Hochschule für Telekommunikation Leipzig
University of Applied Sciences

Hochschule für Telekommunikation Leipzig (FH)

Institut für Telekommunikationsinformatik

Abschlussarbeit zur Erlangung des akademischen Grades

Bachelor of Engineering

Thema: „Untersuchung des Stromverbrauchs von verschiedenen Algorithmen und deren Implementierung auf mobilen Geräten“

Vorgelegt von: Niklas Kluge
Geboren am: 27.06.1996
Geboren in: Wernigerode

Vorgelegt am: 2. November 2020

Erstprüfer: Prof. Dr. Ulf Schemmert

Hochschule für Telekommunikation Leipzig
Gustav-Freytag-Straße 43-45
04277 Leipzig

Zweitprüfer: Philipp Dockhorn, M. Eng.

Hochschule für Telekommunikation Leipzig
Gustav-Freytag-Straße 43-45
04277 Leipzig

Vorwort

An dieser Stelle möchte ich mich bei Herrn Dr. Schemmert für die zuverlässige Betreuung bedanken. Mit seinen Ratschlägen half Herr Schemmert mir bei der Findung zahlreicher Denkansätze, welche in der vorliegenden Arbeit Einzug erhalten haben. Des Weiteren möchte ich meiner Familie danken, die mir trotz familiärer Schicksalsschläge und gesundheitlicher Probleme während des Bearbeitungszeitraums genug Kraft geben konnte, um diese Arbeit rechtzeitig fertigzustellen.

Inhaltsverzeichnis

Abkürzungsverzeichnis

ADB Android-Debug-Bridge 25

ASCII American Standard Code for Information Interchange 12, 24

CPU Central Processing Unit 5, 8, 24, 25, 27, 29, 31, 35, 36, 37, 38, 39, 41

CSS Cascading Style Sheets 12

GB Giga Byte 24

GHz Gigahertz 24, 29, 35

HTML Hypertext Markup Language 12, 25

JVM Java Virtual Machine 8

KB Kilo Byte 12, 21

mA Milliampere 26

mAh Milliamperestunden 24, 26

Mio Millionen 23, 24

ms Millisekunden 23, 27, 29

mV Millivolt 25

OOM Out of Memory 12, 41

ppi Pixel per Inch 24

RFC Request for Comments 12

SDK Software Developoment Kit 20

SMTP Simple Mail Transfer Protocol 12

SoC System on a Chip 24, 29, 35

UI User Interface 8, 13, 21, 22, 23

W Watt 30

Wfi Wireless Local Area Network 25

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

1 Einleitung

1.1 Thematische Einführung

Mobile Gräte wie Smartphones oder Tablets sind aus dem heutigen Alltag der Menschen nicht mehr wegzudenken. Ob zur privaten Unterhaltung beim Spielen aufwendiger 3D-Spiele, beim Streamen von Musik- und Videoinhalten oder als unentbehrliches Werkzeug bei der täglichen Arbeit, mobile Geräte sind nahezu den gesamten Tag über im Einsatz. Auch die Außendiensttechniker der DT Technik GmbH nutzen mobile Applikationen wie die Bestell-App oder die Mess-App zur Verwaltung und Aufstockung ihrer Werkzeuge und Ersatzteile beziehungsweise zur Untersuchung und Konfiguration von Routern. Hohe Akkulaufzeiten sind für diese Art der Benutzung eine Voraussetzung und stellen Smartphone Hersteller sowie Softwareentwickler vor die Herausforderung, energieeffiziente Lösungen zu finden. Dabei werden verschiedene Ansätze verfolgt. So versuchen die Hersteller energiesparende Prozessoren und Displays zu entwickeln und die Hardwarenutzung zu optimieren. Dieses Bestreben steht jedoch im Konflikt mit den Wünschen der Kunden, welche schnellere Mehrkernprozessoren und größere Displays für die neuen Geräte erwarten. Dieser Trend ist auch in der Marktentwicklung der letzten zehn Jahre zu beobachten. So entsprach die durchschnittliche Displaygröße 2009 c.a. 3,2 Zoll. Acht Jahre später waren bereits Displays mit 5,5 Zoll üblich [4]. Auf der anderen Seite versuchen Anwendungsentwickler durch Softwareoptimierung ihre Applikationen ressourcensparender zu gestalten. Dabei können Prozesse wie zum Beispiel größere Downloads für Datenbankupdates als Service oder mithilfe des neuen Android Work Managers als Hintergrundprozess implementiert werden und in Abhängigkeit vom aktuellen Ladestand des Akkus auf günstigere Zeitpunkte verschoben werden [5]. Auch das gewissenhafte Umgehen mit Wake Locks und das Festlegen der Standby Phasen der eigenen Applikation sind wichtige Stellschrauben, über welche ein Anwendungsentwickler energiesparende Anpassungen justieren kann [6]. Dies sind nur einige Beispiele der möglichen Optionen für Energieoptimierungen auf mobilen Geräten. Das Problem der Energieeffizienz auf mobilen Geräten bietet ein breites Feld an Forschungspotential und wird auch zukünftig eine zentrale Rolle in der Geräteherstellung und App-Entwicklung spielen.

1.2 Thematische Abgrenzung und Zielstellung dieser Arbeit

In dieser Arbeit werden verschiedene Implementierungsansätze für Algorithmen und Berechnungen betrachtet und auf deren Einfluss hinsichtlich des Energieverbrauchs mobiler Geräte untersucht. Der Schwerpunkt liegt hierbei auf der Betrachtung von parallelen Berechnungen mithilfe von Multithreading. Dabei wird der Zusammenhang zwischen der Anzahl der parallel laufenden Threads, der Laufzeitveränderung und dem damit einhergehenden Energieverbrauch gemessen. Ziel dieser Untersuchung ist es, herauszufinden, ob es einen Zusammenhang zwischen den drei genannten Parametern gibt und gegebenenfalls eine Empfehlung für die Implementierung von Multithreading zu konstruieren, welche einen sinnvollen Kompromiss aus Laufzeit und Energieverbrauch bereitstellt. Weiterhin wird der Unterschied zwischen rekursiven und iterativen Implementierungen hinsichtlich des Stromverbrauchs betrachtet. Auch hier ist das Ziel, die ressourcenschonendste Variante zu ermitteln.

1.3 Struktureller Aufbau und Vorgehensweise der Untersuchung

Das anschließende Kapitel „Untersuchte Konzepte und Implementierungen“ beschreibt die theoretischen Aspekte von paralleler Programmierung und deren Einfluss auf den Energieverbrauch sowie die für diese Arbeit relevanten Gesetzmäßigkeiten und Berechnungsformeln des Multiprocessings. Außerdem werden die zur Untersuchung entwickelten Implementierungen vorgestellt, welche in der „EnergyEfficience“ Applikation angewandt werden. Für die Umsetzung einer parallelen Ausführung wurde ein Base64 Encoder implementiert. Dieser ist nur ein Mittel zum Zweck und könnte problemlos durch andere parallelisierbare Algorithmen ersetzt werden. Des Weiteren werden auch die Unterschiede von rekursiven und iterativen Algorithmen beleuchtet und gegenübergestellt. Diese Gegenüberstellung bildet den zweiten große Untersuchungsgegenstand dieser Arbeit. Weil der Mergesort Algorithmus aufgrund seiner Vielseitigkeit perfekt für den Vergleich dieser beiden Implementierungsstrategien geeignet ist, werden in diesem Kapitel die verwendeten Mergesort Varianten inklusive ihrer Implementierung vorgestellt.

Im darauffolgenden Kapitel drei „Verwendete Geräte und Tools“ werden alle Programme und Geräte samt deren Spezifikationen vorgestellt, die im Rahmen dieser Untersuchung genutzt wurden. Es wird eine kurze Einführung in die Verwendung des Programms Battery Historian geben. Weiterhin wird die eigens für diese Arbeit entwickelte App vorgestellt. Anschließend wird die Messmethode mithilfe dieser beiden Tools beschrieben.

In Kapitel vier „Energieeffizientes Multithreading“ werden die Ergebnisse der Messungen des parallelen Base64 Encoders dargestellt und ausgewertet. Der Fokus liegt hierbei auf dem Zusammenhang zwischen verwendeter Thread-Anzahl und dem Energieverbrauch.

Daraufhin untersucht Kapitel fünf „Rekursive und Iterative Verfahren im Vergleich“ den Einfluss von iterativer beziehungsweise rekursiver Implementierung auf den Energieverbrauch. Hierfür wird die bereits erwähnte Mergesort Implementierung der „EnergyEfficience“ Applikation genutzt. Außerdem werden die Vorzüge des Fork-Join Thread Pools bei paralleler Rekursion durch Messungen untersucht..

Im abschließenden Kapitel sechs „Auswertung“ ist eine Zusammenfassung der gewonnen Erkenntnisse zu finden. Außerdem werden in einem Ausblick Ansätze für weitere Untersuchungen auf diesem Gebiet formuliert.

2 Untersuchte Konzepte und Implementierungen

2.1 Theoretische Aspekte in die Parallelität

Das Ziel hinter der Parallelisierung von Aufgaben ist die Beschleunigung der Laufzeit bei der Abarbeitung von Programmabläufen und die Minimierung der Wartezeiten des Prozessors. Solche Wartezeiten können entstehen, wenn während der Programmausführung Benutzereingaben nötig sind, bevor die Ausführung fortgesetzt werden kann oder wenn neue Daten aus dem vergleichsweise langsamen Hauptspeicher nachgeladen werden müssen, falls der prozessoreigene Cache nicht groß genug ist, um alle nötigen Daten für die aktuelle Ausführung auf einmal zu laden[7, S. 1135]. Ohne Parallelität würden moderne Softwareanwendungen jeglicher Art nahezu unnutzbar werden. Einfache Vorgänge wie das Laden von Benutzerdaten aus einer lokalen Datenbank oder das Downloaden von Bildern aus dem Netz, würden ohne Parallelität beispielsweise zum Einfrieren der Benutzeroberfläche führen, da bei sequentiellen Programmabläufen alle Aufgaben strikt hintereinander ausgeführt werden müssen. Android selbst wäre ohne Parallelität nicht umsetzbar, da Androids Architektur Multithreading und damit Parallelität voraussetzt.

Für die Realisierung von Parallelität haben sich mit der Evolution der Prozessortechnologie verschieden Ansätze und Techniken entwickelt. Jede dieser Techniken ist bis heute relevant und glänzt in unterschiedlichen Anwendungsfällen.

Pipelining

Beim Pipelining wird die Ausführung von Befehlen in verschiedene Phasen aufgeteilt, die jeweils durch eine eigene Ausführungseinheit bearbeitet werden. Sobald ein Befehl die aktuelle Phase abgeschlossen hat und zur nächsten Phase springt, kann bereits mit der Bearbeitung des nächsten Befehls in der frei gewordenen Phase begonnen werden. In Abbildung 2.1 ist ein Beispiel einer 5-Phasen Pipeline veranschaulicht. Die Bearbeitung jeder Phase dauert im Optimalfall einen Taktzyklus, da andernfalls die Pipeline bei der Bearbeitung der nachfolgenden Befehle in dieser Phase geblockt wird. Dies ist zum Beispiel in Abbildung 2.1 bei Befehl drei während der Execute Phase der Fall. Ein großer Nachteil von Pipelining tritt bei häufigen Programmsprüngen auf, da bei jedem Sprung die komplette Pipeline geleert werden muss und alle Phasen, die bis dorthin vollendet wurden, verworfen werden und umsonst bearbeitet wurden. Dieser Umstand ist besonders bei größeren Pipelines kritisch, da die einzelnen Befehle für längere Zeiträume in der Pipeline gehalten werden [8].

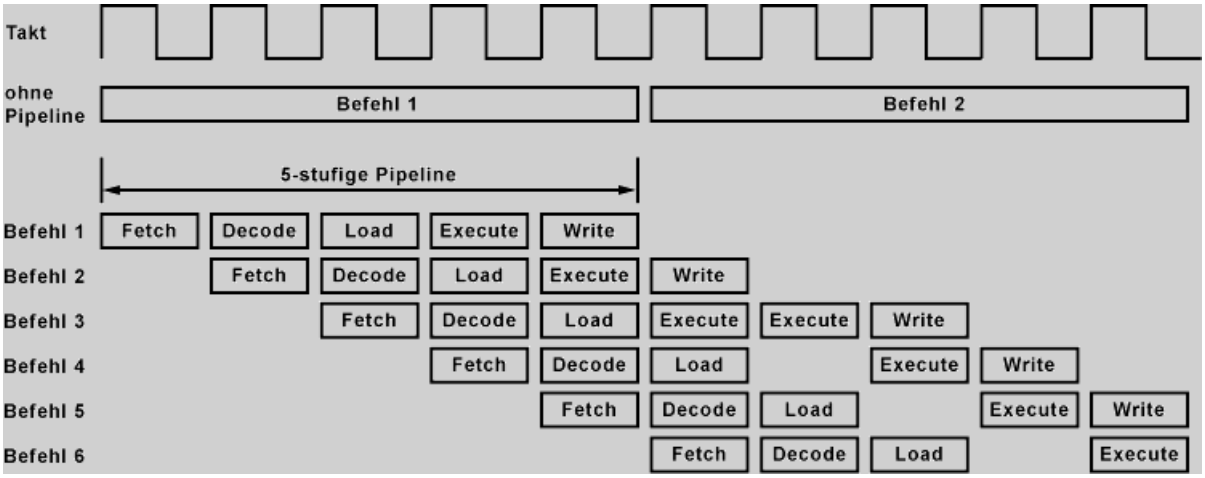

Abbildung 2.1: Beispiel für eine 5-Phasen-Pipeline (Quelle: [8])

simultanes Multithreading

Ein Thread ist ein sequenzieller Ausführungspfaden innerhalb eines Prozesses, der ausgeführt wird. Bei Rechnern mit nur einem physischen Rechenkern kann zu einem Zeitpunkt immer nur ein Thread eines Prozesses gleichzeitig ausgeführt werden. Falls diese Ausführung durch Ereignisse wie Speicherzugriffe oder Nutzereingaben unterbrochen wird, blockiert der zugewiesene Thread die Central Processing Unit (CPU) vollständig, sofern kein Multithreading betrieben wird. Durch sogenanntes simultanes Multithreading wird die verfügbare CPU Rechenzeit auf mehrere Prozesse und Threads aufgeteilt. Die CPU kann dadurch zwischen den Threads hin und her springen und so längere Wartezeiten verhindern. Dies steigert die Effizienz der CPU hinsichtlich der Laufzeit sowie des Energieverbrauchs [9, S. 877]. Diese Technik ermöglicht auch auf Systemen mit nur einem physischen Rechenkern den Eindruck von Parallelität und führt zu besser Nutzbarkeit der Anwendungen.

