Forschungsbericht

Visualisierung HPC IO

Erstellen eines Grafana Plugins
zur Visualisierung von High Performance Computing

Simon Rosenberger

Prof. Dr.-Ing. Rainer Keller

Philipp Köster

Bearbeitungszeitraum
Oktober 2022 bis Februar 2023

Inhaltsverzeichnis

1 Aufgabenstellung/Zielsetzung

Die Aufgabe des Forschungsprojektes besteht in einer Visualisierung mithilfe von Grafana das Live-Tracing von Daten zu ermöglichen und mögliche Bottlenecks zu identifizieren und damit eine Optimierung des I/O-Durchsatzes in High Performance Computing Anwendungen zu ermöglichen.
Das Ziel des ersten Semesters des Forschungsprojektes ist das Erstellen einer Heatmap.
Aus dieser Vorgabe entstand die Idee der ThreadMap. Die ThreadMap ist ein selbst geschriebenes Grafana Plugin, welches Prozesse und Threads (deren ID), deren Auslastung und das Filesystem in welches diese geschrieben haben visualisiert.

2 Einarbeitung

Die Einarbeitung lässt sich in drei Bereiche unterteilen. Diesen geht die Aufsetzung der Entwicklungsumgebung voraus. Da die Umgebung auf Basis eines Windowsbetriebssystems aufgesetzt wird, muss zuerst ein Windows-Subsystem für Linux (WLS2) mit der Ubuntu Distribution, aufgesetzt werden.
Um Grafana und InfluxDB im Localhost verwenden zu können werden diese mithilfe von Docker als Container ausgeführt.
Zuerst erfolgte eine Einarbeitung in InfluxDB, welches als Datenbank-Managementsystem verwendet wird [1]. Da die „Flux“ Sprache für alle Abfragen, um auf die verwendete Timeseries-Datenbank zugreifen zu können verwendet wird, ist eine Einarbeitung in die Syntax dieser Sprache nötig. Hierbei kam vor Allem die InfluxDB-Cloud [2] zur Anwendung. In dieser sind alle möglichen Funktionen und deren Syntax zu finden.

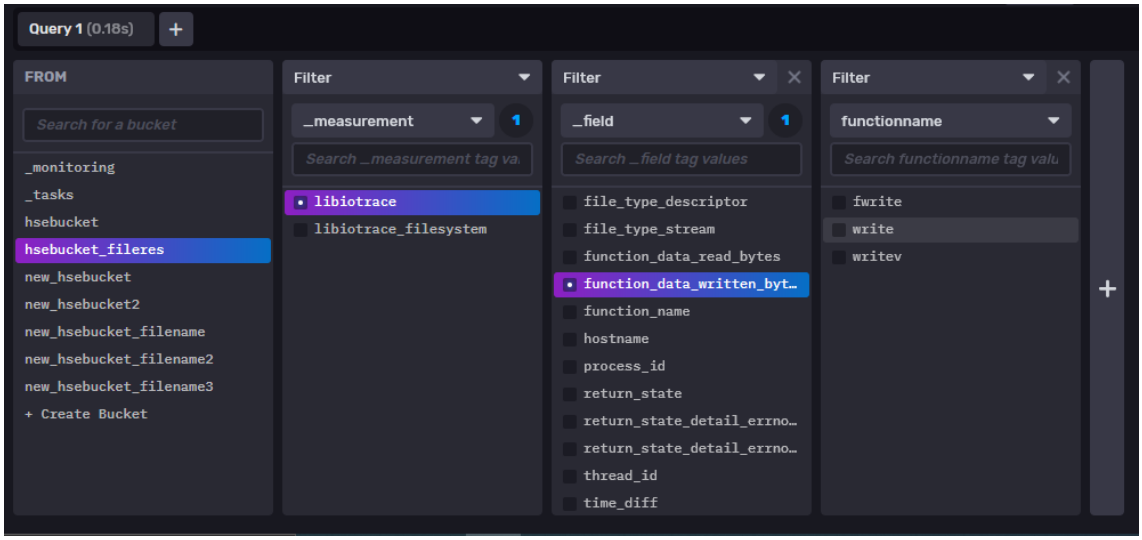

Abbildung 2-1: Anwahl im Influx DataExplorer

Die Datenauswahl kann auch über den DataExplorer (Localhost-Anwendung über Docker) über eine einfache Anwahl konfiguriert werden, für exaktere Dateneingrenzungen ist das manuelle Schreiben der Flux-Query sinnvoller.
Im zweiten Schritt erfolgte die Einarbeitung in die Entwicklungsumgebung von Grafana. Anfangs beschränkte es sich auf das Testen von Grafana bereitgestellten Plugins [3] mit einem kleinen lokal ablaufenden Test (siehe Kapitel 3) und einer festen Flux-Query, mit dem Ziel die Funktionalität der Plugins und deren Grenzen zu verstehen. Da dies mit einem gleichbleibenden Test nicht möglich

