viel zwischen den master und den worker kommuniziert. Jeder Zeilenindex wird an einen worker gesendet, welcher diese Zeile berechnet und dem master den Abschluss der Berechnung mitteilt, damit dieser den nächsten Zeilenindex versenden kann.

	Mandelbrot 1	Mandelbrot 2
Vorteile	Einfach zu implementieren	Flexiblere workload Verteilung
Nachteile	Keine flexible workload Verteilung	Komplexe und fehleranfällige Kommunikation

- Wenn jede Pixelzeile gleich aufwendig zu berechnen ist (workload), dann ist Mandelbrot 1 die vorzuziehende Wahl, weil dieser die Aufgaben gleichmäßig an die Prozessen verteilt, mit wenig overhead.
  Bei unterschiedlichen Zeilenaufwand ist Mandelbrot 2 aufgrund ihrer dynamischen Arbeitsverteilung vorzuziehen.