Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bericht

Diamond

vorgelegt von

Niklas Wittmer

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Informatik
Matrikelnummer:	6800916
Betreuer:	Michael Kuhn

Hamburg, 30. April 2018

Inhaltsverzeichnis

1. Einleitung

Diamond ist ein Monitoring-Tool, welches verschiedene System-Metriken sammeln und diese in unterschiedlichen Formaten weiterverarbeiten und ausgeben kann. Das Tool wurde in Python geschrieben und verfolgt einen modularen Aufbau. Es kann also relativ leicht durch eigene Plugins erweitert werden.

Diamond teilt sich in drei Bestandteile auf. Objekte der Klasse `Collector` dienen dazu spezifische Metriken zu sammeln. Dies können alle Formen von (numerischen) Daten sein, wie etwa CPU-Last oder Speicherverbrauch.

Gesammelte Metriken werden von Objekten der Klasse *Handler* verarbeitet, beispielsweise in Dateien abgelegt oder an MySQL-Datenbanken gesendet.

Der Server koordiniert die Handler und Collectors. Er startet und stoppt sie also und ruft sie mittels Scheduler auf.

Sowohl für Handler als auch für Collectors lassen sich sehr leicht eigene Unterklassen schreiben, die jeweils im simpelsten Falle nur eine Methode implementieren müssen. Den Aufruf dieser Methoden übernimmt sodann der Server.

Laut Aussage der Entwickler kann Diamond bis zu 3 Millionen Datenpunkte pro Minute sammeln und verarbeiten.[1,2]

Score-P ist eine Infrastruktur zur Messung der Performance in parallelen Umgebungen und will eine hoch skalierbare und einfach zu verwendende Menge an Tools anbieten. Der zu analysierende Programm-Code kann dazu in vielen Fällen einfach automatisch instrumentalisiert werden, indem eine komplette Kompilieranweisung an den Befehle `scorep` übergeben wird (z.B. `scorep gcc ...`). So können mithilfe zusätzlicher Tools wie Vampir[3] Zeitpunkte und Dauer von Methodenaufrufen und Interprozess-Kommunikation zu visualisieren. Die Infrastruktur selbst kann mittels Umgebungsvariablen konfiguriert werden.

Zusätzlich bietet Score-P ein Interface zum Schreiben von Plugins an. Diese *Metric-Plugins* können zum Sammeln beliebiger Daten verwendet werden.

[1] Gaurav Jain u. a. *Diamond*. URL: `https://github.com/python-diamond/Diamond`.

[2] python-diamond contributors. *Diamond-Dokumentation*. URL: `http://diamond.readthedocs.io/en/latest/`.

[3] Score-P. *Vampir 9.4*. URL: `https://www.vampir.eu/` (besucht am 21. 03. 2018).

1.1. Aufgabe

Am Arbeitsbereich WR wurde das Tool *pmlib* entwickelt, welches ähnlich zu Diamond mittels unterschiedlicher Plugins Messwerte verarbeiten kann. Für dieses Tool existieren Plugins für unter anderem zwei Leistungsmessgeräte. Das LMG450, welches ein kommerzielles Produkt ist, und das Ardupower, welches von Manuela Beckert und Janosch Hirsch in der Arbeitsgruppe WR entwickelt wurde.[4]

Meine Aufgabe bestand darin, die *pmlib*-Plugins für die Leistungsmessgeräte LMG450 und Ardupower zu Diamond zu portieren. Dazu musste also jeweils ein Collector geschrieben werden, welcher in der Methode `collect` Daten auf der Schnittstelle abfragt.

Anschließend sollte, abhängig von der verfügbaren Zeit, eine Kompatibilität zu dem Framework *Score-P* hergestellt werden. Dies erforderte das Entwickeln eines *Metric Plugins* für Score-P, welches mit Score-P kommunizieren soll, um entsprechende Daten abzufragen.

Die Funktionsweise der Plugins in pmlib unterscheidet sich leicht von der in Diamond. Pmlib startet Plugins einmal und diese sammeln darauf in einer Endlosschleife Datenpunkte und reichen diese an den Server zurück.

In Diamond ruft der Server die `collect`-Methode an jedem aktivierten Plugin auf. Dies geschieht mit einer im Plugin festgelegten Frequenz.

Der Schwerpunkt lag letztendlich darin, die Logik von pmlib in die von Diamond zu übersetzen.

[4] Manuela Beckert und Janosch Hirsch. *A low cost power measurement device to improve energy efficiency of HPC devices*. 5. Aug. 2015. URL: `https://wr.informatik.uni-hamburg.de/_media/teaching/wintersemester_2014_2015/pre-1415-ardupower-report.pdf`.

2. Diamond Collector-Plugins

Die grundlegende Struktur der Collector-Klasse von Diamond ist in Listing 2.1 dargestellt.

```
1  class ExampleCollector(diamond.diamond.collector):
2
3      def get_default_config(self):
4          config = super(ExampleCollector,
               ↪ self).get_default_config()
5          config.update({
6              'path': 'example.metric',
7              'interval': 10,
8              'ttl-multiplier': 2,
9          })
10         return config
11
12     def get_default_config_help(self):
13     config_help = super(ExampleCollector,
         ↪ self).get_default_config_help()
14     config_help.update({
15             'path': "Name of the Collector",
16             'interval': "Number of seconds between two
                 ↪ calls",
17             'ttl-multiplier': "Factor for timeout when
                 ↪ collect failed",
18         })
19         return config_help
20
21
22     def collect(self):
23         metric_name = "example.metric"
24         value = 42
25
26         self.publish(metric_name, value, precision=2)
```

Listing 2.1: Struktur einer von Collector erbenden Klasse

Die wichtigste Methode hier ist die `collect`-Methode. Diese ermittelt die gewünschten Datenpunkte und übergibt sie an den Server. Die Methode `get_default_config` ist nur dann erforderlich, wenn über die Konfiguration eigene Parameter gesetzt werden

müssen oder Diamonds Default-Werte überschrieben werden sollen. Zusätzlich dazu ist es ratsam bei Verwendung eigener Parameter eine Beschreibung dieser in der Methode `get_default_config_help` anzugeben, damit Benutzer den Sinn der Parameter kennen.

Da beide Leistungsmessgeräte über serielle Schnittstellen angesprochen werden, war die `default_config` um entsprechende Parameter zu erweitern. Hier mussten die Parameter für den Konstruktor der Klasse `Serial` aus dem Modul `serial` festgelegt werden können, um eine korrekte serielle Verbindung aufbauen zu können.