ist, erfolgte daraufhin eine Anpassung der im Plugin enthaltenen Flux-Query, sodass mit anderen Daten bzw. Tests experimentiert werden konnte.
Aus den daraus gewonnenen Erkenntnissen wurde klar, dass von bereits vorhandenen Plugins nicht für die Erfüllung der Aufgabenstellung infrage kommen. Daher fiel die Entscheidung ein eigenes Plugin zu erstellen. Infolgedessen war eine Einarbeitung in das Erstellen, Verwalten und Programmieren dieser Plugins notwendig [4].
Um Daten für die Grafana Visualisierungen verwenden zu können, werden die oben bereits erwähnten Tests benötigt. Da das Arbeiten mit den Tests einen nicht zu vernachlässigbaren Teil der Arbeitszeit beansprucht wird im folgenden Kapitel genauer auf diese eingegangen.
Es ist anzumerken, dass die Einarbeitung nicht wie hier beschrieben linear ablief, sondern je nach aktuellem Bedarf alle drei Themengebiete zeitgleich umspannte.

3 Anpassen Testumgebung & Tests

Damit Daten zur Visualisierung vorliegen bzw. diese in Live-Tests generiert werden, ist es nötig Tests auszuführen. Die Daten in die InfluxDB schreiben.

```
rm -f mpi_io_file_test1* && rm -f mpi_file_io.txt && mpirun -np 1 -x IOTRACE_LOG_NAME=mpi_io_file_test1 -x IOTRACE_DATABASE_IP=127.0.0.1 -x IOTRACE_DATABASE_PORT=8086 -x IOTRACE_INFLUX_ORGANIZATION=hse -x IOTRACE_INFLUX_BUCKET=hsebucket -x IOTRACE_INFLUX_TOKEN=OXBWllU1poZotgyBlLlo2XQ_u4AYGYKQmdxvJJeotKRyvdn5mwjEhCXyOjyldpMmNt_9YY4k3CKf5Eh1bN0Ng== -x LD_PRELOAD=../src/libiotrace_shared.so mpi_file_io 1000
```

Listing 1: Aufruf MPI File IO Test

Der erste und einfachste Test bestand aus einem einfachen Aufruf des MPI File IO Tests. Mit diesem Aufruf werden in die Datenbank „hsebucket“ in der InfluxDB (Localhost, Port 8060) 1000-mal der gleiche Befehl „mpi_file_write“ geschrieben. Da dieser Test keine Varianz in der Auslastung besitzt, eignen sich die von ihm erstellten Daten nicht für Tests mit einer Heatmap und ähnlichem.

Zur Lösung dieses Problems wird ein Test aus den Tutorials des Simulationsprogrammes OpenFOAM verwendet. In diesem Fall der openFOAM_motorBike Test, der die Aerodynamik eines Motorrads mit Fahrer simuliert.
Die Konfiguration welche Daten der Test schreiben soll, sind mit CMake konfiguriert. Auf die genaue Konfiguration wird in diesem Dokument nicht weiter eingegangen.
Um ein mehrfaches Aufrufen der Test zu vereinfachen, können diese auch über das File start.sh aufgerufen werden. Dieses File beinhaltet beide Testfälle die nach Belieben ein/auskommentiert werden können.

```
test_gcc_version="12.1"
test_mpi_version="4.1"
test_openfoam_dir="$HOME/test/Projects/OpenFOAM-10"
test_influx_dir="$HOME/test/testfsprj/influxdb/influxdb2-2.0.6-linux-amd64"
test_dir="$HOME/test/fsprj2/libiotrace/scripts/testscripts"
test_processes_per_worker=16

test_script="./openFOAM_motorBike.sh"
test_nodes=3
test_processes_per_worker=16
test_mem="90000mb"
test_time="00:30:00"
test_name="3_nodes_motorBike"

#test_script="./mpi_file_io.sh"
#test_nodes=3
#test_processes_per_worker=40
#test_mem="90000mb"
#test_time="00:10:00"
#test_name="3_nodes_mpi_file_io"

module purge
module load compiler/gnu/${test_gcc_version}
module load mpi/openmpi/${test_mpi_version}

# load and initialize openFOAM
. ${test_openfoam_dir}/etc/bashrc

module purge
module load compiler/gnu/${test_gcc_version}
module load mpi/openmpi/${test_mpi_version}

sbatch -p dev_multiple -N ${test_nodes} --mem=${test_mem} -t ${test_time} libiotracetest.sh
${test_script} ${test_name} ${test_influx_dir} ${test_dir} ${test_openfoam_dir}
${test_processes_per_worker}
```