Hyperthreading

Die Technologie Hyperthreading wurde von Intel mit den Prozessoren Pentium 3, Pentium 4 und Xeon eingeführt. Hierbei wird der Durchsatz von multithreaded Anwendungen im Multitasking erhöht, indem die Auslastung der On-Chip-Ressourcen erhöht wird, die in der Intel-NetBurst-Mikroarchitektur verfügbar sind. Ein typischer Thread belastet nur etwa 35 % der NetBurst-Ausführungsressourcen. Hyperthreading erhöht die Auslastung durch notwendige Logik und Ressourcen, die der CPU hinzugefügt werden. Für die Aufteilung der reinkommenden Daten auf den freien Raum sorgen somit zwei logische Prozessoren, die vom Betriebssystem mittels klassischer Multiprocessing-Verfahren verwaltet werden [7, S. 1138]. Somit stehen dem Rechner, trotz physischer Single-Core CPU, daher zwei logische Rechenkerne zur Verfügung. Dies bietet die Möglichkeit Speicherwartezeit zu vermeiden, die ohne diese Technik die gesamte CPU blockieren würde. Wenn der erste

Thread im Wartezustand ist, kann der Prozessor mithilfe des zweiten logischen Kerns in der Programmausführung der anderen Prozesse fortfahren.

Multicore-Prozessor

Bei Multicore Systemen handelt es sich um Prozessoren mit mehreren physischen Rechenkernen. Diese können voneinander unabhängig arbeiten und ermöglichen echte Parallelität. Alle Rechenkerne nutzen zusätzlich die schon genannten Techniken, um möglichst effiziente Ergebnisse zu erhalten. Der Vorteil liegt hierbei nicht nur in der Steigerung der Geschwindigkeit. Mehrkernprozessoren können wesentlich geringer getaktet werden als Single-Core Prozessoren und ermöglichen trotzdem erhöhte Laufzeitgeschwindigkeit bei geringerer Leistungsaufnahme und Wärmeentwicklung [10]. Es könnte die Annahme getroffen werden, dass mit steigender Rechenkernzahl auch die Rechenleistung äquivalent ansteigt. Sodass die neue Laufzeit im Multi-Core-Betrieb gleich der ursprünglichen Laufzeit mit einem Kern geteilt durch die Anzahl der Rechenkerne ist. In der Realität ist dies jedoch nicht umsetzbar, da verschiedene Faktoren diese Annahme begrenzen. Zunächst einmal wächst mit steigender Anzahl an Rechenkernen auch der Aufwand der Verwaltung durch das Betriebssystem. Außerdem ist es nicht einfach den Zugriff auf geteilte Speicherressourcen durch mehrere Rechenkerne zu synchronisieren. Die Laufzeit für parallele Prozesse setzt sich grob aus folgenden Anteilen zusammen.

Rechenzeit Zeit für die Durchführung von Berechnungen unter Verwendung von Daten im lokalen Speicher der einzelnen Prozessoren.

Kommunikationszeit Zeit für den Austausch von Daten zwischen Prozessoren.

Wartezeit Z.B. aufgrund ungleicher Verteilung von Last zwischen den Rechenkernen, Datenabhängigkeiten im Algorithmus oder Ein- und Ausgabe.

Synchronisationszeit Zeit für die Synchronisation beteiligter Prozesse und von Ressourcenzugriffen.

Platzierungszeit Zeit für die Allokation der Tasks auf die einzelnen Prozessoren, sowie eine mögliche dynamische Lastverteilung zur Programmlaufzeit.

Startzeit Zeit zum Starten der parallelen Threads auf allen Rechenkernen.

[11, S. 313]

Viele Anwendungen können nur für kleine Bestandteile ihrer Ausführung von den Vorteilen eines Multi-core Systems profitieren, da die meisten Anwendungsfälle durch Nutzeraktionen bestimmt sind. Bei Programmabläufen mit strikt aufeinanderfolgenden Abhängigkeiten ist Parallelität ohnehin nicht möglich und resultiert in traditionell sequentielle Abläufe.

Das Admahl'sche Gesetz beschreibt genau diese Grenze. So ist der Beschleunigungsfaktor, auch Speedup genannt, mit zusätzlichen Rechenkernen durch sequentielle Anteile eines Problems begrenzt [11, S. 314]. Das Admahl'sche Gesetz, benannt nach Gene Admahl, dient zur Vorhersage der maximal zu erwartenden Beschleunigung eines Algorithmus durch parallele Ausführung und wird wie folgt beschrieben.

Speedup Beschleunigungsfaktor der Rechenzeit durch p Kerne $S_p(n)$

sequentielle Laufzeit Laufzeit bei Ausführung mit einem Rechenkern $T_{seq}(n)$

Anzahl der Rechenkerne Anzahl der an der parallelen Ausführung beteiligten Prozessorkernen p

sequentieller Anteil Anteil des Problems, welcher ausschließlich sequentiell ausführbar ist f. Es gilt $0 \leq f \leq 1$ wobei $f = 1$ bedeuteten würde, dass das gesamte Problem, also 100 % des Problems, sequentiell ausgeführt werden muss.

paralleler Anteil Anteil des Problems, welcher parallelisierbar ist $(1 - f)$

parallele Laufzeit Laufzeit bei paralleler Ausführung mit p Rechenkernen für ein Problem der Größe n $T_p(n)$

Problemgröße Die Größe der Berechnung des Algorithmus n

$$S_p(n) = \frac{T_{seq}(n)}{T_p(n)} = \frac{T_{seq}(n)}{f * T_{seq}(n) + \frac{(1-f)*T_{seq}}{p}} \tag{2.1}$$

[11, S. 317]

Der Speedup $S_p(n)$ aus Gleichung 2.1 wird im Rahmen dieser Arbeit für die Ermittlung der Effizienz es Multithreadings benötigt, welche wie folgt beschrieben ist.

$$E_p(n) = \frac{S_p(n)}{p} \tag{2.2}$$

[11, S. 316]

Die Effizienz (Gleichung 2.2) eines parallelen Programms gibt die relative Verbesserung des Speedups $S_p(n)$ bezüglich der Anzahl p der Prozessorkerne an, die bei der Ausführung genutzt wurden. Diese Größe wird im Verlauf der Untersuchung der folgenden Kapitel für den Vergleich von Laufzeiteffizienz und Energieeffizienz in Abhängigkeit der Thread Anzahl verwendet.

2.2 Thread Pool Implementierung in Android

Die „EnergyEfficience“ App wurde vollständig in Java entwickelt. An dieser Stelle sei erwähnt, dass mit der immer stärkeren Kotlin-Ausrichtung des Android Frameworks neue Technologien hinsichtlich Multithreading und Synchronisation an Beliebtheit gewinnen. Kotlin bietet neben den traditionellen Java Techniken wie Threading, Callbacks und Futures auch sogenannte Kotilin Corountines. Durch Corountines können Ausführungen von längeren Funktionen beliebig pausiert werden, um anschließend mit anderen Aufgaben fortzufahren. Sobald die CPU wieder freie Ressourcen hat, springt der Programmcounter an die Stelle der pausierten Funktion zurück und nimmt die Ausführung wieder auf. Dies verhindert blockierende Berechnungen, welche beispielsweise zur Einfrierung der User Interface (UI) führen können. Hierbei reicht es das Schlüsselwort „suspend“ bei der Deklaration der Funktion anzugeben. Verglichen mit der Implementierung von Callbacks ist diese Herangehensweise sehr einfach und schnell umzusetzen [12]. In Java wird Nebenläufigkeit traditionell mit der java.lang.Thread Klasse realisiert. Im Konstruktor des Thread-Objekts wird eine Referenz auf ein Objekt vom Typ Runnable verlangt, welches den parallel auszuführenden Programmcode enthält. Das Runnable-Objekt implementiert diesen Code in der vom Rnnable-Interface definierten Methode *run()*. Bei der Nutzung ist es wichtig, den Code nicht einfach durch Aufrufen dieser *run()*-Methode zu starten. Dies würde zu einer normalen sequentiellen Ausführung führen. Um eine parallele Ausführung zu erreichen muss die *start()*-Methode des entsprechenden Thread-Objekts aufgerufen werden. Dadurch wird für diesen Thread eine separate Ablaufumgebung mit den nötigen Systemressourcen erstellt. Nun wird automatisch die interne *run()*-Methode des Thread-Objekts mit der Implementierung des Runnable-Objekts innerhalb der separaten Ablaufumgebung ausgeführt. Sobald die *run()*-Methode terminiert, wird der Thread automatisch beendet und seine Systemressourcen werden freigegeben. Es ist außerdem möglich, eine eigene Klasse zu erstellen, die vom Typ *Thread* erbt und den auszuführenden Code direkt in der eigenen *run()*-Methode implementiert. Diese Variante ist jedoch weniger flexibel als das Benutzen von separaten Runnable-Objekten, da für jedes Problem eine erweiterte Thread-Klasse mit einer eigenen Variante der *run()*-Methode erstellt werden müsste. Bei der Ausführung von mehreren Threads zur gleichen Zeit, sind die genauen Terminierungszeitpunkte der einzelnen Threads selbst bei identischen Aufgaben nicht vorhersehbar, da der Kontextwechsel zwischen parallellaufenden Threads vom Scheduler des Betriebssystems organisiert wird und daher nicht nachvollziehbar ist [13]. Generell muss erwähnt werden, dass die Java Virtual Machine (JVM) die Thread-Verwaltung direkt auf das Betriebssystem abbildet. Das bedeutet, dass die eigentliche Thread-Verwaltung auf technischer Ebene inklusive der einhergehenden Ressourcenverwaltung nicht direkt durch die Java Implementierung beeinflusst werden kann, sondern allein durch das zugrunde liegende Betriebssystem bestimmt wird [13]. Sprachen wie C++ oder C sind für solche Aufgaben besser geeignet und ermöglichen einen maschinennäheren Zugriff und dadurch in der Regel performantere Lösungen.

Das Prinzip der Erstellung von Threads und der Kapselung des Ausführungscodes durch Runnables ist zwar die Grundlage für Multithreading in Java, reicht für die effektive Umsetzung von Threadpools für das dynamische Verwalten von vielen Threads alleine

nicht aus. Die feste Bindung zwischen dem Ausführungs-Thread und dem Runnable-Objekt ist zu unflexible um ein effizientes und benutzerfreundliches Thread-Management zu realisieren. Schon bei der Erzeugung eines Threads muss das Runnable-Objekt im Thread-Konstruktor übergeben werden und kann im Nachhinein nicht mehr geändert werden. Des Weiteren ist es nicht möglich die *start()*-Methode eines Threads mehrmals hintereinander aufzurufen, da dies zu einer Exeption führen würde falls der Thread bereits läuft. Weil das Thread-Objekt nach Abarbeitung des Runnables beendet und verworfen wird, ist es außerdem nicht möglich, Threads wiederzuverwenden. Das bedeutet, dass für jede Abarbeitung eines Raunnables ein neues Thread-Objekt erstellt werden muss. Selbst wenn die abzuarbeitende Aufgabe identisch ist, muss ein neues Objekt mit dem gleichen Runnable instanziiert werden. Das ständige erstellen und verwerfen von Thread-Objekten führt zu einer permanenten Belastung des Garbage Collectors und kann zu Performance-Verlust führen. Um diese ungünstigen Nebenerscheinungen der starren Kopplung von Thread und Runnable-Objekt zu umgehen, bietet Java die Schnittstelle *Executor*, welche die Ausführung des Runnable-Programmcodes von der Initialisierung der Threads trennt. Dieses Interface schreibt die Methode *execute(Runnable command)* vor, mit der beliebig austauschbare Runnables auf einem *Executer* ausgeführt werden können [14]. Für die Umsetzung des parallelen Base64-Encoders wurde ein Threadpool Manager mithilfe der *ThreadPoolExecutor* Klasse umgesetzt. *ThreadPoolExecutor* ist eine Java eigene Implementierung der Schnittstelle *Executor*, um eine Sammlung von Threads aufzubauen, welche beliebig viele Aufgaben (Runnables) koordiniert abarbeiten kann. Dabei werden den Threads ohne Aufgabe neue Runnables aus einer Aufgaben-Queue dynamisch zugeordnet [14]. Der Quellcode 2.1 zeigt die Implementierung des *CustomThreadPoolManagers* in der „EnergyEfficience“ App. Im Konstruktor wird eine Instanz des *ThreadPoolExecutors* initialisiert. Mit *NUMBER_OF_CORES* wird die maximale Anzahl der gleichzeitigen Worker-Threads angegeben. Standardmäßig wird dieser Wert durch Aufruf der Methoden *Runtime.getRuntime().availableProcessors()* mit der Anzahl der physischen Rechenkerne des Gerätes gleichgesetzt. Da die Untersuchung jedoch unter Anderem darauf abzielt, das Verhalten des Energieverbrauchs bei variabler Anzahl von Worker-Threads darzustellen, gibt es eine *setNumberOfCores(int numberOfCores)*-Methode, um diesen Wert während der Laufzeit anzupassen. Die *KEEP_ALIVE_TIME* Variable legt fest, wie lange ein Thread ohne Aufgaben am Leben erhalten wird, um auf neue Runnables zur Ausführung zu warten. Bekommt der Thread innerhalb dieser Zeit keine neue Aufgabe zugewiesen, so werden seine Ressourcen für andere Prozesse freigegeben. Über eine *BlockingQueue* (*mTaskQueue*) werden die vom Thread-Pool abzuarbeitenden Runnables zwischengespeichert. Dies ist eine spezielle Datenstruktur, die Operationen unterstützt, welche nicht sofort ausgeführt werden können. So könnte es zum Beispiel vorkommen, dass ein Thread aus dem Thread-Pool mit der letzten Ausführung fertig ist und nun nach einer weiteren Aufgabe fordert, ohne dass neue Runnables in der *mTaskQueue* vorhanden sind. Die *BlockingQueue* bietet für solche Fälle die *poll(time, unit)*-Methode an. Dadurch wird der aufrufende Thread für den angegebenen Zeitraum geblockt, in welchem er auf neue Aufgaben in Form von Runnables wartet [15]. Dieses Zeitintervall wird in Quellcode 2.1 durch die Variablen *KEEP_ALIVE_TIME* und *KEEP_ALIVE_TIME_UNIT* festgelegt. Da es mit normalen Runnables nur schwer möglich ist, Ergebnisse einer asynchronen Methodenausführung in Form von normalen Rückgabewerten abzugreifen, wurden die Callables eingeführt. Callable-Objekte funktionieren ähnlich wie Runnable-Objekte und können daher auch wie

Runnables behandelt werden. Allerdings implementieren sie statt der *run()*-Methode die *call()*-Methode. Wenn ein Callable-Objekt durch *submit(Callable c)* der Aufgabenqueue (*mTaskQueue*) hinzugefügt wird, dann liefert dieser Aufruf ein Objekt vom Typ *Future*. Dieses *Future-Objekt* dient als Platzhalter für das zukünftige Ergebnis des asynchronen Aufrufs. In der Methode *addCallable(Callable callable)* werden neue Aufgaben in Form von Callables der *mTaskQueue* hinzugefügt und gleichzeitig Future-Objekte für jedes dieser Callalbes in die *mTaskQueue* geschrieben. Mit dieser Struktur ist ein eleganter Zugriff auf die Ergebnisse der asynchronen Thread-Ausführungen möglich, ohne dabei die eigentliche Routine der Threads zu stören. Um zu verhindern, dass mehrere Instanzen dieses Threadpools gleichzeitig existieren können, wurde diese Klasse als sogenanntes Singleton implementiert. Singleton ist die Bezeichnung für ein Design Pattern aus der Softwareentwicklung, bei dem sichergestellt wird, dass von einer Klasse nur eine einzige Instanz existiert. Diese Instanz wird global definiert, sodass es an jeder Stelle im Projekt verfügbar ist. Zur Umsetzung wurde der Konstruktor als *private* definiert, um zu verhindern, dass weitere Instanzen außerhalb des Klassenkontextes erstellt werden können. In Zeile 14 von Quellcode 2.1 wird einmalig eine statische Instanz von *CustomThreadPoolManager* definiert, welche durch die statische Klassenmethode *getInstance()* abrufbar ist. Die *BackgroundThreadFactory* wird benötigt um die Erstellung und Zuweisung von neuen Threads mit Runnable- beziehungsweise Callable-Objekten zu automatisieren.