2.1. Ardupower

In dem Plugin wird - wie bei pmlib üblich - die Methode `read` verwendet, um mit dem Gerät zu kommunizieren. Dort wird zu Anfang ein Setup durchgeführt, welches ausgehend von einer Konfiguration der Leitungen am Arduino die zu messenden Kanäle definiert. Anschließend wird eine *while*-Schleife betreten, welche Wert für Wert an der Schnittstelle ausliest. Voraus geht dabei ein längerer Codeblock, welcher der Fehlerkorrektur beim Lesen der Daten dient. Auf diese werde ich in Kapitel 4 noch einmal eingehen.

Die *while*-Schleife habe ich letztendlich so übernommen. Diese wird von `collect` aufgerufen und verlassen, sobald die Methode `publish` einmal erfolgreich aufgerufen wurde. Dies ist nötig, da gelesene Werte, für den Fall, dass diese nicht korrigiert werden können, komplett verworfen werden. Da ein erneuter Aufruf länger dauern würde als ein erneutes Betreten der Schleife, habe ich mich dafür entschieden, dies so zu lösen. Da zugehörige Zeitstempel erst beim Aufruf von `publish` erfasst werden, führt dies hier auch nicht zu fehlerhaften Daten.

Die ursprüngliche Implementation des Plugins weist leider einige Schwächen auf. So fiel mir zunächst auf, dass vor dem Lesen an der seriellen Schnittstelle ein Aufruf von `flushInput` am Objekt fehlt. Dazu ist zu wissen, dass die Werte, die das Ardupower-Gerät sendet, in einem Input-Puffer abgelegt werden, von welchem sich das Plugin jeweils die nächsten Bytes besorgt.

Dies ist solange kein Problem, wie die Lesegeschwindigkeit mindestens der Schreibgeschwindigkeit entspricht. Laut den Autoren erreicht aber das Leistungsmessgerät Frequenzen von bis zu 5000 Datenpunkten pro Sekunde. In pmlib wurde diese Lesegeschwindigkeit jedoch nie erreicht.

Wenn nun 100 Mal in der Sekunde am Interface gelesen wird, während das Gerät selbst 5000 Mal in derselben Zeit Daten sendet, sind veraltete Daten, die zu spät ausgelesen werden, das Ergebnis. Daher habe ich zusätzlich den Aufruf `self.SERIAL.flushInput()` vor betreten der *while*-Schleife hinzugefügt.

Ein weiterer Fehler lag in der Berechnung der tatsächlichen Leistung, die an einer Leitung gemessen wurde. Jede Leitung benötigt dazu einige vorher zu errechnende Korrekturwerte *offset*, *slope* und *voltage*. Zur Berechnung dieser Werte - abgesehen von *voltages* haben die Autoren zusätzlich ein Skript geschrieben.

Die Leistung an jeder Leitung i wird dann wie folgt berechnet:

$$power_i = (value_i \div slope_i - offset_i) \cdot power_i \tag{2.1}$$

Hier wurde jedoch vergessen, die korrekte Spannung zu wählen. Letztendlich wurde für jede Leitung die Spannung einer undefinierten Leitung „angelegt“, wodurch fehlerhafte Messwerte entstanden.

Den Hauptteil der `collect`-Methode in diesem Plugin macht eine fehlerkorrigierende Routine aus, welche in Listing 2.2 abgebildet ist. Diese versucht bei fehlerhaften Eingaben, ebendiese zu korrigieren. Wie in Kapitel 4 zu sehen entsteht hier vermutlich ein Leistungsengpass.

```
1  data_read = self.SERIAL.read(size=self.n_bytes)
2  var = True
3  var2 = False
4
5  # if MSB != 1 -> out of sync! resync...
6  if ord(data_read[0]) >> 7 == 0:
7      for k in range(self.n_bytes):
8          # If sync possible then continue, otherwise this
               ↪ data
9          # is to be dropped
10         if ord(data_read[k]) >> 7 == 1:
11             # drop the faulty part and append the next k
                   ↪ bytes
12             data_read = data_read[k:] +
                   ↪ self.SERIAL.read(size=k)
13             var = False
14             break
15         # start a new read if resync failed
16         if var:
17             self.log.debug("Unable to correct data. Drop
                   ↪ it.")
18             self.SERIAL.flushInput()
19             continue
20
21 # if any of the bytes is still invalid, drop data and start
       ↪ over
22 for k in range(self.n_bytes - 1):
23     if(ord(data_read[k + 1]) >> 7 == 1):
24         var2 = True
25         break
26     if var2:
27         # starting over
28         self.log.debug("Data looks invalid. Drop it.")
29         self.SERIAL.flushInput()
30         continue
```

Listing 2.2: Fehlerkorrektur in der `collect`-Methode des ArdupowerCollector

2.2. LMG450

```
1  def collect(self):
2      """
3      Collect stats from LMG450
4      """
5      self.SERIAL.flushInput()
6      while True:
7      try:
8          sample = self.SERIAL.read(self.size)
9      except serial.SerialException, e:
10         self.log.error(str(e))
11         continue
12     r = struct.unpack('<7c4fc', sample)
13     self.publish("0", r[7], precision=5)
14     self.publish("1", r[8], precision=5)
15     self.publish("2", r[9], precision=5)
16     self.publish("3", r[10], precision=5)
17     break
```

Listing 2.3: `collect` von Lmg450Collector

Die Portierung des LMG450-Plugins gestaltete sich etwas leichter als die des Ardupower-Plugins. Das Setup wird hier ebenfalls im Konstruktor vorgenommen. In der `collect`-Methode ist es anschließend nur noch nötig eine Leseoperation durchzuführen. Die gelesenen Werte können dann direkt weitergegeben werden. Die Gesamte Methode ist in Listing 2.3 abgebildet.

3. Score-P-Support

Der zweite Teil meiner Aufgabe bestand darin, ein Metric-Plugin[1] für Score-P zu schreiben, um die von Diamond gesammelten Daten visuell mit Programmabläufen vergleichbar zu machen.

Abbildung 3.1 zeigt das Ergebnis einer beispielhaft durchgeführten Messung, bei der über vier Knoten des WR-Cluster und acht Tasks ein MPI-Programm ausgeführt wurde. Aufgezeichnet wurden dabei die Kanäle des LMG450. Zu sehen sind die Messwerte über alle Kanäle *amd5.lmg450.SUM* und die des zweiten Kanals *amd5.lmg450.1*.

3.1. Metric-Plugin für Diamond

Um in Score-P verwendet werden zu können, werden die Plugins als *shared objects* kompiliert. Damit sie gefunden werden, müssen sie in einem Pfad liegen, in dem die Ausführungsumgebung Shared Objects erwartet. Unter Linux ist dies etwa durch aktualisieren der Umgebungsvariable `LD_LIBRARY_PATH` möglich.