Listing 2: start.sh

Dort können die zu verwendeten Software-Version und Spezifikationen der Tests eingestellt werden. Dies umfasst Eigenschaften, wie das Directory des Testskriptes, die Anzahl der Nodes auf denen der Test ausgeführt wird, wie viele Prozesse pro Nodes parallel ausgeführt werden können, der zur Verfügung stehende Arbeitsspeicher, die maximale Testdauer und den Namen des Testfiles.
Da Tests im oben definierten Umfang sehr viel Leistung zur Ausführung benötigen, ist die Ausführung nicht lokal erfolgt, sondern auf dem bwUniCluster. Auf diesem Cluster stehen dem Benutzer unterschiedliche Queues mit bis zu 128 Nodes und Testdauern von maximal 48h zur Verfügung (siehe [5]).
Um die Daten der Tests abzurufen gibt es zwei Möglichkeiten:
Nach erfolgreichem Abschluss des Testes wird im Test-Directory ein Backup mit den Daten des Testes erstellt. Dieses Backup kann nun in die lokale Kopie des HPC-IO Projektes kopiert werden und auf diesem Verzeichnis ein Docker-Volume im Influx-Container erstellen. Der Docker hat dadurch Zugriff auf das Verzeichnis mit den Backupdaten. Abschließend müssen diese in einem influxBucket wiederhergestellt werden. Die Daten können so vollständig ausgelesen werden. Das Arbeiten mit den Backupdaten bietet sich vor Allem während der Plugin Entwicklung an, da so nicht für jede Änderung ein neuer Live Test gestartet werden muss.
Für den Fall, dass ein Live Test benötigt wird, kann während der Ausführung des Testes auf dem bwUniCluster ein Port-Forwarding auf eine Instanz der InfluxDB erfolgen. So werden die aktuell auf dem UniCluster geschriebenen Daten live an die InfluxDB weitergegeben und können lokal im Grafana ausgewertet werden. Live Tests bieten den Vorteil, das Dashboard auf Performance und Verhalten beim Erhalt neuer Daten zu untersuchen und bilden den späteren Anwendungsfall am besten ab.
Das Arbeiten mit den Tests beinhaltet auch das Anpassen und die Fehlerbehebung in der Entwicklungsumgebung und den Test-Aufrufen. Speziell bei der Ausführung des OpenFOAM Tests nahm die Fehlersuche mit Aktivierter Filenameresolution viel Zeit in Anspruch. Nach aktuellem Stand ist der Test fehlerfrei Ausführbar und liefert als Backup vollständige Daten. Ein Livetest in dieser Konfiguration steht noch aus.

4 Umsetzung

4.1 Dashboard

Das aktuell verwendete Dashboard beinhaltet drei voneinander unabhängige Plugins Das wichtigste ist das selbstgeschriebene ThreadMap-Plugin. Dieses zeigt die im angewählten Zeitraum ausgeführten Prozesse und Threads an. Den Prozessen sind die zugehörigen Threads unterlagert zugeordnet. Es werden jeweils die Prozess- bzw. ThreadID angegeben, deren Auslastung anhand farblicher Kennzeichnung und das Filesystem, in dem diese ausgeführt wurden.
Um einen besseren Überblick über die Daten zu erhalten sind zusätzlich zwei Standard-Plugins von Grafana im Dashboard enthalten.