Quellcode 2.1: der CustomThreadManager aus der EnergyEfficience App

```
1  public class CustomThreadPoolManager {
2
3      private  static int NUMBER_OF_CORES = Runtime.getRuntime().
           availableProcessors();
4      private static final int KEEP_ALIVE_TIME = 1;
5      private  static final TimeUnit KEEP_ALIVE_TIME_UNIT;
6      private Handler mainThreadHandler = HandlerCompat.createAsync(Looper.
           getMainLooper());
7      private final ExecutorService mExecuterService;
8      private final BlockingQueue<Runnable> mTaskQueue;
9      private List<Future> mRunningTaskList;
10     private static CustomThreadPoolManager singleInstance = null;
11
12     static{
13         KEEP_ALIVE_TIME_UNIT = TimeUnit.SECONDS;
14         singleInstance = new CustomThreadPoolManager();
15     }
16     private CustomThreadPoolManager(){
17         mTaskQueue = new LinkedBlockingQueue<Runnable>();
18         mRunningTaskList = new ArrayList<>();
19         mExecuterService = new ThreadPoolExecutor(
20                 NUMBER_OF_CORES,
21                 NUMBER_OF_CORES,
22                 KEEP_ALIVE_TIME,
23                 KEEP_ALIVE_TIME_UNIT,
24                 mTaskQueue,
25                 new BackgroundThreadFactory());
26     }
27     public static void setNumberOfCores(int numberOfCores) {
28         if(numberOfCores > 0){
```

```
29              NUMBER_OF_CORES = numberOfCores;
30              singleInstance = new CustomThreadPoolManager();
31          }
32      }
33      public void addCallable(Callable callable){
34          Future future = mExecuterService.submit(callable);
35          mRunningTaskList.add(future);
36      }
37      public Handler getMainThreadHandler(){
38          return this.mainThreadHandler;
39      }
40      public static CustomThreadPoolManager getInstance(){
41          return singleInstance;
42      }
43      public int getNumberOfCores(){
44          return NUMBER_OF_CORES;
45      }
46      public void cancelAllTasks() {
47          synchronized (this) {
48              mTaskQueue.clear();
49              for (Future task : mRunningTaskList) {
50                  if (!task.isDone()) {
51                      task.cancel(true);
52                  }
53              }
54              mRunningTaskList.clear();
55          }
56      }
57      private static class BackgroundThreadFactory implements ThreadFactory {
58          private static int sTag = 1;
59          @Override
60          public Thread newThread(Runnable runnable) {
61              Thread thread = new Thread(runnable);
62              thread.setName("CustomThread" + sTag);
63              sTag++;
64              thread.setPriority(THREAD_PRIORITY_BACKGROUND);
65              thread.setUncaughtExceptionHandler(new Thread.
                    UncaughtExceptionHandler() {
66                  @Override
67                  public void uncaughtException(Thread thread, Throwable ex)
                        {
68                      Log.e("ThreadFactory", thread.getName() + " encountered
                            an error: " + ex.getMessage());
69                  }
70              });
71              return thread;
72          }
73      }
74  }
```

Die Flexibilität dieses Threadpools, der in der variablen Festlegung der Arbeiter Threads liegt, bildet die Grundlage der folgenden Messungen. Ziel ist es, herauszufinden, welche Anzahl von Threads am energieeffizientesten ist und welcher Zusammenhang mit der Laufzeit besteht.

2.3 Ein paralleler Base64 Encoder

Für die Untersuchung der parallelen Ausführung über beliebig viele Threads wurde die Base64-Kodierung gewählt. Hierbei werden 8-Bit-Binärdateien in eine 7-Bit-American Standard Code for Information Interchange (ASCII)-Textrepräsentation umgewandelt. So können binäre Dateiformate wie beispielsweise Bilddateien in ASCII-Textformate umgewandelt werden und direkt in Hypertext Markup Language (HTML)- oder Cascading Style Sheets (CSS)-Dateien eingebunden werden. Auch für das Verschicken von E-Mails mit Anhang wird die Base64-Kodierung noch häufig genutzt, da das Simple Mail Transfer Protocol (SMTP) ursprünglich nur in der Lage war, sieben-Bit-ASCII-Texte zu transportieren. Ohne die Base64-Kodierung wäre es daher Schwierig, empfangene Texte aus anderen Regionen mit verschiedenen Zeichensätzen vernünftig darzustellen. Um Komplikationen durch die Übertragung von länderspezifischen Sonderzeichen oder Steuerzeichen zu verhindern, verwendet Base64 ausschließlich Buchstaben des nicht erweiterten Alphabets (A bis Z, a bis z), die Ziffern von Eins bis Null sowie die Zeichen „/“ und „+“ für die Kodierung. Diese Zeichen sind standardisiert und kommen in den meisten länderspezifischen beziehungsweise betriebssystemspezifischen Zeichensätzen vor. Bei der Kodierung von ASCII-Dateien wird wie folgt vorgegangen. Ein ASCII-Zeichen wird standardmäßig durch acht Bits repräsentiert. Eine ASCII-Datei wird zunächst in Binärcode umgewandelt, welcher eine Aneinanderreihung der 8-Bit-Pakete der einzelnen ASCII-Zeichen ist. Nun wird diese Bytefolge in Abschnitte von jeweils sechs Bits aufgeteilt. Diese 6-Bit-Abschnitte werden entsprechend ihres umgerechneten Zahlenwertes mithilfe einer einfachen Zeichentabelle in Base64-Code umgewandelt. Mit sechs Bits lassen sich $2^6 = 64$ verschieden Zustände darstellen, daher besteht die Base64-Umwandlungstabelle aus 64 verschiedenen Zeichen. Diese Tabelle und die Spezifikation des Base-64-Standards sind im Request for Comments (RFC)-4648 [16] und RFC-2045 festgelegt [17]. Die Komplexität der Base64-Kodierung hängt von der Größe der zu kodierenden Daten ab, also von der Anzahl der Bits der entsprechenden Binärcoderepräsentation. Dabei verhält sich die Laufzeitkomplexität der Kodierung proportional zur Anzahl der Bits. Die Umwandlungsoperation ist mithilfe einer iterativen Schleife umsetzbar. Daher ist die Base64-Kodierung in die Komplexitätsklasse $O(n)$ der O-Notation einzuordnen. Die O-Notation zur Beschreibung der Komplexität von Algorithmen wurde von Donald E. Knuth in seinem Werk „The art of computer programming: Fundamental algorithms“ vorgestellt [18, S. 107].

Im folgenden Codeausschnitt aus der „EnergyEfficience“ Applikation, ist zu sehen, dass die Base64-Kodierung innerhalb einer *Callable*-Klasse implementiert ist. Dadurch können mithilfe des Threadpools beliebig viele Kodierungen parallel durchgeführt werden, indem neue *Base64EncodeCallable* Objekte in die Aufgaben-Queue des Threadpools hinzugefügt werden. Über den Konstruktor kann die Größe des zu kodierenden Textausdrucks festgelegt werden, welcher durch einen *StringBuilder* in Zeile 27 generiert wird. Damit es nicht zu Abstürzen aufgrund von Out of Memory (OOM) Fehlern kommt, wurde die maximale Größe des Strings auf 10000 Kilo Byte (KB) beschränkt. In Zeile 12 wird dieser String in seine Byte-Repräsentation umgewandelt, damit dieser durch den Base64-Encoder kodiert werden kann. Um die Ergebnisse der Kodierung an anderen Stellen der Applikation nutzbar zu machen, muss eine Kommunikation zwischen dem Thread, der ein *Base64Callable*

bearbeitet, und dem UI-Thread hergestellt werden. Androids UI-Thread besitzt eine sogenannte *Message Queue*, welche alle Aktionen, die auf dem UI-Thread ausgeführt werden sollen enthält. Dies können Interaktionen mit der UI oder einfache Funktionsaufrufe sein. Diese *Message Queue* wird ständig von einem assoziierten *Looper*-Objekt durchlaufen. Es handelt sich hierbei im Grunde genommen um eine Endlosschleife, welche alle Objekte der *Message Queue* durchläuft und die Aktion auswählt, die auf dem UI-Thread als Nächstes bearbeitet werden soll. Die Reihenfolge der Durchführung dieser Aktionen wird nach der Priorität und einer Zeitmarke der einzelnen Aktionen festgelegt. Nun ist es möglich durch einen *Handler*, der mit dieser *Message Queue* gekoppelt ist, Einfluss auf die Queue des UI-Threads zu nehmen und neue Aufgabenpakete in diese Queue einzuschieben [19]. Dieses Prinzip wird in Zeile 27 der *notifyResults*-Methode genutzt. Mit *resultHanlder.post()* wird ein neues Runnable in die *Message Queue* des UI-Threads eingefügt. Über ein Callback wird so die Methode *onComplete(result)* auf dem UI-Thread ausgeführt. Somit kann das Ergebnis der Kodierung in den Kontext des UI-Threads zurückgegeben werden. Außerdem findet über diese Callback-Methode auch die Synchronisation der parallelen Threads sowie die Zeitmessung der Ausführung statt.

Quellcode 2.2: Base64-Callable aus derEnergyEfficience App

```
1  public class Base64EncodeCallable implements Callable {
2      private int msgSize = 0; //in KB
3      private Base64Callback callback;
4      private Handler resultHandler;
5      public Base64EncodeCallable(int msgSize, Base64Callback callback,
           Handler resultHandler){
6          this.callback = callback;
7          this.resultHandler = resultHandler;
8          this.msgSize = msgSize;
9      }
10     @Override
11     public Object call() throws Exception {
12         byte[] buffer = createBigString(msgSize).getBytes();
13         notifyResult(Base64.getEncoder().encodeToString(buffer), callback,
               resultHandler);
14         return null;
15     }
16     private void notifyResult(final String result, final Base64Callback
           callback, final Handler resultHandler){
17         resultHandler.post(new Runnable() {
18             @Override
19             public void run() {
20                 callback.onComplete(result);
21             }
22         });
23     }
24     public String createBigString(int msgSize){
25         msgSize = msgSize / 2;
26         msgSize = msgSize* 1024;
27         StringBuilder sb = new StringBuilder(msgSize);
28         for(int i = 0; i< msgSize; i++){
29             sb.append('a');
30         }
31         return sb.toString();
```

```
32      }
33  }
```

2.4 Iteration und Rekursion Vorbetrachtung

Als Iteration wird ein mehrmaliges Ausführen einer Aktion oder Berechnung bezeichnet, welche durch eine Abbruchbedingung oder maximale Anzahl an Ausführungen begrenzt ist. In der Programmierung werden Iterationen hauptsächlich mit Kontrollstrukturen wie Schliefen realisiert [20].

Auch die Rekursion beschreibt eine wiederholte Ausführung, doch werden hierbei keine Zählschleifen verwendet. Rekursionen werden durch Funktionen definiert, die sich selbst solange aufrufen, bis eine Abbruchbedingung erfüllt ist. Es ist allgemein bekannt, dass die rekursive Beschreibung eines Algorithmus zwar übersichtlicher und wesentlich kürzer ausfällt, dafür aber mehr Arbeitsspeicher verbraucht [20]. Dies liegt an der Tatsache, dass Funktionsaufrufe einen Overhead an Daten erzeugen, welcher für das Betriebssystem notwendig zur Koordination des Programmablaufs ist. Für die Realisierung von Funktionsaufrufen in einem Programm müssen in der Regel folgende Schritte bewältigt werden.

- Die Funktionsparameter müssen entsprechend ihrer Beschreibung eine Speicheradresse erhalten, welche ausreichend groß für deren Typen ist
- Der Rückgabewert der Funktion benötigt ebenfalls eine passende Speicherreservierung.
- Der Funktionsname muss aufgelöst werden, um die richtige Stelle im Speicher zu finden, auf welche der *Program Counter* zur Ausführung der Funktion springen muss.
- Die Parameter und die *return*-Adresse des Funktionsaufrufs [1] müssen auf den *Stack* geschrieben werden.
- Nachdem alle nötigen Daten auf dem *stack* platziert sind, springt der *Program Counter* zur Adresse der Funktion um die Ausführung zu ermöglichen.
- Nach der erfolgreichen Abarbeitung der Funktion, müssen die Parameter und der Rückgabewert der Funktion wieder vom *Stack* entfernt werden und das Ergebnis wird an der *return*-Adresse gespeichert.

Die meisten dieser Operationen sind optimiert und kosten sehr wenig Zeit. Trotzdem kann es bei hoher Rekursionstiefe zu erheblicher Speicherallokation kommen, da mit jedem

[1] Speicheradresse an die nach der Ausführung der Funktion zurückgesprungen werden soll und das Ergebnis gespeichert wird

Rekursionsschritt ein neuer Funktionsaufruf nötig ist und somit Overhead produziert wird, welcher erst nach Beendigung der Rekursion wieder freigegeben wird. Iterative Implementierungen sind zwar aufwendiger zu entwickeln, benötigen diesen Overhead jedoch nicht. Ob sich dieser Effekt auch signifikant auf den Energieverbrauch auswirkt, wird durch die Betrachtung in Kapitel 5 untersucht.

2.5 Gegenüberstellung anhand verschiedener Mergesort Varianten

2.5.1 Rekursiver Mergesort

Um die rekursive mit der iterativen Ausführung sinnvoll vergleichen zu können, wird ein Algorithmus benötigt, welcher sowohl mit seiner iterativen als auch mit seiner rekursiven Version in der selben Komplexitätsklasse der $O-$Notation nach Donald E. Knuth liegt. Für diese Untersuchung wurde für diesen Zweck der Mergesort Algorithmus ausgewählt. Es handelt sich hierbei um ein stabiles Sortierverfahren [2], welches im Worst-, Best- und Average-Case die Laufzeitkomplexität von $O(n * log(n))$ besitzt [21, S. 96]. Außerdem gibt es rekursive und iterative Versionen des Algorithmus mit der selben Laufzeitkomplexität [22, 134 f.]. Der Nachteil des hier verwendeten Verfahrens ist, dass es vergleichsweise viel Speicher benötigt, da es sich um ein *out-of-place* Verfahren handelt. Das bedeutet, dass für die Durchführung der Sortierung eine gesonderte Speicherung der Daten zur Bearbeitung notwendig ist und nicht ausschließlich mit dem Speicherbedarf der Eingabedaten gearbeitet wird.