Um das Metric-Plugin Diamond zu aktivieren sind folgende Schritte notwendig:

1. Plugin-Namen übergeben mit `SCOREP_METRIC_PLUGINS=Diamond_plugin`
2. Paramter an das Plugin übergeben:
   `SCOREP_METRIC_DIAMOND_PLUGIN="HOST:PORT:DEVICE:LINES"`
3. *Tracing* aktivieren: `SCOREP_ENABLE_TRACING=true`
4. *Profiling* deaktivieren: `SCOREP_ENABLE_PROFILING=false`
5. Pfad zum `LD_LIBRARY_PATH` hinzufügen,
   etwa `LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$PWD`

Der `Parameter-String` ist also ein reiner Textstring, dessen Inhalt vom Plugin selbst entsprechend geparsed werden muss. Der Vorteil hier liegt darin, dass so beliebige Parameter in beliebigen Formaten übergeben werden können.

Vom Plugin erwartet Score-P dann einige Funktionen, welche zum Setup, Betrieb und sauberen Beenden benötigt werden. Im folgenden Abschnitt möchte ich auf die für das Diamond-Plugin relevanten Funktionen eingehen. Für eine vollständige Liste sei hier auf die Dokumentation [Par18] von Score-P verwiesen.

[1] Score-P Partners. *Score-P: Introduction*. 2018. URL: `https://silc.zih.tu-dresden.de/scorep-current/index.html` (besucht am 21. 03. 2018).

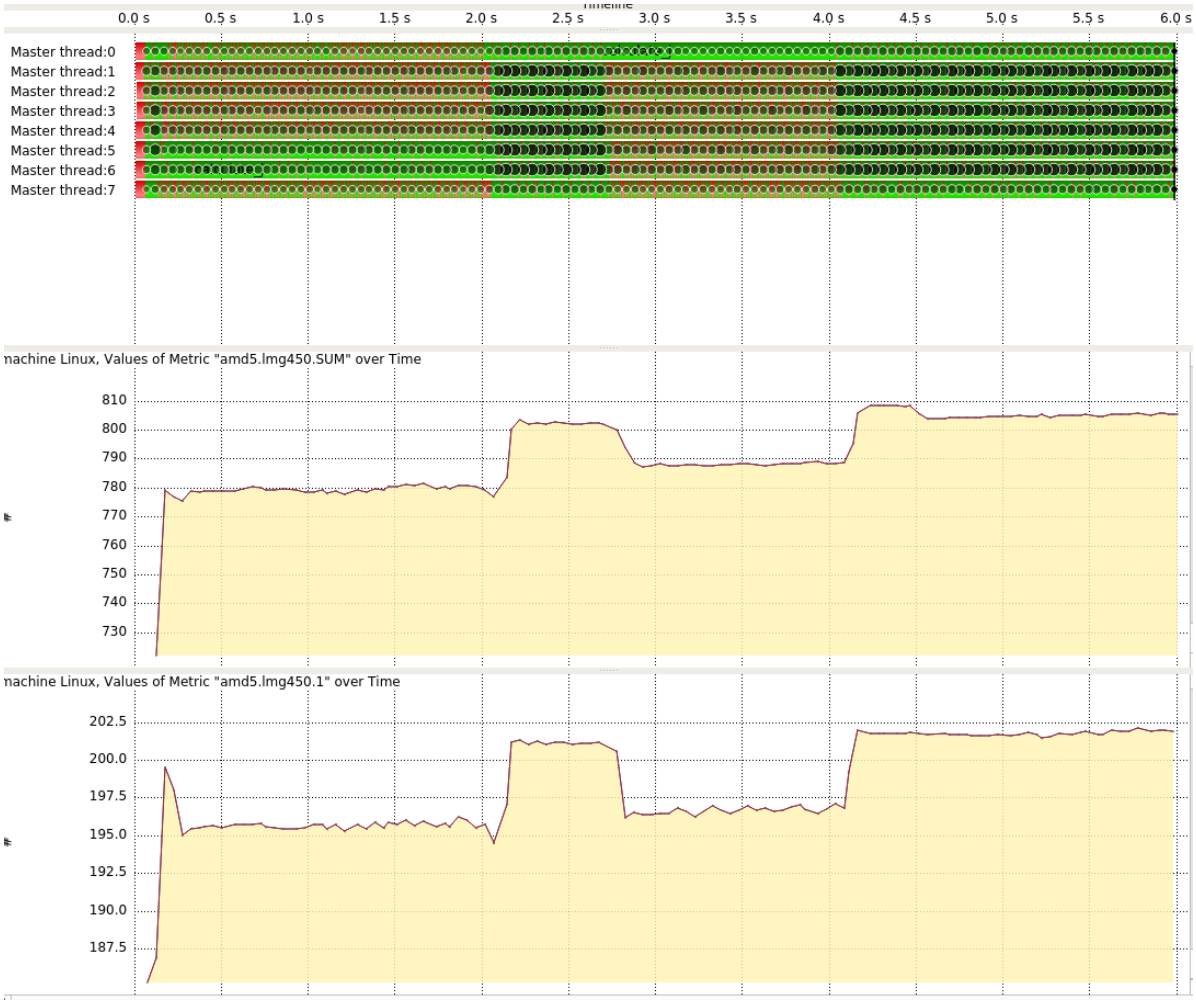

Abbildung 3.1.: Darstellung einer Messung mit Score-P und Diamond bei aktiviertem lmg450-Plugin

3.1.1. Kommunikationsprotokoll

Die Kommunikation zwischen Server und Client sollte möglichst einfach gehalten werden.

Der Client sendet zu Beginn jeder Kommunikation einen Integer als Request. Der Server wird dann in einen entsprechenden Zustand versetzt, woraufhin eine definierte Abfolge von Nachrichten ausgetauscht wird.

Die Beschreibung des Protokolls ist in Listing 3.1 dargestellt.

```
1  % Create Counter:
2  Score-P Client --------------------------- Diamond Server
3
4                 D_CREATE ----------------->
5                 counter name length ------>
6                 counter name ------------->
7
8                 <--------------confirmation
9
10 % Get All Values:
11 Score-P Client --------------------------- Diamond Server
12
13                D_GET_VALUES ------------->
14                strlen(counter name)------>
15                counter name ------------->
16
17                <number of time/value pairs
18                <---Diamond start timestamp
19                <----Diamond stop Timestamp
20                <----------time/value pairs
21
22 % Synchronize:
23 Score-P Client --------------------------- Diamond Server
24
25                D_[START|STOP] ----------->
26
27                <--------------confirmation
```

Listing 3.1: Nachrichten-Protokoll des Metric-Plugins

3.2. Aufbau des Plugins

Das Plugin wird zunächst mittels des Makros `SCOREP_METRIC_PLUGIN_ENTRY(Diamond_plugin)` registriert. Hier werden die Callbacks zu den von Score-P verlangten Funktionen definiert.