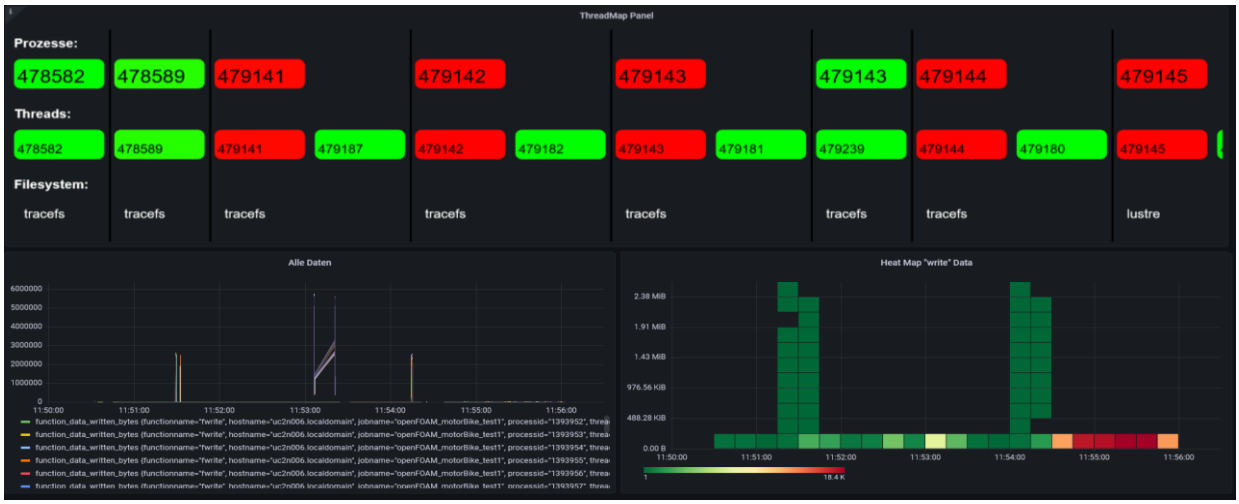

Abbildung 4-1: Dashboard ThreadMap & Performance

Auf der linken Seite ein Time Series Plugin zur Überprüfung und Darstellung des zeitlichen Verlaufs der Daten und auf der rechten Seite eine Heatmap die die Auslastung über einen zeitlichen Verlauf anzeigt. Das ThreadMap Plugin zeigt die Auslastung nur über den gesamten Zeitraum, so kann mithilfe der Heatmap der exakte Zeitpunkt eines Bottlenecks leichter gefunden werden.
Das Dashboard stellt in diesem Beispiel Daten des openFOAM_motorBike Testes dar.

Jedes Plugin innerhalb eines Dashboards besitzt eine eigene Flux-Query, um Daten zu erhalten. Bei Bedarf können auch mehrere Queries pro Plugin verwendet werden.

Flux 1: Get DATA

```
from(bucket: "new_hsebucket_filename3")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "libiotrace")
  |> filter(fn: (r) => r["_field"] == "function_data_written_bytes")
  |> aggregateWindow(every: v.windowPeriod, fn: sum, createEmpty: false)
  |> yield(name: "sum")
```

Flux 2: Get Filesystem

```
from(bucket: "new_hsebucket_filename3")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "libiotrace_filesystem")
  |> filter(fn: (r) => r["_field"] == "mount_type")
  |> aggregateWindow(every: v.windowPeriod, fn: last, createEmpty: false)
  |> yield(name: "last")
```

Listing 3: Flux-Queries ThreadMap

Um die in der in Grafana vorhanden Daten zu begrenzen, werden die Daten vor der Verwendung gefiltert. Die Daten werden hierbei aus einem Bucket der InfluxDB geladen. Der Zeitraum entspricht dem im Dashboard angewählten Zeitrahmen. In der erste Flux-Query werden alle Prozesse und Thread die im Test Bytes geschrieben haben und deren Auslastung ausgelesen. Die Auslastung wird pro Thread über den angewählten Zeitraum aufsummiert.
Die zweite Flux-Query dient zum Auslesen des Filesystems in welches geschrieben wurde.
Die Datenpunkte mit der Auslastung sind im Bereich 0 bis datalength (libiotrace) gespeichert und die Informationen zum verwendeten Filesystem ab datalength+1 (libiotrace_filesystem).

4.2 Plugin: ThreadMap

Kleiner Funktionen und Abfragen werden im Folgenden ignoriert und nur die wichtigsten Funktionen erklärt, die für das Verständnis des Plugins nötig sind.

4.2.1 Function Colour Steps:

Anhand der Datenpunkte wird die gleiche Anzahl an Farbpunkten gebildet. Die Anzahl Datenpunkte entspricht der Anzahl der ausgeführten Threads. Die Farbpunkte erhalten einen Farbverlauf von Grün über Gelb nach Rot. Grün steht hierbei für minimale und Rot für die maximale Auslastung.

```
function ColourSteps()
{
  let Colour = new Array
  const step = 255 / ((datalength-1)*0.5)
  Colour[0] = "#00ff00"

  for (let i = 1; i < (datalength); i++) {
    if(i < (datalength/2)) { //Grün #00FF00 bis Gelb #FFFF00
      if ((i*step) < 16) {
        Colour[i] = "#0" + (i*step).toString(16).substring(0,1) + "ff00"
      }
      else {
      Colour[i] = "#" + (i*step).toString(16).substring(0,2) + "ff00"
      }
    }
    else { //Gelb #FFFF00 bis Rot #FF0000
      if (((i-datalength/2)*step) >= 240) {
        Colour[i] = "#ff0" + (255-((i-datalength/2)*step))
          .toString(16).substring(0,1) + "00"
      }
      else {
      Colour[(i)] = "#ff" + (255-((i-datalength/2)*step))
          .toString(16).substring(0,2) + "00"
      }
    }
  }
  return(Colour)
}
```

Listing 4: Function Colour Step

Startend bei der Farbe Grün wird so lange der Rot-Anteil erhöht bis man die Farbe Gelb erreicht. Anschließend wird der Grün-Anteil reduziert bist man die Farbe Rot erhält.
Da in Typescript ein direktes Rechnen mit Hexadezimalenzahlen nicht möglich ist wurde in diesem Fall ein Workaround verwendet der diese Zahlen (00FF00 ➔ FFFF00 ➔ FF0000) in einen String umwandelt. Das Hexadezimale Format wird benötigt, um die entsprechende Farbe korrekt im CSS darstellen zu können (SVG fill akzeptiert Farben nur mit Hexadezimalwerten oder dem Farbnamen als string).
Der Return Wert der Funktion enthält ein Array mit einer Länge entsprechend der Anzahl der aus Influx erhaltenen Datenpunkte, welches einen linearen Farbverlauf in Hexadezimaler Darstellung beinhaltet.