Der Mergesort ist eine Anwendung des *Teile und herrsche Verfahrens* aus der Informatik. Hierbei geht es darum, ein komplexes Hauptproblem solange in seine Teilprobleme aufzuteilen, bis die einzelnen Teilprobleme mit geringem Aufwand lösbar sind. Anschließend werden die gelösten Teilprobleme wieder zusammengeführt, bis die Gesamtlösung des Hauptproblems erreicht wurde [21, S. 9]. Der Mergesort setzt dieses Prinzip mithilfe von drei Teilschritten um.

1. Die zu sortierende Menge wird solange zerlegt, bis nur noch einelementige Mengen vorhanden sind.

2. Die Teilmengen werden nun einzeln während der Zusammenführung sortiert

3. Die so entstandenen sortierten Teilmengen müssen zum Schluss mithilfe eines Suchkriteriums wieder zusammengeführt werden

[2] Stabile Sortierverfahren behalten die relative Reihenfolge von gleichgroßen Elementen bei.

In Quellcode 2.3 ist die rekursive Java-Implementierung zu sehen, welche in der „EnergyEfficience“ Applikation verwendet wurde. Zunächst wird ein Array vom Typ *int* im Konstruktor übergeben, welches die Eingabemenge enthält. In Zeile sieben ist zu erkennen, dass der Mergesort ein Hilfs-Array der Größe der Eingabedaten benötigt. In der *sort(int l, int r)*-Methode findet die Aufteilung der Eingabemenge in seine Teilmengen statt. Hierfür werden mit l und r jeweils der Anfangs- und Endindex des zu sortierenden Arrays angegeben. Mit diesen beiden Werten wird der Mittelindex q der Eingabemenge ermittelt. Nun erfolgen die rekursiven Aufrufe der *sort*-Methode um jeweils die linke und die rechte Hälfte der Eingabemenge zu sortieren. Zum Schluss wird für den aktuellen Abschnitt die *merge*-Methode aufgerufen. Diese ist für die rekursive Zusammenführung der einzelnen Teilstücke zuständig. Im Zuge dieser Zusammenführung werden die einzelnen Teilstücke sortiert. Diese Rekursion wird solange fortgesetzt, bis die Bedingung $l < r$ nicht mehr erfüllt wird. Sobald dieser Punkt erreicht ist, wurde die Eingabemenge in seine einelementigen Teilmengen gespalten und der Funktionsstack kann abgearbeitet werden. In der *merge*-Methode werden drei Schleifen benötigt. Die erste Schleife initialisiert den Hilfs-Array $arr[]$ mit der linken Hälfte der aktuellen Teilmenge aus dem Eingabe-Array $intArr[]$. Hierzu werden die Parameter l und q als Indexgrenzen verwendet. Analog wird in der zweiten Schleife die Rechte Hälfte der aktuellen Teilmenge in den Hilfs-Array geschrieben. Die vierte for-Schleife dient der Zusammenführung der beiden Hälften. Dabei wird aufsteigend der Größe nach sortiert und in den Haupt-Array $intArr[]$ zurückgeschrieben.

Quellcode 2.3: rekursiver Merge sort (Quelle: [1])

```
1  public class MergeSortImplementation {
2      public static int[] intArr;
3      public static int[] arr;
4
5      public MergeSortImplementation(int[] intArr) {
6          this.intArr = intArr;
7          this.arr = new int[intArr.length];
8      }
9
10     public int[] sort(int l, int r) {
11         if (l < r) {
12             int q = (l + r) / 2;
13             sort(l, q);
14             sort(q + 1, r);
15             merge(l, q, r);
16         }
17         return intArr;
18     }
19
20     private void merge(int l, int q, int r) {
21         int i, j;
22         for (i = l; i <= q; i++) {
23             arr[i] = intArr[i];
24         }
25         for (j = q + 1; j <= r; j++) {
26             arr[r + q + 1 - j] = intArr[j];
27         }
28         i = l;
29         j = r;
```

```
30          for (int k = l; k <= r; k++) {
31              if (arr[i] <= arr[j]) {
32                  intArr[k] = arr[i];
33                  i++;
34              } else {
35                  intArr[k] = arr[j];
36                  j--;
37              }
38          }
39      }
40  }
```

2.5.2 Iterativer Mergesort

In Quellcode 2.4 ist eine iterative Variante des Mergesorts zu sehen. Es ist zu erkennen, dass die iterative Version nicht ganz so intuitiv zu verstehen ist wie die Rekursion. Die Erfahrung zeigt, dass iterative Lösungen meist aufwendiger zu entwickeln sind als rekursive. Dafür verlangen iterative Implementierung weitaus weniger Speicher während der Ausführung. Auch in Quellcode 2.4 werden neben dem Konstruktor zwei Methoden für den eigentlichen Algorithmus benötigt. Die $mergesort$-Methode implementiert zwei ineinander verschachtelte Schleifen. Die äußere Schleife iteriert über alle Elemente des Eingabearrays. In der Inneren Schleife wird für einen stetig wachsenden Bereich der Eingabemenge, die $merge$-Methode aufgerufen. Dabei wird der Hauptarray $a[]$ von rechts nach links, Stück für Stück sortiert. Nach jeder Iteration wird der an die $merge$-Methode übergebene Bereich erweitert. Die zusätzlichen Elemente werden dann in die schon sortierte Teilmenge an die richtige Stelle eingefügt. Die $merge$-Methode funktioniert prinzipiell genauso wie die der rekursiven Implementierung.

Quellcode 2.4: iterativer Mergesort (Quelle: [2])

```
1  public class MergeSortImplementationIterative {
2      private int[] a;
3      private int[] b;    // Hilfsarray
4      private int n;
5
6      public void sort(int[] a) {
7          this.a = a;
8          n = a.length;
9          b = new int[n / 2];
10         mergesort();
11     }
12
13     private void mergesort() {
14         int m, s;
15         for (s = 1; s < n; s += s)
16             for (m = n - 1 - s; m >= 0; m -= s + s)
17                 merge(max(m - s + 1, 0), m, m + s);
18     }
19
```

```
20      void merge(int lo, int m, int hi) {
21          int i, j, k;
22          i = 0;
23          j = lo;
24          // vordere Hälfte von a in Hilfsarray b kopieren
25          while (j <= m)
26              b[i++] = a[j++];
27          i = 0;
28          k = lo;
29          // jeweils das nächstgrößte Element zurückkopieren
30          while (k < j && j <= hi)
31              if (b[i] <= a[j])
32                  a[k++] = b[i++];
33              else
34                  a[k++] = a[j++];
35          // Rest von b falls vorhanden zurückkopieren
36          while (k < j)
37              a[k++] = b[i++];
38      }
39
40      private int max(int a, int b) {
41          return a > b ? a : b;
42      }
43  }
```

2.5.3 Paralleler Mergesort

Für den parallelen Mergesort wurde wurde ein *ForkJoinPool* implementiert. Hierbei handelt es sich um eine spezielle Form des *ExecuterService*, welche für die Ausführung von parallelen, rekursiven Algorithmen optimal geeignet ist. Für den *ForkJoinPool* gibt es spezielle Ableitungen der *Runnable*-Klasse um das Prinzip des „work-stealing's“ umzusetzen [23]. Diese Klassen sind zum einen *ForkJoinTask<T>* und zum anderen *RecursiveAction<T>*, welche in Quellcode 2.5 verwendete wurde. In der *compute*-Methode ist zu sehen, dass für jede Teilung der Eingabemenge ein neues Objekt vom Typ *RecursiveAction* erzeugt wird. Diese Objekte fungieren als *Runnable* für den *ForkJoinPool*. Die Besonderheit dieser Struktur ist, dass jedes Objekt vom Typ *RecursiveAction* seine eigene Queue hat, in welche durch Rekursion wiederum neue Aufgabenpakete vom Typ *RecursiveAction* hinzugefügt werden können. Der *ForkJoinPool* ist in der Lage auf jede dieser Queues zuzugreifen und bei Bedarf Aufgabenpakete aus diesen Queues an einen freien Thread zuzuteilen. Außerdem kümmert sich der *ForkJoinPool* um die Synchronisierung der Zusammenführung der einzelnen Teilergebnisse der terminierten *RecursiveActions* [24]. Damit von der parallelen Ausführung der einzelnen Teilaufgaben der Rekursion auch wirklich profitiert werden kann, muss eine Obergrenze für die Teilung der Eingabemenge gesetzt werden. in Zeile sechs ist zu sehen, dass diese Grenze auf 8192 Elemente angegeben wurde. Dies bedeutet, dass die Eingabemenge nur solange in Teilmengen zerlegt wird, bis die einzelnen Teilmengen aus maximal 8192 Elementen bestehen. Ist dieser Punkt erreicht, werden diese Teilmengen nach dem sequentiellen Verfahren aus Quellcode 2.3 sortiert. Dieser Schritt

ist notwendig, da es zu ressourcenaufwendig wäre, für jedes Element ein vergleichsweise speicherintensives *RecursiveAction*-Objekt zu erstellen. Ansonsten würde die Ausführung sogar langsamer ausfallen als die rein sequentielle Sortierung.

Quellcode 2.5: paralleler Mergesort (Quelle: [3])

```
1  public class ParallelMergeSort extends RecursiveAction {
2      private final int[] array;
3      private  final int[] helper;
4      private  final int low;
5      private final int high;
6      private final int MAX = 8192;
7      public ParallelMergeSort(final int[] array, final int[] helper, final
           int low, final int high){
8          this.array = array;
9          this.low = low;
10         this.high = high;
11         this.helper = helper;
12     }
13     @Override
14     protected void compute() {
15         if (low < high) {
16             if (high - low <= MAX) { // Sequential implementation
17                  sort(low, high);
18             } else { // Parallel implementation
19                 final int middle = (low + high) / 2;
20                 final ParallelMergeSort left =
21                         new ParallelMergeSort(array,helper, low, middle);
22                 final ParallelMergeSort right =
23                         new ParallelMergeSort(array,helper,middle + 1, high
                               );
24                 invokeAll(left, right);
25                 merge(low, middle, high);
26             }
27         }
28     }
29
30 /*****Nutzung der der RucursiveAction******/
31
32     ForkJoinPool forkJoinPool = new ForkJoinPool(Runtime.getRuntime().
           availableProcessors() - 1);
33     forkJoinPool.invoke(new ParallelMergeSort(...));
```

3 Verwendete Geräte und Tools

3.1 Vorstellung der Test-App für diese Arbeit

Im Rahmen dieser Arbeit wurde die „EnergyEfficience“ App entwickelt. In dieser App wurden verschiedene Implementierungen von Algorithmen umgesetzt, mit deren Hilfe Laufzeit- und Energieeffizienz dieser Berechnungen gemessen werden kann. Die „EnergyEfficience“ App wurde auf Basis des Android Software Developoment Kit (SDK) 30 entwickelt, welche die zum Zeitpunkt der Erstellung dieser Arbeit die aktuellste Android API ist und somit für Android 11 empfohlen wird. Für die Programmierung selbst wurde ausschließlich mit Java gearbeitet. Allerdings ist Kotlin mittlerweile die empfohlene Programmiersprache für die Androidentwicklung, da die Bereitstellung der neuen SDK's und der Android Support Bibliotheken immer stärker auf das Motto „Kotlin-first“ [25] ausgerichtet wird. Inwiefern die Java Unterstützung für zukünftige SDK Versionen bestehen bleibt, ist zum aktuellen Zeitpunkt nicht absehbar. Für die persistente Speicherung von beispielsweise Testdaten und Nutzereinstellungen wurde die Room Persistence Library in der Version 2.2.5 genutzt.

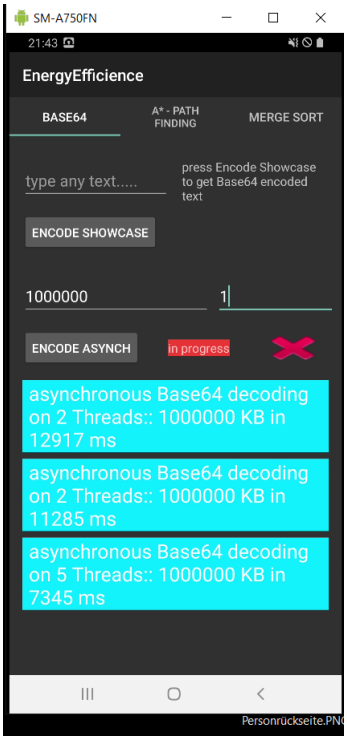

Abbildung 3.1: MainActivity der EnergyEfficience App (eigene Abbildung)

In Abbildung 3.1 ist das UI der Launch Activity der App zu sehen. Über ein dreiteiliges Tab Layout Menü sind drei verschiedene Fragmente erreichbar, welche jeweils einen Algorithmus thematisieren. Die Fragmente sind über einen View Pager und einen Fragment State Adapter mit dem Tab Layout gekoppelt. Das Base64 Fragment implementiert einen einfachen Base64 Encoder. Über die Schaltfläche „ENCODE SHOWCASE“ wird der Text aus dem darüber gelegenen Eingabefeld in das entsprechende Base64-Format kodiert und in der rechts angrenzenden TextView ausgegeben. Die für die Messung relevanten Berechnungen lassen sich über den „ENCODE ASYNCH“ Button starten. Bevor dies geschieht, sollte in das links darüber gelegene Textfeld die Größe des zu kodierenden Strings angeben werden. Dabei ist die Einheit KB und es sollte darauf geachtet werden, dass aus Gründen der internen Implementierung die Mindestgröße von 10000 KB nicht unterschritten wird. Über das rechts daneben platzierte Textfeld lässt sich die Anzahl der Threads festlegen, die zur parallelen Ausführung der Kodierung angewandt werden soll. Die detaillierte Implementierung dieser Parallelität mithilfe eines Threadpools wurde bereits in Quellcode 2.1 aus Kapitel 2 beschrieben. Während der Laufzeit der Kodierung erscheint eine rot unterlegte „in progress“ Meldung, welche nach Terminierung wieder verschwindet. Die Laufzeitdaten und Parameter des Durchlaufs werden anschließend in blau gestalteten Textboxen innerhalb einer scrollbaren Recycler View dargestellt. Mithilfe des roten Kreuzes kann diese Recycler View wieder geleert werden.

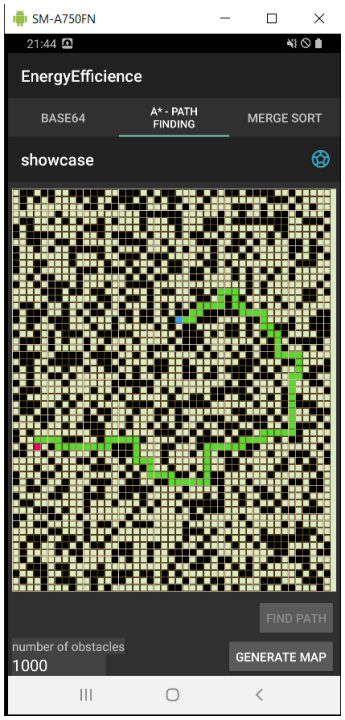

Abbildung 3.2: A*-Path finding Fragment (eigene Abbildung)

Im A*-Path finding Fragment wurde eine Implementierung des A*-Pathfinding Algorithmus umgesetzt. Die UI ist in Abbildung 3.2 zu sehen. Über die Schaltfläche „GENERATE MAP“ wird eine zweidimensionale Rasterkarte mit zufällig generierten Start- und Endpunkten einer Route und beliebig vielen Hindernissen erstellt. Hierbei werden Hindernisse durch schwarze Kacheln dargestellt. Nach Betätigung der „FIND PATH“ Schaltfläche wird, falls vorhanden, der kürzeste Weg zwischen Start- und Endpunkt ermittelt und anhand einer grünen Markierung gekennzeichnet.