1. `run_per = SCOREP_METRIC_ONCE` bestimmt, dass alle Aufrufe des Plugins von einem Host ausgeführt werden.

2. `sync = SCOREP_METRIC_ASYNC` sorgt dafür, dass alle gesammelten Metriken erst zum Ende der Laufzeit des instrumentierten Programms eingesammelt werden. Da die Kommunikation über TCP die Ausführung andernfalls stark verzögern würde, was zu weniger eindeutigen Ergebnissen führen könnte, habe ich mich gegen die synchrone Methode entschieden.

3. Der Aufruf zu `initialize` wird einmal zu Beginn der Ausführung aufgerufen. Im vorliegenden Plugin wird hier lediglich der Variable `counters` Speicher zugewiesen.

4. Komplementär zu `initialize` wird `finalize` einmal am Ende der Ausführung aufgerufen, um das Plugin ordentlich zu beenden.

5. Die Funktion `get_event_info` bekommt als Eingabe den Parameter-String übergeben und erzeugt entsprechend dieser die gewünschten *Counter*.

   Das Diamond-Plugin akzeptiert einen String der Form `HOST:PORT:DEVICE:LINES`

   `HOST` steht für den Hostnamen des Diamond-Servers. Entsprechend ist `PORT` der Port, auf dem der Server lauscht. `DEVICE` bestimmt das Gerät, von welchem Daten gesammelt werden. Dies könnte hier also etwa `lmg450` oder `ardupower` sein. Wichtig ist, dass die Schreibweise der *Metric-Names* in Diamond entspricht.

   `LINES` sind die aufzuzeichnenden Counter in Diamond. Das Plugin akzeptiert hier entweder das Stichwort `SUM`, welches die Summe aller verfügbaren Counter liefert, eine einzelne Zahl, eine Reihe durch kleinste und größte Zahl begrenzt oder einen beliebigen anderen Namen.

   In `get_event_info` werden nun also alle nötigen Zähler angelegt. Die nötigen Informationen werden zum einen in einer `struct counter` abgelegt und zusätzlich an Score-P übergeben. Der Aufbau `struct counter` ist in Listing 3.2 dargestellt.

```
1 struct counter {
2 char * ip;
3 int    port;
4 char   name[128];
5 int    line;
6 };
```

Listing 3.2: Aufbau der `struct counter`

   Ip-Adresse und Port werden später zur Kommunikation mit Diamond benötigt. Der Name entspricht dem Gerätenamen plus den spezifischen Teil des Counters. Wird der Counter über eine Zahl definiert, wird diese zusätzlich in `line` gespeichert.

6. In `add_counter` werden Counter-spezifisch Konfigurationsschritte vorgenommen. In diesem Fall wird dem ScorepHandler in Diamond mitgeteilt, dass er die Daten dieses Counters aufzeichnen soll.

7. Nachdem das eigentliche Programm ausgeführt wurde, werden per `get_all_values` alle aufgezeichneten Daten von Diamond angefragt und an Score-P übergeben.

   Hier wird jeweils ein Counter per ID übergeben, welche dem Counter in `add_counter` zugewiesen wurde. Dem Kommunikationsprotokoll entsprechend werden dann nach Beginn der Kommunikation alle Wert-/Zeit-Paare übermittelt. Da diese einen Zeitstempel besitzen, welcher von einem anderen Server stammt, müssen die Zeitstempel noch durch eine einfache lineare Interpolation zu denen von Score-P umgerechnet werden. Kleinere Abweichungen werden sich hier trotzdem nicht vermeiden lassen. Dies liegt jedoch in der Natur unabhängiger Server, welche die Zeit niemals exakt gleich messen werden.

   Die Werte werden in einem Array gespeichert und an Score-P übergeben.

8. `set_timer` legt fest, welche Funktion zum Messen der Zeit verwendet wird. Diese Information holt sich das Plugin zur Laufzeit über die Umgebungsvariable `SCOREP_TIMER`. Diese Funktion wird noch vor `initialize` aufgerufen.

9. `synchronize` wird direkt vor Beginn und nach Ende der Ausführung des zu messenden Programms aufgerufen, um so die Start- und Stopzeitpunkte bestimmen zu können. Hier wird ein entsprechender Request an alle beteiligten Diamond-Instanzen gesendet, die dann mit der Aufzeichnung der Daten beginnen.

Die Aufrufe all dieser Funktionen werden von der Score-P-Infrastruktur selbst übernommen. Der Entwickler des Plugins stellt diese lediglich bereit.

3.3. Diamond-Handler

Die Implementation des TCP-Servers, der mit dem Score-P-Plugin kommunizieren soll, stellte mich vor kleinere Hürden, welche vermutlich durch Python bedingt wurden.

```
1  def process(self, metric):
2      """
3      Process metric and send it through
4      the pipe to ScorepHandler's receiver.
5      """
6
7      path = metric.getMetricPath()
8      value = metric.value
9      timestamp = metric.timestamp
10     name = metric.getCollectorPath()
11
12     device = name + "." + path
13
14     self.parent_pipe.send([device, value, timestamp])
```

Listing 3.3: `process`-Methode des ScorepHandler

Diamond lässt Collectors und Handler mithilfe des Moduls *multiprocessing* in eigenen Prozessen mit ihren eigenen Namespaces laufen. Dies hat den offensichtlichen Vorteil, dass die Last, welche durch das Sammeln und Verarbeiten der Daten entsteht, auf Prozessorkerne verteilt werden kann.

Problematisch daran ist jedoch, dass diese Prozesse innerhalb eigener, unabhängiger Python-Instanzen laufen. Dies führte bei mir zu dem Problem, dass dort, wo vermeintlich ein und dasselbe Objekt agiert, tatsächlich zwei Kopien vollkommen unabhängig voneinander arbeiten. Die Folge war, dass ein Objekt von selbst getätigten Änderungen scheinbar nichts mitbekam. Da jedoch die Methode `process` Daten empfängt, welche vom Handler über einen Server weiterverarbeitet werden sollen, war es zwingend notwendig, dass der ScorepHandler zuverlässig Zugriff auf alle gesammelten Metriken hat.

Grundsätzlich können Prozesse über gemeinsamen Speicher Daten austauschen. Dazu dienen Threads. Threads werden in Python durch das Modul *threading* verwendet werden. In Python können Threads jedoch nicht parallel ausgeführt werden, was die Nutzung von Prozessen unverzichtbar macht.