4.2.2 Function Colour Assignment

Die Funktion ColourAssignment wird für die Zuweisung der Datenpunkte anhand ihrer Auslastung an die entsprechende Farbe benötigt. Die Zuweisung erfolgt für Prozesse und Threads.
Dem Array ThreadHeatValue werden zuerst die Werte des Arrays values zugewiesen (Das Array values der erhaltenen Daten beinhaltet das Array buffer, auf welches nicht direkt zugegriffen werden kann). Anschließend wird die aktuelle Position des Arrays ThreadHeatvalues (dieses beinhaltet nun das Sub-Array buffer) mit der Summe des Sub-Arrays überschrieben. Nach durchlaufen der For-Schleife beinhaltet das Array die Summe aller Bytes, die ein Thread während des ausgewählten Zeitraumes geschrieben hat.
Für die Auslastung der Prozesse wird dem Array ProcessHeatValue die zuvor gebildeten Auslastungen der Threads zugewiesen. Da ein Prozess mehrere Threads untergeordnet haben kann ist zusätzlich eine Abfrage eingebaut, ob einem Prozess bereits etwas zugewiesen wurde. Falls ja, wird die Auslastung aufaddiert. Übersprungene Werte, bleiben als „undefined“ im Array zurück. Dies ist notwendig, da die Zuordnung von Prozessen und Threads anhand der Array-Position erfolgt. Da durch das influxQuery (Listing 3) die Daten immer in Reihenfolge sortiert ausgegeben werden, reicht es den nachfolgenden Prozess auf Gleichheit zu überprüfen.

```
function ColourAssignment(Colour: any) {
    var ThreadHeatValue = new Array
    var ProcessHeatValue = new Array
    var ReturnColourThreads = new Array
    var ReturnColourProcess = new Array

    for (let i = 0; i < datalength; i++) {
      ThreadHeatValue[i] =data.series[i].fields[1].values
      ThreadHeatValue[i] = _.sum(ThreadHeatValue[i].buffer)
    }

    for (let iproc = 0; iproc < datalength; iproc++) {
      if ((iproc < (datalength - 1))) {
        if (!(data.series[iproc].fields[1].labels?.processid ==
             data.series[iproc+1].fields[1].labels?.processid)) {
          ProcessHeatValue[iproc] = ThreadHeatValue[iproc]
        }
        else{
          ProcessHeatValue[iproc] = ThreadHeatValue[iproc] +
          ThreadHeatValue[iproc+1]
          iproc++
        }
      }
      else{
        ProcessHeatValue[iproc] = ThreadHeatValue[iproc]
      }
    }
    ReturnColourThreads = Assignment(ThreadHeatValue)
    ReturnColourProcess = Assignment(ProcessHeatValue)

    return[ReturnColourThreads, ReturnColourProcess];
  }
```

Listing 5: Function ColourAssignment

Den Prozessen und Threads wird in der Hilfsfunktion Assignment, die zugehörige Farbe entsprechend der Auslastung zugewiesen.

4.2.3 Hilfsfunktion Assignment

Die Funktion Assignment filtert in der ersten Abfrage undefinierte Werte heraus. Dies ist notwendig um die minimale bzw. maximale Auslastung mithilfe von Math.min /.max zu bestimmen. Beinhaltet ein Array den Wert „undefined“ liefern die Aufrufe Math.min bzw- Math.max „NAN“ als Ergebnis.

```
function Assignment(HeatValue :any)
{
  //remove undefined Values for min/max
  const MathValue = HeatValue.filter(function(element : any) {
    return element !== undefined;
 });
  let min = Math.min(...MathValue)
  let max = Math.max(...MathValue)
  let test = 0
  var ReturnColour = new Array
  let j = 0
  for (; HeatValue[j] >= (min+test*((max-min)/datalength)) &&
       (j < datalength);) {
    if (HeatValue[j] <= (min+(test+1)*((max-min)/datalength))){
      ReturnColour[j] = Colour[test]
      j++
      test=0
      if (HeatValue[j] == undefined) {
        j++
      }
    }
    else{
      test++
    }
  }
  return(ReturnColour);
}
```

Listing 6: Function Assignment

Die Zuweisung der Farben erfolgt über eine schrittweise Abfrage. Die Schrittweite wird durch den Bereich zwischen minimaler und maximaler Auslastung, geteilt durch die gesamte Datenlänge gebildet. Dadurch wird eine lineare Verteilung des Farbverlaufes auf die Auslastung gewährleistet. Jede Position des Arrays Heat-Value durchläuft diese Abfrage bis die Auslastung kleiner oder gleich dem Wert (min + Schritt*Schrittweite) der aktuellen Abfrage ist

Für maximale Auslastung (min + maximale Anzahl Schritte*Schrittweite) entspricht mit maximale Anzahl Schritte der höchsten Position des Arrays Colour (gebildet in Coloursteps) und erhält eine rote Färbung.