In Abbildung 3.3 ist auf der linken Seite das Mergesort Fragment zu sehen, welches zur Veranschaulichung der verschiedenen Merge Sort Varianten dient und auf der rechten Seite die Mergesort Activty, welche zwar keine visuelle Darstellung der sortierten Daten enthält, dafür aber die Möglichkeit bietet, beliebig lang andauernde Sortierungen durchzuführen. Für die Durchführung der Messungen ist daher diese Activity vorgesehen.

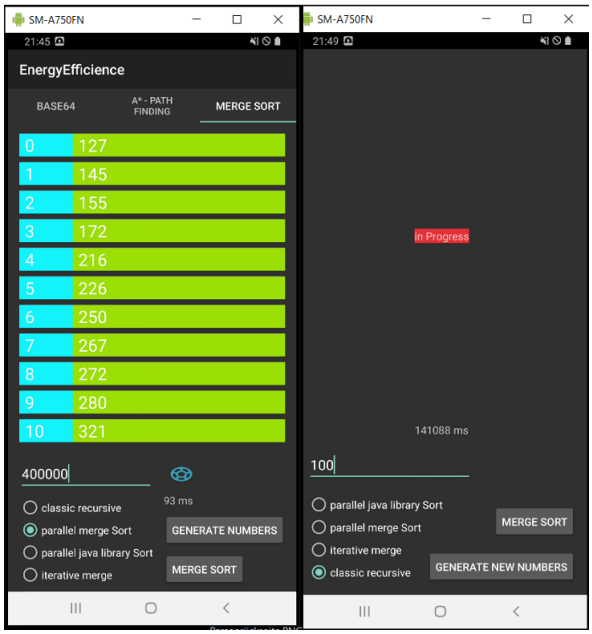

Abbildung 3.3: MergeSort Fragment und Activity zur Messung (eigene Abbildung)

Über das Textfeld kann eine beliebige Anzahl von Nummern festgelegt werden, welche durch den Algorithmus sortiert werden soll. Es wird empfohlen, dabei die Obergrenze von Millionen (Mio) nicht zu überschreiten, da es bei der Allokation von Arbeitsspeicher für Datenstrukturen wie Arrays oder Listen mit so vielen Elementen zu Out of Memory Erros kommen kann. Diese führen unweigerlich zum Absturz der Applikation. Durch betätigen der „GENERATE NUMBERS“ Schaltfläche werden dieser Anzahl entsprechend viele Zufallszahlen zwischen 0 und 9000000 in zufälliger Reihenfolge generiert und Recycler View dargestellt. Dabei gibt der blau hinterlegte Index die Position der Zahl im internen Array an. Über die vier Radio Buttons kann zwischen verschiedenen Sortierverfahren gewählt werden. Für die Untersuchung im Rahmen dieser Arbeit sind der klassische, rekursive Mergesort, der iterative Mergesort und der parallele Mergesort relevant. Über die „MERGE SORT“ Schaltfläche kann die Sortierung gestartet werden. Auch hier wird nach Terminierung der Berechnung die Laufzeit in Millisekunden (ms) ausgegeben. Die Elemente in der Recycle View werden anschließend der neuen Sortierung entsprechend aktualisiert. Die Einschränkung durch den vergleichsweise geringen Arbeitsspeicher auf mobilien Geräten verhindert die Generierung von ausreichend vielen Testdaten um eine für Messungen günstige Laufzeit zu erreichen. Daher wurde eine weitere Activity erstellt, um dieses Problem für längere Messungen zu umgehen. Auf der rechten Seite von Abbildung 3.3 ist zu erkennen, dass das UI bis auf das Fehlen der Recycler View einen ähnlichen Aufbau wie das Mergesort Fragment besitzt. Hierbei ist jedoch zu Beachten, dass über das Textfeld nicht die Anzahl der Elemente eingestellt wird, sondern die Anzahl der Iterationen einer Sortierung. In jeder Iteration werden 3 Mio Zahlen sortiert. Dieses Vorgehen verhindert die gleichzeitige Allokation von zu viel Heap Speicher und somit Out of Memory Errors und gewährleistet außerdem eine beliebig andauernde Laufzeit für die Messung. Des weiteren bleiben die Testdaten im Gegensatz zum Mergesort Fragment konstant, da 3 Mio

Zahlen mithilfe einer Room Database persistent gespeichert werden. Über die Schaltfläche „GENERATE NEW NUMBERS“ können 3 Mio neue Zufallszahlen zwischen 0 und 9 Mio generiert und in die Datenbank gespeichert werden. Über das blaue, kreisförmige Icon des Mergesort Fragments, ist diese Activty zu erreichen. Bei der Nutzung dieser Activity ist zu beachten, dass das die erstmalige Betätigen der glqq MERGE SORT“ Schaltfläche nach Anwendungsstart zu einer kurzen Wartezeit führt, da zunächst die Daten aus der Datenbank geladen werden. Diese initiale Ladezeit ist natürlich aus der Zeitmessung ausgeschlossen.

3.2 Gerätespezifikationen und Battery Historian zur Messdatenermittlung

Als Testgerät wurde das Samsung Galaxy A7 aus dem Jahr 2018 gewählt. Es besitzt 64 Giga Byte (GB) Festplattenspeicher und vier GB Arbeitsspeicher. Der Akku besitzt eine Kapazität von 3300 Milliamperestunden (mAh). Bei dem sechs Zoll großen Display handelt es ich um einen Super-AMOLED mit einer Auflösung von 1080 x 2220 Pixel bei einer Pixeldichte von 411 Pixel per Inch (ppi) [26]. Die CPU ist ein Achtkernsystem mit der Architektur eines Exynos 7885 System on a Chip (SoC) von Samsung. Diese Architektur besteht zum einen aus zwei schnellen Cortex-A73 Rechenkernen mit 2,20 Gigahertz (GHz) und zum anderen aus sechs stromsparenden Cortex-A53 Kernen mit jeweils 1,6 GHz Taktfrequenz. Exynos ist eine SoC Familie, deren Mikroprozessorkomponenten auf der ARM-Architektur basieren. Diese Architektur zeichnet sich durch einen effizienten Befehlssatz aus, der eine kompakte Implementierung im ASCII-Design erlaubt und daher für Optimierungen im Bereich des Energiesparens sehr gut geeignet ist. Dieses Design wurde ursprünglich vom britischen Unternehmen ARM entwickelt. Große Unternehmen wie Apple, Huawei, Qualcomm oder Samsung besitzen Lizenzen für diese Architektur und können daher hauseigene Lösungen mit dieser Technik für ihre Geräte entwickeln [27]. Die mit dieser Architektur einhergehende Besonderheit, dass Kerne aus zwei verschiedenen Leistungsklassen verbaut werden, spiegelt sich auch in den Messergebnissen der Untersuchung in Kapitel 4 „Energieeffizientes Multithreading“ wieder.

Zur Messung des Strom- und Spannungsverlaufs des Akkus während der Ausführung der Berechnungen wurde das Programm Battery Historian in der Version 3.0 genutzt. Um dieses Tool nutzen zu können, ist eine aktuelle Docker Umgebung nötig.

Im Rahmen dieser Arbeit wurde die Version v19.03.12 genutzt. Bei Docker handelt es sich um eine Anwendung zur Erstellung und Verwaltung von sogenannten Linux Containern. Dies ermöglicht eine komfortable und schnelle Installation und Skalierung von Software jeglicher Art. Sofern eine Installation der Docker Engine vorhanden ist, lassen sich so neue Softwareinstanzen eines beliebigen Programms unabhängig vom installierten Betriebssystem und vorhanden Bibliotheken mit wenig Installationsaufwand aufsetzen. Alle Voraussetzungen und nötigen Konfigurationen für das Aufsetzen der gewünschten

Software oder Anwendungsinstanz werden hierbei durch ein Docker Image vorgegeben und müssen nicht mehr manuell umgesetzt werden. Virtuelle Maschinen liefern zwar einen ähnlichen Komfort, sind jedoch vergleichsweise ressourcenaufwendig, da Hier ganze Betriebssysteme mit hohem Speicherbedarf installiert werden. Die Docker Engine hingegen ermöglicht einen direkte Zugriff auf den Kernel des Host-Betriebssystems und spart somit Speicher und ist wesentlich schneller in der Ausführung als eine herkömmliche virtuelle Maschine [28]. In Verbindung mit zusätzlichen Tools wie Kubernetes, kann die Skalierung und Verwaltung der Docker Container und somit auch der eigenen Softwareinstanzen fast vollständig automatisiert werden [29].

Soblad das aktuelle Docker Image von Battery Historian läuft, ist es möglich, die von Android gesammelten Logdateien bezüglich des Batteriestatus in einer HTML Visualisierung darzustellen. Informationen über den Energieverbrauch und Statistiken über die Nutzung von sämtlichen Hardwarekomponenten des Gerätes werden permanent im Hintergrund gesammelt. Hierzu wird unter anderem das Tool Batterystats genutz, welches in das Android Framework integriert ist [30]. Mithilfe der Android-Debug-Bridge (ADB) [1] ist es möglich, diese Logdaten in ein ZIP-Dateiformat zu komprimieren und auf ein anderes Gerät zu laden. Hierfür wird der Befehl *adb bugreport bugreport.zip* genutzt [31].

In Abbildung 3.4 ist die grafische Darstellung dieser Logdateien zu sehen. Die horizontal verlaufenden farbigen Balken kennzeichnen die Aktivität verschiedener Komponenten des Smartphones, welche zum jeweiligen Zeitpunkt Strom verbrauchen. So lassen sich CPU Aktivitäten, Wakelocks oder Wireless Local Area Network (Wfi) Aktivitäten nachvollziehen. Für netzwerkspezfische Datensätze wie „Mobile signal strength“ oder „Wifi signal strength“ nimmt der Balken in Abhängigkeit der Signalstärke verschiedene Farben an. So steht grün für optimale Signalstärke und rot für eine schlechte Verbindung. Sehr nützlich für die Untersuchung sind vor allem die Informationen zur aktuellen Betriebstemperatur des Chips und die Anzeige von benutzerimplizierten Wakelocks. Die Betriebstemperatur ist ein signifikanter Faktor, der die Leistungsfähigkeit der CPU beeinflusst und daher während der Messungen möglichst konstant gehalten werden muss, um vergleichbare Ergebnisse zu erhalten. Bei vielen Geräte wird bei zu hoher Betriebstemperatur die Taktrate des Systems verringert, um die eine Überhitzung zu vermeiden und die Lebensdauer der Chips zu verlängern.

Über ein Dropdown-Menü rechts oberhalb des Diagramms lassen sich verschiedene Graphen Anzeigen. In Abbildung 3.4 ist der Voltage Graph zu sehen, welcher den Spannungsverlauf in Millivolt (mV) über der Ziet darstellt. Das Tool zeichnet Messwerte im Intervall von 30 Sekunden auf. Daher sind relativ lang andauernde Messungen über mehrere Minuten notwendig, um sinnvolle Wertereihen zu erhalten. Die Spannung wird für jedes 30 Sekundenintervall als Spannungsbereich mit Ober- und Untergrenze angegeben. In Abbildung 3.4 ist beispielsweise ein Messpunk ausgewählt mit einer Spannung, die im aktuellen 30 Sekundenintervall zwischen 4127 mV und 3986 mV liegt. Für die Berechnungen im Kontext der Untersuchung dieser Arbeit wurden stets die Mittelwerte dieser Spannungsbe-

[1] die ADB ist ein Schnittstellen Programm zwischen Computer und Androidgerät zur Ausführung von Befehlen und zur Übertragung von Dateien

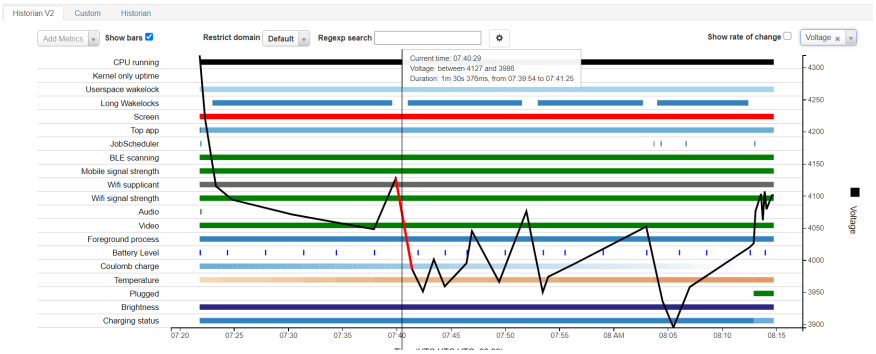

Abbildung 3.4: Battery Historian Benutzeroberfläche mit Spannungsverlauf „Voltage“ (eigene Abbildung)

reiche ermittelt und genutzt. Um Werte für die Stromstärke in Abhängigkeit von der Zeit einsehen zu können, muss entweder der Graph *Battery Level* oder *Coulomb charge* genutzt werden. Auch hier werden die Werte in Intervallen von 30 Sekunden angezeigt. Allerdings wird hier ein genauer Wert in Milliampere (mA) angegeben, welcher neben dem Wert für den aktuellen Füllstand des Akkus in mAh als „Charge rate“ abzulesen ist. Mithilfe der beiden Graphen für den Spannungs- und Entladestromverlauf kann die elektrische Leistung über der Zeit abgebildet werden. Anschließend kann durch Integration dieses Graphen der Energieverbrauch insgesamt als verrichtete elektrische Arbeit ermittelt werden. Dieses Vorgehen wird in den folgenden Kapiteln mit der Anwendung auf die Messwerte näher beschrieben. Alle Messungen wurden unter möglichst identischen Grundvoraussetzungen durchgeführt. Es wurde stets das gleiche Gerät verwendet. Jede Messung wurde mit identischen Akkufüllstand gestartet. Außerdem wurde darauf geachtet, dass während den Messungen keine Netzwerkverbindungen aktiv waren. Einstellungen wie Bildschirmhelligkeit oder Betriebsmodus wurden nicht variiert. Im Zeitraum der Untersuchung wurden außerdem keine Updates des Betriebssystems zugelassen um eine möglichst konstante Laufzeitumgebung zu gewährleisten.