Es existieren zwei Konzepte, mit denen Python-Instanzen Daten austauschen können. Dies sind *Pipes* und *Queues*. Eine Pipe ist ein Kanal, über den Prozesse mittels Funktionen `send` und `recv` Daten senden und empfangen können. Queues bilden eine globale Datenstruktur, in die Prozesse, wie bei Warteschlangen üblich Daten der Reihe nach schreiben und wieder auslesen können.

Da ich mit Queues ein ähnliches Problem wie das eingangs beschriebene hatte, entschied ich mich für die Verwendung einer Pipe. Dazu entwickelte ich für den ScorepHandler eine zusätzliche Methode `receive`, welche die Daten, die von der Methode `process` verarbeitet wurde empfängt. Die `process`-Methode tut daher kaum mehr, als die Daten über `send` zu verschicken. Dies hat zusätzlich den Vorteil, dass die Abarbeitung ankommender Metriken schnell erfolgen kann.

Listing 3.3 zeigt die `process`-Methode des ScorepHandler. Das verwendete Objekt `self.parent_pipe` ist eine Referenz auf das eine Ende der Pipe. Das andere Ende `self.child_pipe` wird in der Methode `receive` (Listing 3.4) verwendet, um die Daten zu empfangen. Diese Methode wird in einem eigenen Thread ausgeführt und empfängt Daten innerhalb einer *while*-Schleife, welche erst mit Beendigung von Diamond verlassen wird.

```
1  def receive(self, child_pipe):
2      """
3      Receive, check and store metrics from process function
4      """
5      global metrics
6      global counters
7      global start_time
8      global stop_time
9      global running
10     while True:
11         new_metric = self.child_pipe.recv()
```

```
12          metric_time = new_metric[2]
13          metric_value = new_metric[1]
14          metric_name  = new_metric[0]
15
16          if metric_name not in counters:
17              self.log.info("Adding " + metric_name + " to
                  ↪ counters list.")
18              counters.add(metric_name)
19              counters.add(metric_name.split(".")[0] + ".SUM")
20          if metric_name not in metrics:
21              metrics[metric_name] = []
22          if stop_time is not None and stop_time <=
              ↪ metric_time and running:
23              running = False
24          elif start_time is not None and start_time <=
              ↪ metric_time:
25              metrics[metric_name].append([metric_value,
                  ↪ metric_time])
```

Listing 3.4: `receive`-Methode des ScorepHandler

3.3.1. Zeitstempel in Diamond

Nach derzeitigem Stand sieht Diamond nicht vor, Zeitstempel mit einer höheren Genauigkeit als in Sekunden aufzuzeichnen. Etwaige Nachkommastellen werden abgeschnitten.

Dies ist jedoch nicht sinnvoll, wenn die Frequenz, in der Daten gesammelt werden möglichst hoch sein soll. Daher musste ich in dem Modul `Metric` das Verarbeiten des Zeitstempels so anpassen, dass Gleitkommazahlen akzeptiert werden.

```
1  if timestamp is None:
2      timestamp = int(time.time())
3  else:
4      # If the timestamp isn't an int, then make it one
5      if not isinstance(timestamp, int):
6      try:
7          timestamp = int(timestamp)
8      except ValueError as e:
9          raise DiamondException(("Invalid timestamp when "
10                                 "creating new Metric %r:
                                     ↪ %s")
11                                 % (path, e))
```

Listing 3.5: Initialisierung des Timestamps im Metric-Modul

Listing 3.5 zeigt den Code des Master-Branches im Metric-Modul. Es ist schnell ersichtlich, dass jeder eingegebene Zeitstempel als Ganzzahl ausgegeben werden wird.

Daher habe ich in Listing 3.6 den Code so verändert, dass Ganzzahlen nur dann verwendet werden, wenn die Präzision auf 0 gesetzt wurde. Dies ist eine hinreichende Lösung für das vorliegende Projekt. Da sich der Parameter `precision` jedoch eigentlich auf den eingegebenen Messwert bezieht, habe ich für den Patch bisher keinen Pull-Request an das Diamond-Projekt durchgeführt.

```
1  # If no timestamp was passed in, set it to the current time
2  if timestamp is None:
3      timestamp = time.time()
4      if precision == 0:
5          timestamp = int(timestamp)
6  else:
7      # If the timestamp isn't an int (or float), then make
       ↪ it one
8      if not isinstance(timestamp, (float, int)):
9          try:
10             timestamp = float(timestamp)
11             if precision == 0:
12                 timestamp = int(timestamp)
13         except ValueError as e:
14             raise DiamondException(("Invalid timestamp when"
15                                     " creating new Metric %r: %s")
16                                     % (path, e))
```

Listing 3.6: Initialisierung des Timestamps im Metric-Modul mit float-Patch

Abhängig vom verwendeten System werden nun präzisere Zeitstempel verwendet, sodass später im Metric-Plugin von Score-P genauere Umrechnungen vorgenommen werden können.

3.4. Der TCP-Server

Das Modul `scorep` enthält neben der Klasse `ScorepHandler` zusätzlich die Klassen `ThreadedTCPRequestHandler` sowie `ThreadedTCPServer`. Diese implementieren einen TCP-Server, welcher für jede eingehende Verbindung einen eigenen Request-Handler erzeugt.

Der Request-Handler bearbeitet also alle Requests, prüft somit, ob Counter wirklich existieren, bevor die Aufzeichnung beginnt. Dazu führt wiederum der ScorepHandler eine Liste aller aktiven Geräte und deren Counter.

4. Analyse

In diesem Abschnitt beziehe ich mich auf folgenden Versuchsaufbau:

Diamond lief auf dem nehalem5-Knoten des WR-Rechenclusters. Es waren der ArduPowerCollector und der ScorepHandler durchgehend aktiv. Gleichzeitig wurde auf Knoten 1 bis 4 aus der *amd*-Partition gerechnet.

Dazu verwendete ich den partiellen Differentialgleichungslöser (im folgenden *PDE* oder *partdiff-par* genannt), welcher an der TU München von Prof. Dr. Thomas Ludwig et al. entwickelt wurde. Dieser kann unter Verwendung des Jacobi- oder Gauß-Seidel-Verfahrens große partielle Differentialgleichungen lösen. Am Arbeitsbereich WR wird er hauptsächlich zu Lehrzwecken eingesetzt, dient aber auch hervorragend als einfacher Benchmark.

Das Programm lief mit folgender Konfiguration:

Auf jedem Knoten wurden zwei Tasks ausgeführt. Es wurde die Jacobi-Methode verwendet bei 4000 Interlines und der Störfunktion $f(x) = 0$. Es wurden außerdem 4000 Iterationen durchlaufen.