Für Arraypositionen mit dem Wert „undefined“ wird die Zuweisung übersprungen.

4.2.4 Function Filesystem Assignment

```
function FilesystemAssignment(lFilesystemValue: any){
  var HelpArr = new Array
  for (let i = 0; i < data.series.length-datalength; i++) {

    HelpArr[i] =data.series[i+datalength].fields[1].values

  }
  let j = 0,k = 0
  for (let i = 0; i < datalength; i++) {
    lFilesystemValue[i] = HelpArr[j].buffer[k]
    k++
    if (HelpArr[j].buffer[k] == undefined) {
      j++
      if (j >= data.series.length-datalength) {
        break
      }
      k=0
    }
  }
  return(lFilesystemValue)
}
```

Listing 7: Function FilesystemAssignment

Die Funktion FilesystemAssignment weist Prozessen das Filesystem zu, in dem sie ausgeführt wurden. Hierbei wie in der Function ColourAssignment ein Workaround verwendet, um auf das Sub-Array buffer zuzugreifen. Dieses enthält das verwendeten Filesystem. Dem Array lFilesystemValue werden die im HelpArr.buffer enthaltene Werte zugewiesen. Da die Größe des Sub-Arrays buffer variiert ist eine Fallabfrage für das Überschreiten der Länge des Sub-Arrays.

4.2.5 Functions Build Panel

Die Funktion BuildPanel für Prozesse, Threads und Filesystems unterscheiden sich minimal, aber die zugrundeliegende Logik ist identisch. Daher wird im Folgendem nur die Funktion BuildPanelProcess betrachet.

```
function BuildPanelProcess(i: number,Colour: any)
.{
.  return (
.  //horizontal
.  <g>
.  <rect x={(5+150*i)} y={(50)} width={135} height={50} rx="10" fill={Colour}
.  <text x={(10+150*i)} y= {(90)} font-family="Arial" font-
    size="30" .fill="black">{data.series[i].fields[1].labels?.processid}</tex
.  <rect x={(150*i-3)} y={(0)} width={3} height={height} fill="black"/>
.  </g>
.  );
}
```

Listing 8: Function BuildPanel

Die Aufgabe der BuildPanel Funktionen ist es, die für die Visualisierung im Dashboard benötigten Daten im CSS-Format einzubetten und die Darstellung zu parametrieren. BuildPanel Funktionen werden pro Prozess / Thread je einmal aufgerufen. Doppelte Abfragen für Prozesse und die Filesystem-Ausgabe werden durch eine zusätzliche Ausgabe verhindert.

```
//Generieren der Css-Daten für alle Prozesse & Threads
let j = 0
for (let i = 0;i < datalength; i++) {
  if(ProcessColour[i] !== undefined)
  {
    ProcessIDArrCss[i] = BuildPanelProcess(i, ProcessColour[i])
    FilesystemArrCSS[i] = BuildPanelFilesystem(i, j)
    j++;
  }
  ThreadIDArrCss[i] = BuildPanelThread(i, ThreadColour[i])
}
```

Listing 9: Aufruf BuildPanel Funktionen

4.2.6 Gesamtausgabe

Ergebnisse der BuildPanel Funktionen mit einigen zusätzlichen CSS-Elementen, um das angezeigte Plugin lesbarer zu machen. Das Element „CustomScrollbar“ aktiviert die Option innerhalb des Plugins im Dashboard zu scrollen, sofern der bei height und width angegebene Bereich nicht vollständig darstellbar ist. Da vertikales Scrollen nicht möglich sein soll, entspricht die Variable height immer der vorgegebenen Höhe des Plugins innerhalb des Dashboards. Das horizontale Scrollen ist möglich und im Bereich der angezeigten Prozesse möglich. Das „CustomScrollbar“-Element kann aus dem offiziellen Grafana GitHub entnommen werden [1]

```
return(
  <div>
    <CustomScrollbar>
      <svg width={datalength*150} height={height}>
        <text x={(5)} y= {(30)} font-family="Arial" font-size="20"
       .°fill="White" font-weight="bold">Prozesse:</text>
        {ProcessIDArrCss}
        <text x={(5)} y= {(150)} font-family="Arial" font-size="20"
       .°fill="White" font-weight="bold">Threads:</text>
        {ThreadIDArrCss}
        <text x={(5)} y= {(270)} font-family="Arial" font-size="20"
       .°fill="White" font-weight="bold">Filesystem:</text>
        {FilesystemArrCSS}
      </svg>
    </CustomScrollbar>
  </div>
```

Listing 10: Return ThreadMap

4.3 Zusatz Optionen in Grafanaoberfläche integrieren

Um die Benutzerfreundlichkeit des Plugins weiter zu erhöhen, ist die Funktionalität der in der Oberfläche parametrierbaren Optionen mit verwendet.