4 Energieeffizientes Multithreading

4.1 Darstellung der Messwerte

In diesem Kapitel werden die Ergebnisse der Messungen für die Untersuchung des Zusammenhangs zwischen Multithreading und der Energieeffizienz vorgestellt. Für diese Untersuchung wurden insgesamt 16 Messungen der Ausführung des Base64-Encoders mit steigender Anzahl von Threads unternommen. Dabei wurde mit jeder weiteren Messung ein neuer Thread zur Ausführung hinzugefügt, bis die Obergrenze dieser Betrachtung von 16 Threads erreicht wurde. Dies entspricht der doppelten Anzahl an physischen Rechenkernen des verwendeten Gerätes. Um den Zeitrahmen dieser Untersuchung nicht zu sprengen, wurden keine Messungen mit einer noch größeren Anzahl von Threads durchgeführt. An dieser Stelle sei gesagt, dass trotz der Begrenzung auf 16 Threads eine klare Tendenz zu erkennen ist, welche weitere Messungen überflüssig machen könnte.

In Tabelle 4.1 sind zunächst die ermittelten Laufzeiten der Ausführungen mit steigender Anzahl von Threads zu sehen. Es ist erkennbar, dass bis zu der Marke von 11 Threads ein stetiger Zuwachs an Ausführungsgeschwindigkeit zu erkennen ist, da die Laufzeit bis zu diesem Punkt konstant abnimmt. Zwar ist die Ausführung ab 12 Threads immer noch performanter als ursprünglich, verliert jedoch mit wachsender Anzahl an Threads allmählich an Geschwindigkeit.

In Abbildung 4.2 ist ein Diagramm dieser Entwicklung zu sehen. Da im Testgerät acht physische Rechenkerne verbaut sind, kann das Gerät nur bis zu acht Threads wirklich parallel bearbeiten. Ab der Marke von neun Threads ist die CPU gezwungen ständig zwischen den Threads hin und her zu wechseln. Diese Art der Ausführung ist zwar noch nebenläufig aber nicht vollständig parallel. Bei jedem Kontextwechsel muss der Status der Ausführung des aktuellen Threads gespeichert werden, sodass die CPU später an der richtigen Stelle mit der Berechnung fortfahren kann. Das Speichern und Laden der Register- und Prozessinformationen bei solch einem Kontextwechsel fordert Rechenaufwand und Zeit [32, S. 2]. Mit steigender Anzahl von Threads verstärkt sich dieser Effekt. Nun könnte die Annahme getroffen werden, dass aus diesem Grund die optimale Thread-Anzahl gleich der Anzahl an physischen Threads ist. Dies würde im Fall des hier verwendeten Gerätes bedeuten, dass acht Threads die schnellste Laufzeit erreichen müssten. Es ist jedoch zu erkennen, dass die Ausführung mit 11 Threads das beste Ergebnis liefert. Hierbei wurde mit 355677 ms 64,2 % weniger Laufzeit benötigt als mit einem Thread. Mit acht Threads hingegen waren es mit 364560 ms nur 63,3 % weniger. Eine weitere Auffälligkeit ist der massive Geschwindigkeitszuwachs bei dem Übergang von einem Thread zu zwei Threads. Der Speedup $S_p(2)$ nach Gleichung 2.1 beträgt 1,57 und die daraus ermittelte

Tabelle 4.1: Laufzeit in Abhängigkeit von der Thread Anzahl

Threads	t in ms	Speedup	Effizienz
1	994358	1,000	1,000
2	631836	1,574	0,787
3	507871	1,958	0,653
4	441615	2,252	0,563
5	416036	2,390	0,478
6	387519	2,566	0,428
7	369424	2,692	0,385
8	364560	2,728	0,341
9	364637	2,727	0,303
10	355893	2,794	0,279
11	355677	2,796	0,254
12	364313	2,729	0,227
13	363055	2,739	0,211
14	377380	2,635	0,188
15	383928	2,590	0,173
16	395553	2,514	0,157

Laufzeiteffizienz für zwei Threads $E_p(2)$ nach Gleichung 2.2 beträgt 0,79. Statt den ursprünglichen 994358 ms mit einem Thread benötigt die Kodierung mit zwei Threads nur noch 631836 ms bis zur Terminierung. Das entspricht einer Laufzeitverringerung von 36,5 %. Nach der Zuschaltung von Thread 3 beträgt die Laufzeit 507871 ms. Das sind nur 12,4 % weniger als die Kodierungszeit mit zwei Threads. Die Laufzeiteffizienz $E_p(3)$ beträgt daher nur noch 0,65 für die Ausführung mit drei Threads.

Hierfür gibt es zwei Hauptgründe. Der massive Performancezuwachs bei der Nutzung von zwei Threads ist mit der Exynos 7885 SoC-Architektur der CPU des verwendeten Gerätes zu erklären. Diese Architektur kombiniert zwei leistungsstarke Cortex-A73 Rechenkerne mit jeweils 2,20 GHz Taktrate und vier energiesparende Cortext-A53 Kerne mit jeweils 1,60 GHz Taktrate. Bei der Nutzung von zwei Threads, wird zunächst der Cortex-A73 zugeschaltet. Danach stehen ausschließlich die restlichen Cortex-A53 Kerne zur Verfügung, welche mit ihrer geringeren Taktrate natürlich auch einen geringeren Speedup $S_p(n)$ liefern. Die Tatsache, dass die Ausführungsgeschwindigkeit bis zu der Marke von 11 Threads ansteigt, obwohl nur acht physische Kerne vorhanden sind, ist ebenfalls mit der Exynos 7885 SoC-Architektur zu begründen. Ab der Marke von neun Threads, werden mehr *Callables* bearbeitet als physische Kerne vorhanden sind. Das bedeutet, dass ab neun Threads zusätzliche Kontextwechsel durchgeführt werden müssen, um alle Threads mit Rechenzeit versorgen zu können. Da trotzdem bessere Laufzeiten bis zur Marke von 11 Threads zustande kommen, liegt die Vermutung nahe, dass die beiden Cortex-A73 Rechenkerne aufgrund ihrer höheren Rechenleistung für die Kontextwechsel priorisiert werden und daher letztendlich mehr Anteile der Kodierung durchführen. In der Implementierung des Base64-Encoders werden die einzelnen Aufgabenpakte für die Threads so vergeben, dass jedes Aufgabenpaket (*Callable*) die gleiche Textmenge kodiert. Die beiden Cortex-A73 Rechenkerne sind aufgrund ihrer höheren Taktrate natürlich schneller mit ihrem Aufgabenpaket fertig als die andern Kerne. Somit befinden sich die Cortex-A73 Rechenkerne bis zu der Marke von acht Threads im Leerlauf, sobald sie ihre Aufgabenpakete abgearbeitet haben und auf die Terminierung der restlichen Kerne warten. Ab der Nutzung von weiteren Threads kommt es nicht mehr zu diesen Leerlaufzeiten, da durch den Kontextwechsel die nebenläufige Bearbeitung von mehreren Arbeitspaketen durch den selben Rechenkern möglich wird. Der prozentuale Anteil des kodierten Textes durch die Cortex-A73 Rechenkerne steigt also ab der neun Thread-Marke. Die bessere Auslastung der beiden schnellen Rechenkerne relativiert bis zur Marke von 11 Threads den Zeitaufwand für die Kontextwechsel und führt zu einer schnelleren Laufzeit. Ab der Marke von 12 Threads nimmt der Aufwand für die Kontextwechsel zwischen den vielen Threads allerdings so stark zu, dass der Speedup allmählich sinkt und die Laufzeitkosten wieder steigen.

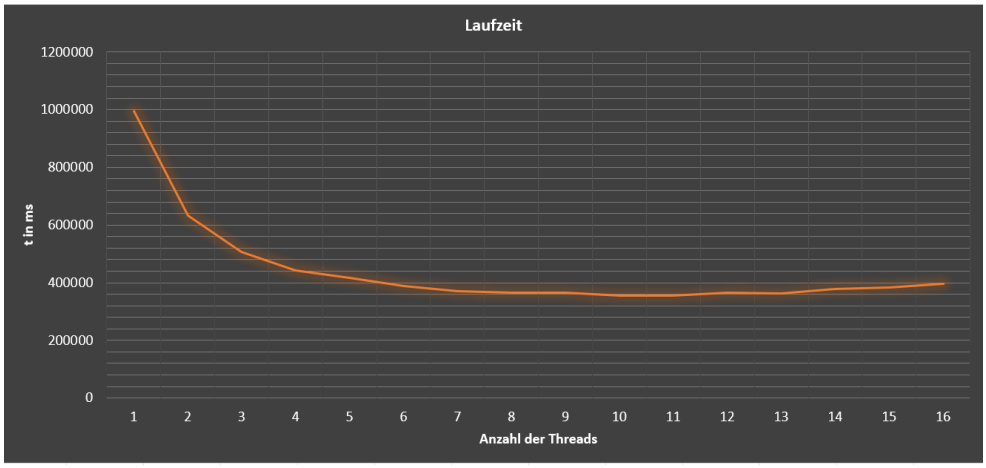

Abbildung 4.1: Laufzeitdiagramm (eigene Abbildung)

Ob die Entwicklung der Laufzeit mit dem Energieverbrauch korreliert, wird anhand der folgen Diagramme diskutiert. In Abbildung 4.2 wird zunächst die durchschnittliche Leistungsaufnahme des Akkus während der Base64-Kodierung in Abhängigkeit von der Thread-Anzahl dargestellt. Die Leistungswerte für dieses Diagramm und die dazugehörige Tabelle 4.2 wurden ermittelt, indem das arithmetische Mittel aus den einzelnen Messergebnissen ermittelt wurde. Diese Messergebnisse sind in tabellarischer Form im Anhang zu finden. Der Graph in Abbildung 4.2 zeigt, dass die stärkste Zunahme der durchschnittlichen Leistungsaufnahme bei der Zuschaltung von zwei und drei Threads zu verzeichnen ist. Auch hierfür ist die initiale Auslastung der beiden Cortex-A73 Rechenkerne verantwortlich. Die höhere Taktrate dieser beiden Kerne verursacht auch eine höhere Leistungsaufnahme verglichen mit den niedrig getakteten Cortex-A53 Kernen. Bis zur Marke von acht Threads ist, wie zu erwarten war, ein relativ konstanter Anstieg im Graphen zu erkennen. Hierbei werden nach und nach die restlichen Cortex-A53 Kerne für die parallele Ausführung hinzugezogen. Erst ab der Verwendung von neun Threads wird der Verlauf unstetig. So fällt die durchschnittliche Leistungsaufnahme bei neun Threads von 3,772 Watt (W) auf 3,662 W ab. Danach steig der Graph jedoch wieder bis zum Höchstwert von 3,824 W bei 11 Threads an. Der kurzzeitige Einbruch bei neun Threads könnte durch die Ungenauigkeit der Messmethode mittels Battery Historian zustande gekommen sein. Da die Messzeitpunkte für Stromstärke und Spannung in relativ großen Abständen von 30 Sekunden liegen, fallen kurzzeitige Spannungsabfälle und niedrige Entladeströme bei den anschließenden Berechnungen stärker ins Gewicht. Davon abgesehen kann dieser Verlauf mit dem Diagram aus Abbildung 4.1 in Zusammenhang gebracht werden. Die schnellste Ausführung bei 11 Threads zieht demnach die höchste Leistungsaufnahme nach sich. Diese Tatsache stützt außerdem die Begründung der besseren Auslastung der leistungsfähigen Cortex-A73 Rechenkerne bei 11 Threads. Diese erhalten durch den Kontextwechsel mehr Anteile an der Kodierung, was zu höherer Auslastung führt und einen größeren Leistungsaufwand nach sich zieht. Ab der Nutzung von 12 Threads ist ein starker Einbruch der Leistungsaufnahme auf einen Durchschnittswert von 3,48 Watt zu verzeichnen. Die Ursache für diesen plötzliche Einbruch im Graphen konnte im Rahmen

dieser Untersuchung nicht eindeutig geklärt werden. Eine Vermutung liegt darin, dass ab dieser Anzahl von Threads vermehrte Speicherzugriffe durch den erhöhten Aufwand beim Scheduling zu Wartezeiten führt. Jedes mal wenn ein Kontextwechsel vollzogen wird, muss die CPU darauf warten, dass alle nötigen Daten des neuen Threads in die Register geladen werden. Allerdings spricht der weiter Verlauf des Graphen bis zur finalen Thread-Anzahl von 16 gegen diese Annahme, da ein kontinuierlicher Anstieg zu erkennen ist. Daher ist ein Fehler bei der Durchführung der Messung ab der 12 Thread-Marke nicht auszuschließen. So könnte die Erhöhung der Temperatur aufgrund der hohen Auslastung während der Messung eine automatische Reduzierung der Taktrate ausgelöst haben um die Hardware zu schonen. Da eine erneute Durchführung der Messungen den Zeitrahmen dieser Arbeit sprengen würde, wird trotzdem mit den Vorhanden Werten gearbeitet.

Tabelle 4.2: durchschnittliche Leistungsaufnahme in Abhängigkeit von der Thread Anzahl

Threads	ØU in mV	ØI in mA	ØP in W
1	4077,382	565,944	2,306
2	3997,636	733,983	2,931
3	4029,556	860,404	3,462
4	4056,719	859,178	3,479
5	4043,867	869,086	3,505
6	4028,429	900,618	3,621
7	4017,545	923,407	3,705
8	4042,727	933,792	3,772
9	4063,227	901,938	3,662
10	4046,045	934,489	3,775
11	4034,455	948,283	3,824
12	4074,458	856,408	3,485
13	4049,000	962,385	3,486
14	4063,917	915,918	3,586
15	4021,708	926,291	3,726
16	4023,542	946,001	3,806

Anhand der bisherigen Betrachtung des Laufzeit- und Leistungsdiagramms konnte festgestellt werden, dass die parallele Ausführung mit steigender Thread-Anzahl bis zur Grenze von 11 Threads zwar schneller verläuft, jedoch mehr Leistungsaufnahme und damit hö-