Das Programm wurde einmal ohne Score-P-Instrumentierung ausgeführt und ein weiteres Mal mit.

Der Versuchsaufbau diente dazu, die Plugins bei hohem Datenaufkommen zu beobachten.

4.1. Diamond-Plugins

Um sinnvoll eingesetzt werden zu können, sollten die Plugins eine möglichst hohe Aufzeichnungsfrequenz erreichen, damit Lasten auch im Millisekundenbereich erkennbar sind.

Das LMG450 ist auf dem Knoten *amd5* auf dem Rechen-Cluster des WR angeschlossen. Das Ardupower auf dem Knoten *nehalem5*. Die Spezifikationen der Hardware dieser beiden Knoten findet sich im Anhang unter Listing A.1 und Listing A.2.

Das LMG450 erreicht keine Frequenz höher als 20Hz. Dies ist ebenfalls so in pmlib konfiguriert, sodass ich davon ausgehe, dass dies mit dem Setup des Gerätes zusammenhängt.

Das Ardupower-Plugin weist eine relativ komplexe Fehlerkorrektur auf. Tests ergaben hier Frequenzen von bis zu etwa 330Hz pro Counter. Dies ist zwar um einiges höher als die Ergebnisse des LMG450Collectors; jedoch sollten auch hier noch um einiges bessere Raten erzielt werden. Die Autoren des Ardupower beschreiben Ausgabefrequenzen des Gerätes selbst von etwa 480Hz pro Counter. [BH15, Seite 33]