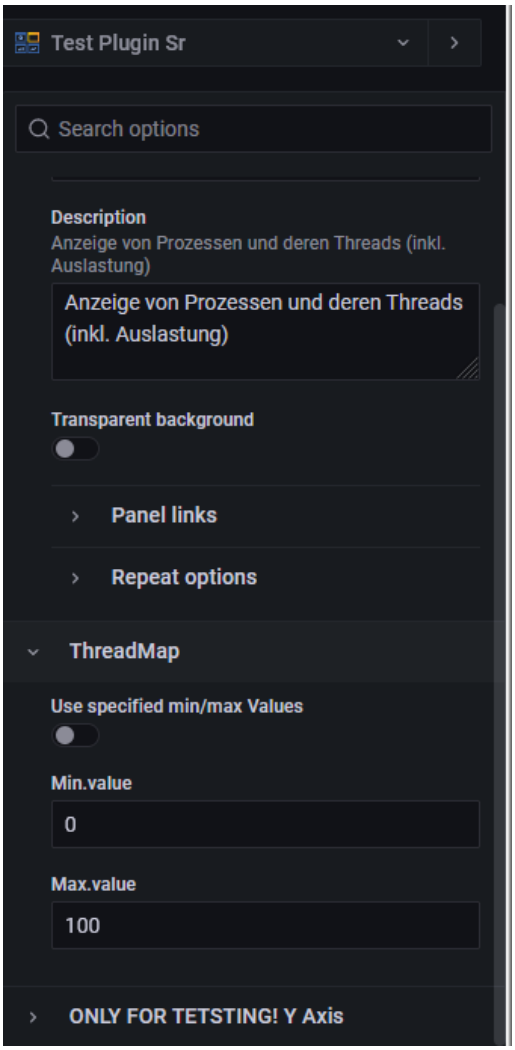

Abbildung 4-2: Parametrierbare Optionen

Wie in Abbildung 4-2 zu sehen ist, können dort aktuell spezifische Werte für die minimal Werte, die eine grüne Färbung erhalten sollen und für die maximal Werte, die eine rote Färbung erhalten, eingetragen werden. Alle Werte kleiner, größer dieses Bereichs sind entweder Grün oder Rot. Werten innerhalb dieses Bereiches werden mithilfe der Funktion ColourAssignment entsprechende Färbungen zugewiesen.

Hierbei ist anzumerken, dass diese Funktionalität aktuell noch nicht fehlerfrei umgesetzt ist, da die Anbindung am die Funktion ColourAssignment noch nicht umgesetzt ist.
Das hinzufügen weiterer parametrierbarer Optionen ist leicht machbar, da die dafür notwendige Architektur bereits erstellt bzw. angepasst wurde.

```
//in file options.ts (gekürzte Version für Dokumentation)
export interface ThreadMapColor {
    min?: number
    max?: number
}

export interface PanelOptions {
  minmax: ThreadMapColor
}

export const defaultPanelOptions: PanelOptions = {
    minmax: {
        min: 0,
        max: 100
    }
};
```

Listing 11: Define Parametrierbare Optionen

In der Datei options.ts können die gewünschten Optionen vordefiniert werden. Dazu wird ein Interface mit den pro Option benötigten Variablen erstellt und im Interface PanelOptions aufgerufen.
Das abgeleitete Interface defaultPanelOptions weist den verwendeten Optionen vorab Parameter zu, die den Standardeinstellungen des Panels entsprechen (Für minmax gibt es einen zusätzlichen Boolean Switch, falls man diese verwenden will. Siehe Listing 12: Build Parametrierbare Optionen).
Das Interface PanelOptions gibt die gewünschten Parameter über das Interface ThreadMapPanelProps an die „Main“-Funktion ThreadMap weiter, sodass die Parameter dort verwendet werden können. Zusätzlich wird das Interface PanelOptions in der Datei module.ts aufgerufen, die für das Erstellen der grafischen Oberfläche zur Verwendung der Optionen zuständig ist.

```
//in file module.ts (gekürzte Version für Dokumentation)
export const plugin = new PanelPlugin<PanelOptions>(ThreadMap).setPanelOp-
tions(builder => {
  let category = ['ThreadMap']
  builder
  .addBooleanSwitch({
    path: 'UseMinMaxBoolean',
    name: 'Use specified min/max Values',
    defaultValue: false,
    category,
  })
    .addNumberInput({
      path: 'ThreadMapColor.min',
      name: 'Min.value',
      defaultValue: defaultPanelOptions.minmax.min,
      settings: {
        placeholder: 'Auto',
      },
      category,
    })
    .addNumberInput({
      path: 'ThreadMapColor.max',
      name: 'Max.value',
      defaultValue: defaultPanelOptions.minmax.max,
      settings: {
        placeholder: 'Auto',
      },
      category,
    })
  }
})
```

Listing 12: Build Parametrierbare Optionen

In der Datei module.ts werden, wie bereits oben erwähnt die für die Optionen benötigten grafischen Elemente erstellt.