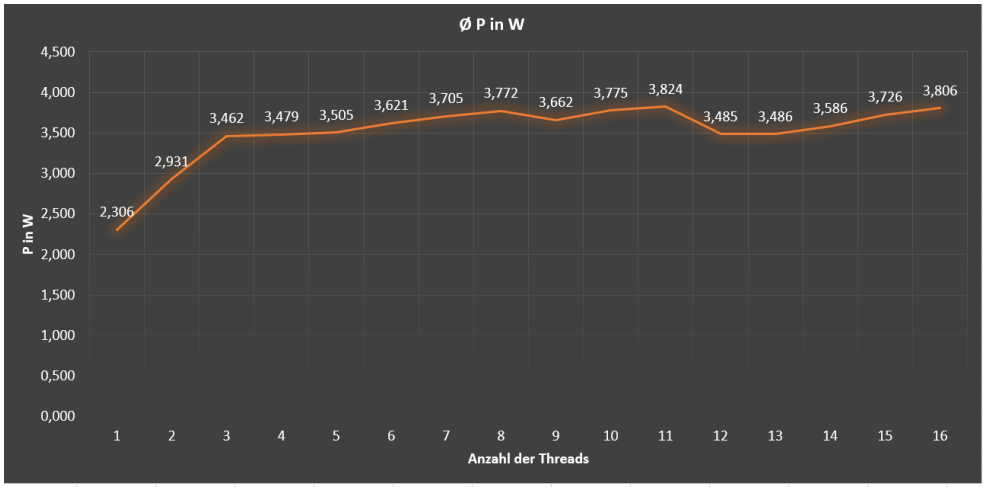

Abbildung 4.2: durchschnittliche elektrische Leistung in Abhängigkeit von der Thread Anzahl (eigene Abbildung)

heren Akkuverbrauch fordert. Nun gilt es zu klären, welcher der beiden Faktoren die Oberhand behält und ob die schnellere Laufzeit trotz der erhöhten Leistungsaufnahme insgesamt zu einer energieeffizienteren Kodierung führt. Zu diesen Zweck wurde für jede Messung die elektrische Leistung über der Zeit abgebildet. Aus dieser Betrachtung ergeben sich für jede Ausführung mit entsprechender Thread-Anzahl Diagrammdarstellungen der Leistung über die Zeit der Ausführung. In Abbildung 4.3 ist exemplarisch ein Leistungsdiagramm der Ausführung mit einem Thread zu sehen. Um eine Bewertung des Energieverbrauchs durchführen zu können, wurde mithilfe dieses Graphen die während der Kodierung verrichtete elektrische Arbeit des Gerätes ermittelt. Hierzu musste die orange unterlegte Fläche unterhalb des Graphen der Leistungskurve bestimmt werden. Da es sich hierbei um unregelmäßige Messewerte handelt, welche sich schlecht auf eine sinnvoll integrierbare Funktionsgleichung abbilden lassen, musste auf das Riemannsche Integral zurückgegriffen werden. Hierbei handelt es sich um ein mathematisches Näherungsverfahren zur Bestimmung des Integrals einer reellen Funktion auf einem festgelegten Intervall. Bei diesem Verfahre wird die Fläche unterhalb eines Graphen in beliebig viele Rechtecke aufgeteilt. Die Flächen der Rechtecke werden anschließend aufsummiert und liefern schlussendlich eine Approximation der Gesamtfläche zwischen dem Graphen und der X-Achse bezüglich des gewählten Intervalls [33, S. 479]. Um so feingliedriger die Gesamtfläche aufgeteilt wird, um so genauer wird auch die Approximation. Da die Messwertaufnahmen durch Battery Historian in 30 Sekundenabständen durchgeführt werden, ist für die Approximation der elektrischen Arbeit diese Feingliedrigkeit beschränkt. So sind die Rechteckflächen stets 30 Zeiteinheiten breit. Eine so ermittelte Teilfläche, repräsentiert somit die elektrische Arbeit, die während dieser 30 Sekunden verrichtet wurde. Durch die Aufsummierung dieser Werte wird die gesamte elektrische Arbeit während einer Kodierung approximiert. Eine Übersicht der so ermittelten Ergebnisse für die parallele Base64-Kodierung mit unterschiedlich vielen Threads ist in Tabelle 4.3 zu finden.

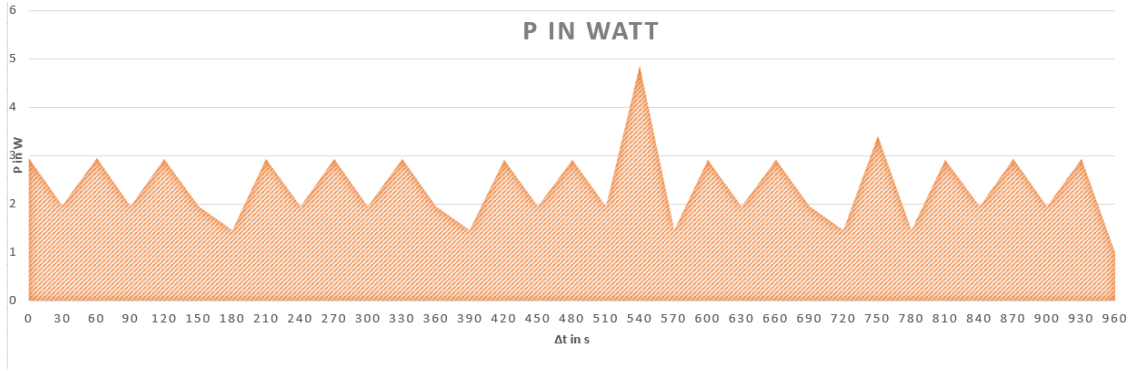

Abbildung 4.3: Verlauf der elektrischen Leistung mit einem Thread (eigene Abbildung)

Tabelle 4.3: elektrische Arbeit in Abhängigkeit von der Thread Anzahl

Threads	W in Ws
1	2330,170
2	1891,658
3	1815,574
4	1594,628
5	1532,691
6	1473,056
7	1347,732
8	1296,977
9	1252,836
10	1290,069
11	1306,284
12	1316,109
13	1322,537
14	1374,502
15	1393,833
16	1409,848

Der Graph von Abbildung 4.4 zeigt einen kontinuierlichen Abfall der verrichteten Arbeit mit steigender Thread-Anzahl bis zur Marke von neun Threads. Ab 10 Threads steigt der Graph allmählich wieder. Daraus lässt sich ableiten, dass die parallele Durchführung mit neun Threads am energieeffizientesten ist. Allerdings muss an dieser Stelle erwähnt werden, dass durch die verzerrte Leistungsmessung bei der Kodierung mit neun Threads, welche schon in vorherigen Abschnitten erläutert wurde, eine Verschiebung des Wertes der elektrischen Arbeit für neun Threads zustande gekommen sein könnte. Auffällig ist hier der leichte Einbruch des Graphen bei neun Threads, welcher sich ebenfalls in Abbildung 4.2 wiederspiegelt. Es könnte daher durchaus diskutiert werden, ob die energieeffizienteste Kodierung doch mit acht Threads erreicht ist. Da der Graph einen stetigen Abfall bis zur Marke von neun Threads zeigt, ist anzunehmen, dass bis zu diesem Punkt der Geschwindigkeitsvorteil und die damit verbundene Verkürzung der Rechenzeit schwerer wiegt als der Zuwachs der Leistungsaufnahme. Trotz der Tatsache, dass die optimale Laufzeit mit 11 Threads erreicht wurde, steigt der Energieverbrauch ab 10 Threads stetig, da die benötigte Laufzeit für die Kodierung mit neun Threads bis zur Kodierung mit 11 Threads nur noch um 0,9 % sinkt und die durchschnittliche Leistungsaufnahme im Gegenzug um sieben Prozent ansteigt. Ab diesem Punkt wiegt der Zuwachs an elektrischer Leistungsaufnahme also stärker als die verringerte Laufzeit. Somit kommt es trotz der schnellsten Ausführung mit 11 Threads zu einer Steigerung des Energieverbrauchs. Ab der Marke von 12 Threads ist der stetige Anstieg des Energieverbrauchs einerseits mit der allmählich erhöhten Laufzeit durch das aufwendige Thread-Management und andererseits mit der steigenden Leistungsaufnahme ab 12 Threads zu begründen.

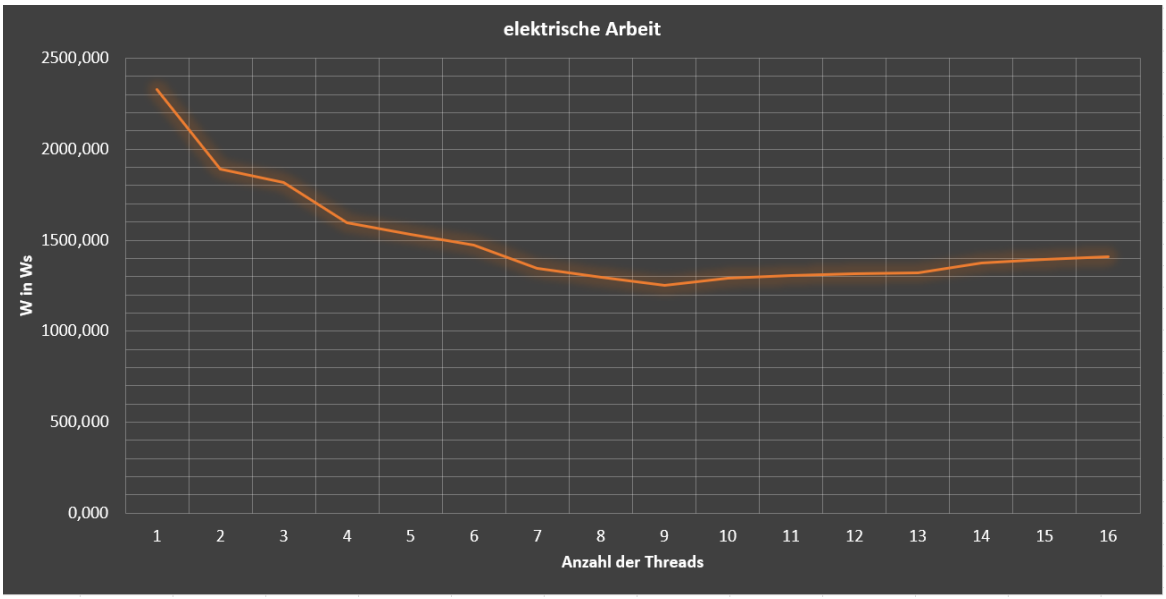

Abbildung 4.4: elektrische Arbeit in Abhängigkeit der Thread Anzahl (eigene Abbildung)

4.2 Gewonnene Erkenntnisse

Nachdem die Messergebnisse der parallelen Base64-Kodierung im vorherigen Abschnitt dargestellt wurden und mögliche Ursachen für die beobachtete Entwicklung erläutert wurden, folgt in diesem Unterkapitel die Aufzählung der gewonnene Erkenntnisse hinsichtlich der parallelen Ausführung auf mobilen Geräten.

Es konnte bestätigt werden, dass auf einem Gerät mit Mehrkernprozessor die parallele Ausführung von Threads bis zu einem bestimmten Punkt nicht nur schneller sondern auch stromsparender wird. Dabei ist allerdings zu beachten, dass die optimale Thread-Anzahl hinsichtlich der Laufzeitoptimierung nicht immer gleich der optimalen Thread-Anzahl hinsichtlich der Energieeffizienz ist. So wurde mit dem hier verwendeten Samsung Galaxy A7 bei der Verwendung von 11 Threads zwar die schnellste Ausführung festgestellt, die energiesparendste Variante war jedoch die Kodierung mit neun Threads. Ein wesentlicher Faktor für diesen Sachverhalt ist der immer kleiner werdende Gewinn an Ausführungsgeschwindigkeit (Speedup) pro zusätzlichen Thread, verglichen mit der steigenden Leistungsaufnahme. Diese Entwicklung ist sehr gut im Kurvenverlauf von Abbildung 4.1 zu erkennen. Außerdem wurde festgestellt, dass die Architektur der CPU einen erheblichen Einfluss auf die optimale Anzahl von Threads haben kann. So wurde aufgrund der hier verwendeten Exynos 7885 SoC-Architektur mit zwei 2,20 GHz Kernen und vier 1,60 GHz Kernen die erwartete Optimalananzahl von acht Threads, entsprechend der vorhanden acht physischen Rechenkerne, nach oben verschoben. Dies liegt an der besseren Auslastung der zwei stärkeren Kerne durch vermehrten Kontextwechsel bei der Verwendung von mehr Threads als CPU-Kernen. Es ist allerdings zu beachten, dass dies nur im Spezialfall einer Chiparchitektur mit unterschiedlich getakteten Kernen zutrifft. Bei homogenen Architekturen mit ausschließlich gleichartigen Rechenkernen ist zu vermuten, dass die optimale Thread-Anzahl für parallele Berechnungen gleich der Anzahl an vorhanden Prozessorkernen ist. Dies ist hauptsächlich für die Laufzeit der Fall. Hinsichtlich der Energieeffizienz ist zu beachten, dass der prozentuale Gewinn an Laufzeitgeschwindigkeit nicht den Zuwachs an Leistungsaufnahme pro zusätzlichen Thread unterschreiten sollte. Da die Graphen aus Abbildung 4.4 und Abbildung 4.1 ähnlich verlaufen, ist eine linearer Abhängigkeit zwischen Energieverbrauch und der Laufzeit anzunehmen.

5 Rekursive und Iterative Verfahren im Vergleich

5.1 Darstellung der Messwerte

Dieses Kapitel befasst sich mit der Gegenüberstellung der iterativen, rekursiven und parallel-rekursiven Ausführung hinsichtlich der Energieeffizienz. Hierbei werden die verschieden Mergesort-Implementierungen der „EnergyEfficience“ Applikation genutzt. Die „EnergyEfficience“ Applikation bietet eine sequentielle iterative Mergesort Variante sowie eine rekursive Mergesort Variante. Beide besitzen nach der O-Notation theoretisch die selbe Laufzeitkomplexität. Da rekursive Implementierungen aufgrund der sich selbst aufrufenden Funktionen allerdings mehr Overhead produzieren als iterative Implementierungen, ist die Laufzeitgeschwindigkeit von iterativen Ausführungen meist höher. Ob sich dieser Umstand ebenfalls im Energieverbrauch wiederspiegelt, wird mit folgenden Messergebnissen diskutiert. Außerdem wird eine parallele Rekursion mittels *ForkJoinPool* Implementierung dargestellt. Die theoretischen Eigenschaften der Rekursion und die jeweiligen Implementierungen der Mergesort Varianten sind in Kapitel zwei zu finden.

Tabelle 5.1 zeigt, dass das durchschnittliche Spannungsniveau bei allen drei Varianten relativ ähnlich bleibt. Der gemessene Entladungsstrom ist bei den beiden sequentiellen Merge Sort Verfahren ebenfalls fast gleich, was in einer annähernd identischen Leistungsaufnahme für diese beiden Verfahren resultiert. Daraus ist abzuleiten, dass die sequentielle Ausführung von rekursiven und iterativen Algorithmen keine unterschiedliche Belastung für die CPU mit sich bringt. Der Mehraufwand durch das Verwalten des größeren Overheads bei rekursiven Verfahren beansprucht also hauptsächlich den Speicher. Bei der parallelen Ausführung mittels *ForkJoinPool* ist hingegen ein klarer Anstieg des Leistungsniveaus zu erkennen. Dies ist natürlich auf die parallele Verwendung aller vorhandenen Rechenkerne zurückzuführen.