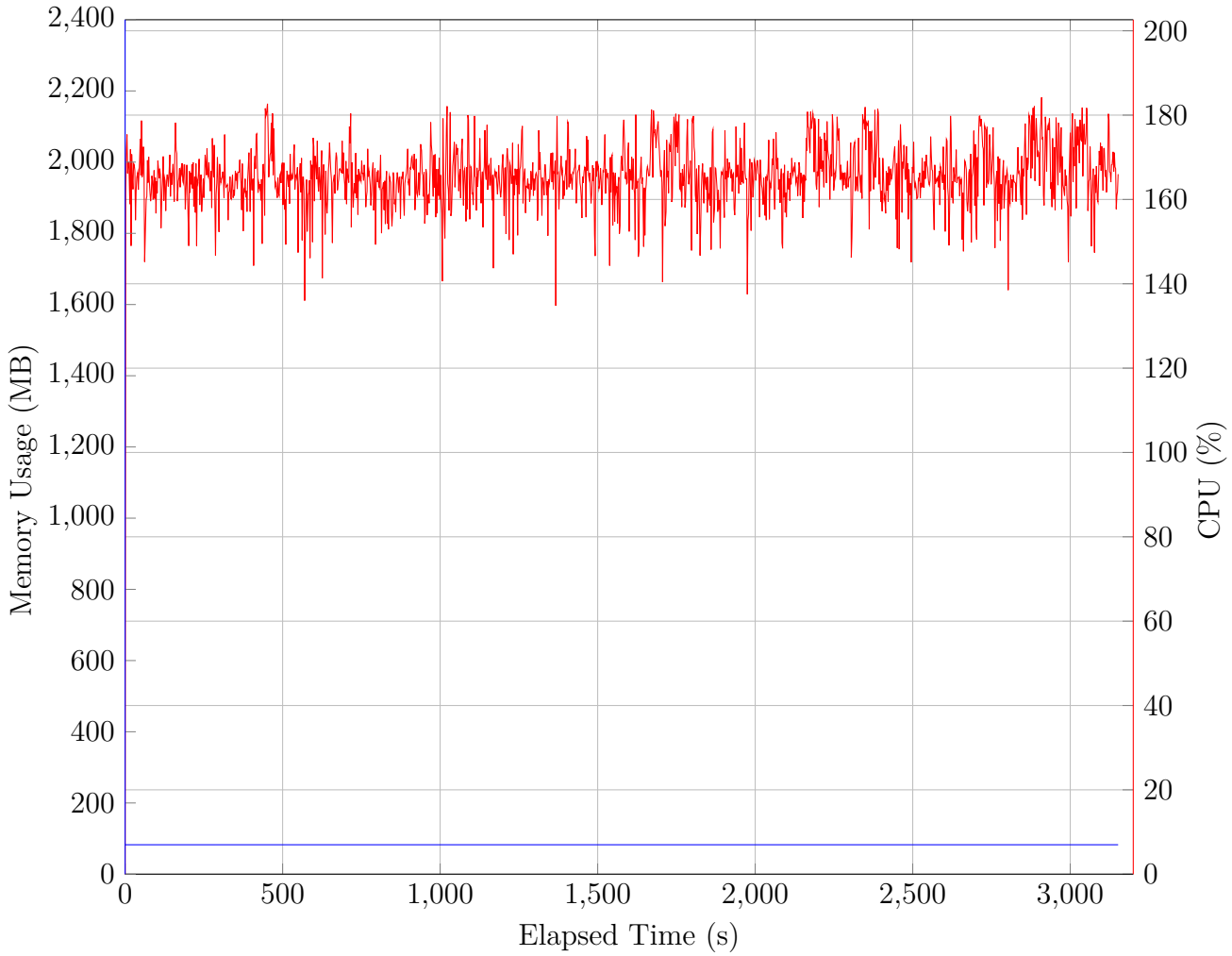

Abbildung 4.1.: Auslastung des nehalem5-Knoten durch Diamond

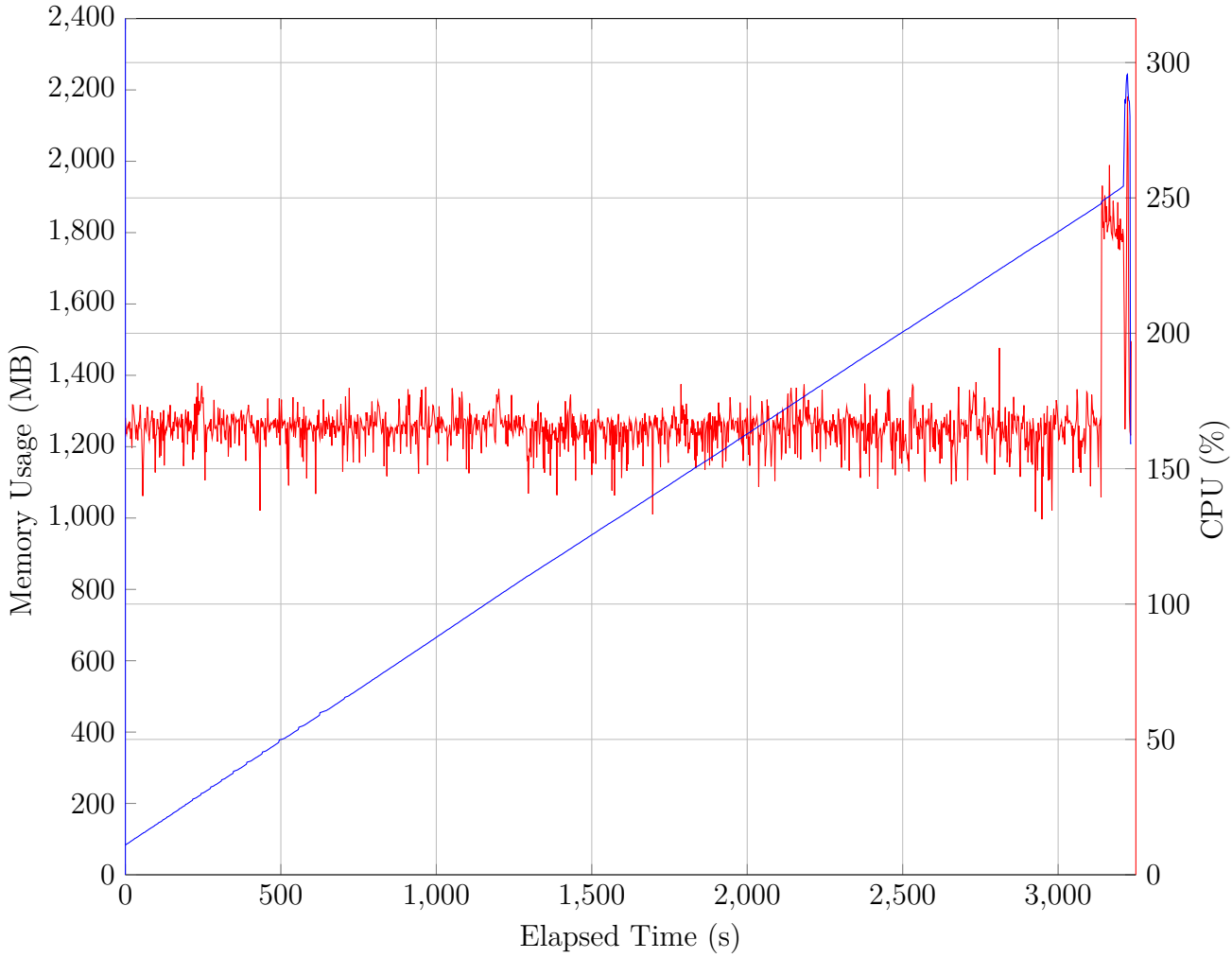

Abbildung 4.2.: Auslastung des nehalem5-Knoten, während Diamond läuft. Zusätzlich wird ein PDE mit Scorep-Instrumentierung ausgeführt.

Ich habe daher zunächst versuchsweise die komplette Fehlerkorrektur deaktiviert. Stattdessen sollte das Plugin asynchrone Eingaben einfach verwerfen und die nächste Eingabe lesen. Dies führte jedoch zu noch schlechteren Ergebnissen. Offenbar geht die Synchronizität hier sehr häufig verloren, sodass pro Runde mehrere Leseoperationen am seriellen Interface durchgeführt werden müssen. Die Ursache liegt darin, dass der Arduino schneller Daten sendet, als dass sie weder pmlib noch Diamond schnell genug verarbeiten könnten. In der Folge läuft der Input-Puffer über und Daten werden verworfen.

4.1.1. Systemlast durch Diamond

Die Tests zur Systemlast wurden lediglich für das Ardupower-Plugin getestet, da die Menge der anfallenden Daten insgesamt deutlich höher ist, als beim LMG450-Plugin.

Abbildung 4.1 zeigt die von Diamond auf dem nehalem5-Knoten erzeugte Last auf CPU und Hauptspeicher. Es waren der ArduPowerCollector und der ScorepHandler aktiviert. Der ScorepHandler hat jedoch keine Daten aufgezeichnet. Gemessen wurde mit dem Python-Tool *psrecord*. Die Messung erfolgte zeitgleich zur Ausführung des PDE-Programms. Die CPU-Last schwankt zwischen 140 und 180%. Es werden konstant etwa 82 MB Hauptspeicher belegt.

Abbildung 4.2 stellt die erzeugte Last dar, wenn zusätzlich das PDE-Programm mit Score-P instrumentalisiert läuft und das Score-P-Diamond-Plugin aktiviert ist. Hier ist gut erkennbar, dass der benötigte Speicher linear ansteigt, etwa um ein halbes Megabyte pro Sekunde. Die CPU-List entspricht die meiste Zeit der aus Abbildung 4.1, lediglich zum Ende der Ausführung des PDE-Programmes ist eine erhöhte Last erkennbar. Diese hat ihre Höhepunkt bei 287%. Dies dürfte mit dem Übertragen der gesammelten Metriken an das Score-P-Plugin zu erklären sein.

4.2. Metric-Plugin

Die Instrumentierung eines Programms mittels Score-P erzeugt selbst eine gewisse Menge an Overhead. Bei Verwendung synchroner Plugins wirkt sich dies auch auf die Laufzeit des Programms aus. Im vorliegenden Fall wird jedoch ein asynchrones Plugin verwendet. Dies bedeutet, dass Start und Stop des Programms verzögert werden. Besonders nach Beendigung des Programms kann durch das Sammeln und transportieren der Metriken einige zusätzliche Zeit vergehen.

Der ScorepHandler wartet nach Empfangen des Stop-Signals vom Score-P-Plugin ab, bis alle gemessenen gemessenen Metriken des Messzeitraumes von der `process`-Methode verarbeitet wurden. Es werden zwar permanent Metriken verarbeitet, sobald diese ankommen. Jedoch kann kein Handler schneller arbeiten als die Collectors. Um auf einen möglichen Datenrückstau zu prüfen, habe ich daher die Laufzeiten der beiden Testläufe verglichen. Die gesamte Ausführungszeit betrug im ersten Fall 3138 Sekunden, im zweiten 3234. Dies ist ein Unterschied fast 100 Sekunden. Das Metric-Plugin erzeugt hier also einen spürbaren Overhead.