Der String category gibt hierbei die zugehörigkeit der Optionen an den entsprechenden Reiter an. Mithilfe eines Buillders werden anschließend die vordefinierten benötigten Grafik-Elemnte generiert, beschrieben und den zugehörigen Variablen der Optionen zugewiesen.

5 Ergebnis

Da die aktuelle Funktionalität bereits im vorherigen Kapitel ausführlich erwähnt wird folgt hier nur eine kurze Auflistung von bereits implementierten und geplanten, aber nicht umgesetzten Funktionen. Auf Tätigkeiten, wie das Anpassen von Test wird nicht weiter eingegangen (genauere Informationen siehe Kapitel 3).

Implementierte Funktionen:

- ProzessIDs nicht doppelt abbilden
- Einbinden einer Scrollbar in CSS
- Färbung Threads anhand deren Auslastung (Anzahl written Bytes)
- Färbung der Prozesse anhand der unterlagerten Threads
- Verbindung von Prozessen und Threads
- Umschreiben der Properties:
    - Dem Plugin können im FrontEnd Optionen zugewiesen werden, die im Backend ausgewertet werden
    - Einstellbare min. und max. Werte für die Färbung
- Filesystem unterhalb der Heatmap anzeigen

Aktuell vorhandene Fehler:

- Einstellbare min und max. Werte für die Färbung
    - FrontEnd ist implementiert, Anbindung Backend ist fehlerhaft
- Filesystem unterhalb der Heatmap anzeigen
    - Nach aktuellem Stand nur mit Daten aus fehlerhaftem Test realisiert, zusätzliche Anpassungen mit fehlerfreien Daten notwendig
- - Sortierung nach Auslastung
    - Vorsortieren über Flux Query nicht sinnvoll lösbar
        - Daten müssen gruppiert werden
        - Dadurch werden Properties die zuvor einmal geschrieben wurden für jeden Datenpunkt einzeln geschrieben
        - Unnötige Vergrößerung der Daten
    - In Plugin anhand von Bubblesort möglich
        - Auswirkungen auf Performance aktuell unklar

Ausblick:

- Allgemeine Optimierung und Anpassung
    - Automatisch Anpassbare Größe der Umrahmung von Prozess-/ThreadID anhand max. Zeichenlänge derer
    - Erweiterung horizontal/vertikal (Plugin passt CSS automatisch an voreingestelltes Fenster an)
- Live-Test und Lasttests
    - Aktuell nur mit Backuptests
    - Verhalten bei Live-Änderungen aktuell unklar
    - Wie verhält sich das Plugin bei größeren Datenmengen
- Sortierung anhand des Filesystems
    - Ziel: Verwendete Filesystem als Bottleneck identifizieren
- Drill Down der Heatmap für genauere Informationen
    - Forcegraph als zusätzlicher Plugin im Dashboard
    - Node des angewählten Prozesses/Threads wird im Forcegraph dargestellt

6 Literaturverzeichnis

[1] „InfluxDB OSS 2.0 Documentation,“ [Online]. Available: https://docs.influxdata.com/influxdb/v2.0/get-started/. [Zugriff am 26 02 2023].

[2] „InfluxDB Cloud Query Data,“ [Online]. Available: https://docs.influxdata.com/influxdb/cloud/query-data/. [Zugriff am 26 02 2023].

[3] „Grafana Documentation Visualization,“ [Online]. Available: https://grafana.com/docs/grafana/v8.4/visualizations/. [Zugriff am 27 02 2023].

[4] „Build a panel plugin with D3.js,“ Grafana Labs Team, [Online]. Available: https://grafana.com/tutorials/build-a-panel-plugin-with-d3/. [Zugriff am 27 012 2023].

[5] „BwUniCluster Wiki,“ [Online]. Available: https://wiki.bwhpc.de/e/BwUniCluster2.0/Batch_Queues. [Zugriff am 27 02 2023].

[6] „Grafana Custom Scrollbar,“ [Online]. Available: https://github.com/grafana/grafana/tree/main/packages/grafana-ui/src/components/CustomScrollbar. [Zugriff am 20 02 2023].

7 Abbildungsverzeichnis

8 Listings

9 Anhang

Plugin ThreadMap:

https://github.com/hpcraink/fsprj2/tree/master/Live-Tracing/grafana-Plugins/ThreadMap-panel

Dashboard:

https://github.com/hpcraink/fsprj2/blob/master/Live-Tracing/provisioning/dash-boards/ThreadMap_dashboard.json