5. Mögliche Weiterentwicklungen

Wie in Kapitel 4 gezeigt, könnte unter gewissen Umständen der Hauptspeicher des Rechners, der Diamond betreibt volllaufen. Auf nehalem5 etwa hätte dazu eine Laufzeit von 4 Stunden gereicht. Als Verbesserung könnte es sinnvoll sein, die gesammelten Metriken nicht nur einmal sondern etwa alle 60 Minuten zu übertragen. Möglich wäre dies durch Veränderung der Variable `info.delta_t` (s. [Par17, S. 127]), welche derzeit noch auf den höchsten Wert gesetzt ist. Alternativ ist es auch denkbar, die Daten in Dateien auszulagern. Ob dies jedoch der Verwendung des Swap-Speichers gegenüber Vorteile brächte, ist fragwürdig, da beides teure I/O verursachen würde.

Zusätzliche sinnvolle Features wären außerdem, dem Plugin zusätzlich die Einheit, in der die Daten gemessen werden (z.B. Watt) zu übergeben. Dazu müsste lediglich der Parameter-String, um einen weiteren (optionalen) Parameter erweitert werden.

6. Zusammenfassung

Nach Testen der Plugins ist festzustellen, dass Diamond zwar zumindest die Leistung von pmlib erreicht, was die Leserate an den seriellen Schnittstellen betrifft; jedoch erreicht zumindest das Ardupower-Plugin nicht die maximal mögliche Frequenz von 480Hz, wie sie in [BH15] beschrieben wird. Auch wenn es so aussieht, als läge dies an einer ineffizienten Fehlerkorrektur, kann dies ohne mehr Wissen über das Programm, welches auf dem Ardupower läuft, nicht gesagt werden.

Im Hinblick auf Laufzeiten und Anzahl gleichzeitig laufender Collectors ist bei der Implementation des ScorepHandlers wahrscheinlich noch der Umgang mit den gesammelten Daten zu optimieren, da die jetzige Version den Speicher schnell volllaufen lassen könnte. Für Laufzeiten unter vier Stunden ist sie aber in jedem Fall einsatzbereit.

Dieser Projektbericht behandelt ein Python-Tool zur Leistungsevaluierung und eine für C/C++ entwickelte Infrastruktur zur Performance-Analyse. Python war für mich eine fast komplett neue Sprache, da ich mit dieser bisher nur oberflächlich zu tun hatte. Insbesondere das Konzept der Objektorientierung scheint in Python nicht so ausgereift, wie in anderen Sprachen, was mich bei der Entwicklung des ScorepHandler viel Zeit gekostet hat.

Auch in C musste ich mir einige Konzepte erst aneignen. So habe ich zum Beispiel zuvor noch nicht mit TCP-Kommunikation auf dem Level, wie er zur Programmierung notwendig ist zu tun gehabt.

Anfänglich hatte ich auch Probleme damit, einen funktionierenden Datenaustausch zwischen Python und C herzustellen. Dies erwies sich letzten Endes jedoch als relativ leicht, da Python viele Möglichkeiten bietet, Daten in C-kompatible Strukturen umzuwandeln.

Zusammenfassend kann ich sagen, dass ich - unabhängig vom Ergebnis - durch dieses Projekt Einblick in einige neue Thematiken erhalten und dabei viel lernen konnte.

Literatur

[Knü+12] Andreas Knüpfer u. a. "Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir". In: *Tools for High Performance Computing 2011*. Springer, Berlin, Heidelberg, 2012, S. 79–91. ISBN: 978-3-642-31475-9 978-3-642-31476-6. DOI: 10.1007/978-3-642-31476-6_7. URL: https://link.springer.com/chapter/10.1007/978-3-642-31476-6_7 (besucht am 21. 03. 2018).

[BH15] Manuela Beckert und Janosch Hirsch. *A low cost power measurement device to improve energy efficiency of HPC devices*. 5. Aug. 2015. URL: https://wr.informatik.uni-hamburg.de/_media/teaching/wintersemester_2014_2015/pre-1415-ardupower-report.pdf.

[Par17] Score-P Partners. *Score-P Scalable perormance measurement infrastructure for parallel codes*. 30. Mai 2017. URL: https://silc.zih.tu-dresden.de/scorep-current.pdf (besucht am 15. 11. 2017).

[Par18] Score-P Partners. *Score-P: Introduction*. 2018. URL: https://silc.zih.tu-dresden.de/scorep-current/index.html (besucht am 21. 03. 2018).

[con] python-diamond contributors. *Diamond-Dokumentation*. URL: http://diamond.readthedocs.io/en/latest/.

[Jai+] Gaurav Jain u. a. *Diamond*. URL: https://github.com/python-diamond/Diamond.

[Sco] Score-P. *Vampir 9.4*. URL: https://www.vampir.eu/ (besucht am 21. 03. 2018).

Appendix

A. Hardware-Informationen

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 26
Model name:            Intel(R) Xeon(R) CPU X5560  @ 2.80GHz
Stepping:              5
CPU MHz:               1600.000
CPU max MHz:           2801.0000
CPU min MHz:           1600.0000
BogoMIPS:              5600.04
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K
NUMA node0 CPU(s):     0-3
NUMA node1 CPU(s):     4-7
Flags:                 fpu vme de pse tsc msr pae mce cx8
apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx
fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc
arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc
aperfmperf pni dtes64 monitor ds_cpl vmx est tm2 ssse3 cx16
xtprpdcm dca sse4_1 sse4_2 popcnt lahf_lm retpoline kaiser
tpr_shadow vnmiflexpriority ept vpid dtherm ida
```

Listing A.1: Die Hardware des nehalem5-Knoten

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
```

```
Byte Order:            Little Endian
CPU(s):                24
On-line CPU(s) list:   0-23
Thread(s) per core:    1
Core(s) per socket:    12
Socket(s):             2
NUMA node(s):          4
Vendor ID:             AuthenticAMD
CPU family:            16
Model:                 9
Model name:            AMD Opteron(tm) Processor 6168
Stepping:              1
CPU MHz:               800.000
CPU max MHz:           1900.0000
CPU min MHz:           800.0000
BogoMIPS:              3800.08
Virtualization:        AMD-V
L1d cache:             64K
L1i cache:             64K
L2 cache:              512K
L3 cache:              5118K
NUMA node0 CPU(s):     0-5
NUMA node1 CPU(s):     6-11
NUMA node2 CPU(s):     18-23
NUMA node3 CPU(s):     12-17
Flags:                 fpu vme de pse tsc msr pae mce cx8
apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse
sse2 ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm
3dnowext 3dnow constant_tsc rep_good nopl nonstop_tsc
extd_apicid amd_dcm pni monitor cx16 popcnt lahf_lm
cmp_legacy svm extapic cr8_legacy abm sse4a misalignsse
3dnowprefetch osvw ibs skinit wdt nodeid_msr hw_pstate
retpoline retpoline_amd vmmcall npt lbrv svm_lock nrip_save
pausefilter
```

Listing A.2: Die Hardware der amd-Knoten

List of Listings