Tabelle 5.1: durchschnittliche elektrische Leistung, Stromstärke, Spannung der Merge Sort Verfahren

Merge Sort Variante	ØU in mV	ØI in mA	ØP in Watt
rekursiv	4115,286	495,052	2,037
iterativ	4148,460	487,577	2,022
parallel	4111,500	778,158	3,197

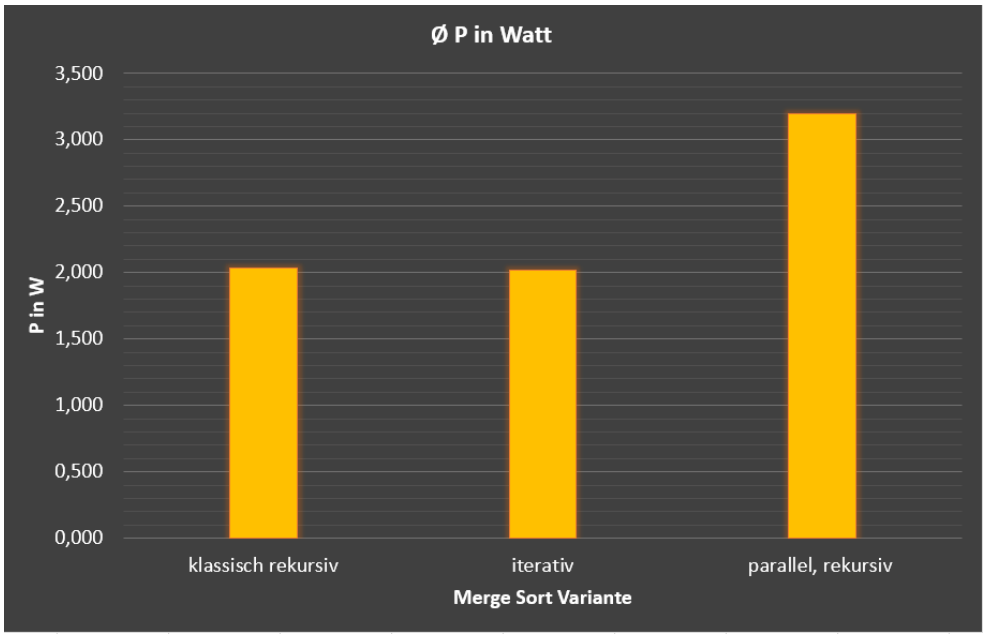

Abbildung 5.1: Merge Sort: durchschnittliche elektrische Leistung pro Verfahren (eigene Abbildung)

In Tabelle 5.2 ist eine Gegenüberstellung der Laufzeiten und des Energieverbrauchs der verschieden Mergesort Varianten zu sehen. Die grafische Darstellung erfolgt in Abbildung 5.2 in einem Balkendiagramm. Da die durchschnittliche Leistungsaufnahme der sequentiell, rekursiven und iterativen Sortierung beinahe identisch ausfällt, ist für den Vergleich der Energieeffizienz der beiden Verfahren die Laufzeit der ausschlaggebende Faktor. So ist die iterative Sortierung mit einer Laufzeit von 714,388 Sekunden ungefähr 11,2 % schneller als die rekursive Sortierung. Aufgrund der fast identischen Leistungsaufnahme der beiden Verfahren sinkt auch die verrichtete elektrische Arbeit der iterativen Sortierung um vergleichbare 11,5 %. Die parallele Sortierung mittels *ForkJoinPool* kann aufgrund der Nutzung von allen Rechenkernen eine deutliche Verringerung der Laufzeit um 64,7 % vorweisen. Da jedoch die Leistungsaufnahme aufgrund der stärkeren Auslastung der CPU während der Ausführung wesentlich höher ist als die der beiden sequentiellen Varianten, sinkt die benötigte elektrische Arbeit nur um 41,1 % bezüglich der langsams-

ten, sequentiellen Ausführung. Trotzdem ist dies mit Abstand die energieeffizienteste Implementierung.

Tabelle 5.2: elektrische Arbeit und Laufzeit der Merge Sort Varianten Gegenüberstellung

Merge Sort Variante	t in s	W in Ws
rekursiv	804,252	1662,462
iterativ	714,388	1471,137
parallel	284,021	979,787

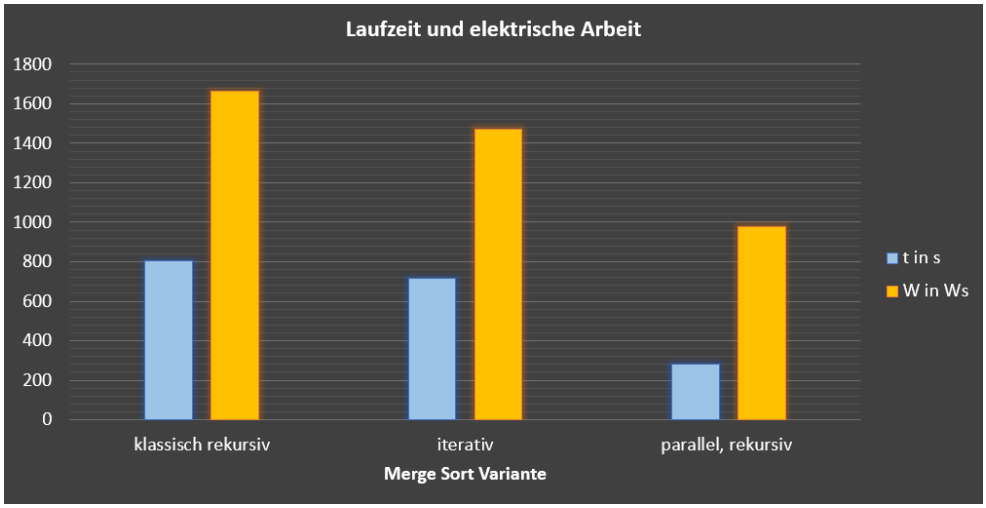

Abbildung 5.2: Merge Sort: Gegenüberstellung von Laufzeiten und elektrischer Arbeit (eigene Abbildung)

5.2 Gewonnene Erkenntnisse

Für die Gegenüberstellung von iterativer und rekursiver Implementierung lässt sich nach der Betrachtung der Messergebnisse festhalten, dass iterative Implementierungen nicht nur laufzeiteffizienter sind, sonder auch energieeffizienter. Dies liegt allerdings nicht an einer höheren Auslastung der CPU, welche für iterative und rekursive Implementierungen gleicher Komplexität nahezu identisch ausfällt. Der entscheidende Faktor, ist die erhöhte Laufzeit bei rekursiven Verfahren, welcher mit der stärkeren Belastung des Speichers zu begründen ist. Aufgrund der verschachtelten Funktionsaufrufe bei rekursiven Algorithmen sind wesentlich mehr Speicherzugriffe notwendig. Diese Speicherzugriffe benötigen jedes Mal vergleichsweise viel Zeit und bremsen die Berechnung der CPU. Für jeden Funktionsaufruf, muss der entsprechende Overhead des Funktionsaufrufs auf dem Heap des Speichers allokiert werden bevor die eigentliche Berechnung der CPU starten kann.

Während der Zeit der Allokation, befindet sich die CPU im Wartezustand und ist daher nicht optimal ausgelastet. Ein Beweis für diesen Effekt kann die minimal geringere Leistungsaufnahme während der rekursiven Sortierung sein. Dies ist in Tabelle 5.1 dargestellt. Die parallele Merge Sort Variante mittels *ForkJoinPool* ist mit Abstand am energiesparendsten. Allerdings muss bei der Implementierung rekursiver Algorithmen mit einem *ForkJoinPool* darauf geachtet werden, dass die Rekursionstiefe sinnvoll begrenzt wird. Da für jede Teilung des Problems (Teile und Herrsche Verfahren) jeweils ein neues Java-Objekt (*RecursiveAction*) erstellt wird, würde eine Zerstückelung der Eingabemenge bis hin zu Mengen mit nur einem Element zu viel Speicher kosten und den Geschwindigkeitsvorteil zunichte machen. Generell sollten daher iterative Lösungen der Rekursion vorgezogen werden. Besonders im Kontext von mobilen Geräten ist dies empfehlenswert, da vor allem der Arbeitsspeicher durch rekursive Algorithmen stärker belastet wird und dies bis heute ein begrenzender Faktor bei mobilen Geräten ist.

6 Auswertung

6.1 Zusammenfassung

Mobile Geräte erfreuen sich einer stetig wachsenden Beliebtheit in fast allen Lebensbereichen und sind mittlerweile in vielen Branchen unverzichtbare Arbeitswerkzeuge. Leider ist die permanente Nutzung dieser nützlichen Hilfsmittel durch die Akkukapazität zeitlich begrenzt. Ziel dieser Arbeit war es daher, Implementierungen von Algorithmen möglichst energieeffizient umzusetzen. Dabei lagen die Schwerpunkt zum einen auf der Untersuchung paralleler Berechnungen mithilfe von Multithreading und zum anderen auf der Gegenüberstellung rekursiver und iterativer Ausführungen hinsichtlich des Energieverbrauchs.

Um das Verhalten des Energieverbrauchs in Abhängigkeit der Thread-Anzahl und des gewählten Verfahrens zu messen, wurde eine Android-Applikation entwickelt. Hier wurde ein paralleler Base64-Decoder mit dynamisch wählbarer Thread-Anzahl implementiert, um die Abhängigkeit zwischen Energieverbrauch und Multithreading zu untersuchen. Des Weiteren wurde der Mergesort-Algorithmus in verschieden Ausführungen in die App integriert, da dieser sich sehr gut für eine Gegenüberstellung von iterativer und rekursiver Ausführung eignet. Es ist anzumerken, dass die hier genutzten Algorithmen durchaus austauschbar sind, da es hauptsächlich um die Untersuchung der Konzepte Multithreading und Rekursion beziehungsweise Iteration ging. Beispielsweise hätte die Untersuchung des Multithreadings auch mit anderen parallelisierbaren Algorithmen durchgeführt werden können. Mithilfe des Programms *Battery Historian* wurden die Android-Logdateien hinsichtlich der Spannung und des Entladestroms ausgelesen und anschließend grafisch ausgewertet. Aus den so ermittelten Messwerten und Diagrammen konnten einige Erkenntnisse bezüglich der Zielstellung gewonnen werden.

Es wurde festgestellt, dass mit steigender Thread-Anzahl bei der parallelen Ausführung nicht nur die Laufzeit sondern auch der Energieverbrauch solange sinkt, bis eine optimale Anzahl an Threads erreicht wurde. Bei weiterer Erhöhung der Thread-Anzahl steigen Energiebedarf und Laufzeit wieder an. Die optimale Anzahl an Threads ist hierbei abhängig von der vorliegenden Prozessorarchitektur. Im Fall des hier verwendeten Samsung Galaxy A7 mit seinen unterschiedlich starken Rechenkernen übersteigt die optimale Thread-Anzahl leicht die Anzahl der physischen Kerne. Für Geräte mit ausschließlich gleichstarken Rechenkernen ist jedoch eine Thread-Anzahl zu empfehlen, die der Anzahl an physischen Kernen entspricht.

Weiterhin konnte gezeigt werden, dass iterative Implementierungen energiesparender als rekursive Lösungen sind, sofern die zugrundeliegenden Algorithmen in der selben Komplexitätsklasse liegen. Da die Auslastung der CPU für beide Ansätze nahezu identisch ausfällt, ist der erhöhte Energieaufwand rekursiver Verfahren mit der höheren Speicherbelastung zu begründen. Häufige Speicherzugriffe sind charakteristisch für rekursive Verfahren und führen zu erhöhten Laufzeiten. Vor allem für mobile Geräte ist ein speicherschonendes Vorgehen angebracht, da der vergleichsweise kleine Arbeitsspeicher neben der Akkukapazität ein weiterer begrenzender Faktor ist. So kann eine aufwendige Rekursion eine mögliche Ursache für häufig auftretenden OOM-Abstürze sein.

6.2 Ausblick

Das Thema der energieeffizienten Implementierung auf mobilen Geräten bietet auch für künftige Arbeiten und Untersuchungen weitere Ansätze mit anderen Schwerpunkten. Da der Einfluss der Prozessorarchitektur im Rahmen der hier vorgenommen Messung eindeutig zutage kam, ist es sinnvoll diesen Aspekt zukünftig näher zu beleuchten. So könnten verschiedene Prozessorarchitekturen mobiler Geräte hinsichtlich der Energieeffizienz verglichen werden. Darüber hinaus könnte für eine spezifische Architektur, die optimale Implementierung für parallele Ausführung ermittelt werden. Einen weiteren Ansatz bietet die Betrachtung der Speichereffizienz von verschiedenen Algorithmen und Vorgehensweisen, da Speicherzugriffe signifikanten Einfluss auf die Laufzeit haben und dadurch auch den Energieverbrauch beeinflussen.

Literaturverzeichnis

[1] javabeginners.de. *Mergesort*. 3. Juli 2016. URL: https://javabeginners.de/Algorithmen/Sortieralgorithmen/Mergesort.php (besucht am 19. 10. 2020) (zitiert auf den Seiten x, 16).

[2] H. W. Lang. *Mergesort iterativ*. 16. Apr. 2005. URL: https://www.inf.hs-flensburg.de/lang/algorithmen/sortieren/merge/mergiter.htm (besucht am 19. 10. 2020) (zitiert auf den Seiten x, 17).

[3] T. Harsanyi. *Parallel Merge Sort in Java*. 8. Okt. 2018. URL: https://medium.com/@teivah/parallel-merge-sort-in-java-e3213ae9fa2c (besucht am 19. 10. 2020) (zitiert auf den Seiten x, 19).

[4] C. Gulz. *Große Smartphones ab 5,5 Zoll besonders gefragt*. 20. Feb. 2019. URL: https://www.idealo.de/magazin/2019/02/20/grosse-smartphones-immer-beliebter/#:~:text=Seit%202010%20sind%20Smartphones%20von,%2DZoll%2DDisplays%20am%20beliebtesten (besucht am 28. 10. 2020) (zitiert auf Seite 1).

[5] Google. *Schedule tasks with WorkManager*. 19. Okt. 2020. URL: https://developer.android.com/topic/libraries/architecture/workmanager (besucht am 28. 10. 2020) (zitiert auf Seite 1).

[6] Google. *Keep the device awake*. 27. Dez. 2019. URL: https://developer.android.com/training/scheduling/wakelock (besucht am 28. 10. 2020) (zitiert auf Seite 1).

[7] J. Wolf. *C von A bis Z: das umfassende Handbuch*. Galileo computing. Rheinwerk Verlag, 2020. ISBN: 978-3-8362-3973-8. URL: http://openbook.rheinwerk-verlag.de/c_von_a_bis_z/026_c_paralleles_rechnen_002.htm (zitiert auf den Seiten 4, 5).

[8] Elektronik-Kompendium. *Pipelining*. 14. Okt. 2020. URL: https://www.elektronik-kompendium.de/sites/com/1705221.htm (besucht am 14. 10. 2020) (zitiert auf den Seiten 4, 5).

[9] U. Kirch und P. Prinz. *C++ lernen und professionell anwenden*. mitp Verlag, 2012. ISBN: 978-3-8266-9195-9 (zitiert auf Seite 5).

[10] Elektronik-Kompendium. *Vom Takt-orientierten Prozessor zum Mehr-Kern-Prozessor*. 14. Okt. 2020. URL: https://www.elektronik-kompendium.de/sites/com/1203171.htm (besucht am 14. 10. 2020) (zitiert auf Seite 6).

[11] G. Bengel, C. Baun und M. Kunze. *Masterkurs Parallele und Verteilte Systeme*. GWV Fachverlage GmbH, Wiesbaden, 2008. ISBN: 978-3-8348-0394-8 (zitiert auf den Seiten 6, 7).

[12] Google. *Asynchronous Programming Techniques*. 2020. URL: https://kotlinlang.org/docs/tutorials/coroutines/async-programming.html (besucht am 16. 10. 2020) (zitiert auf Seite 8).

[13] C. Ullenboom. *14.1.1 Threads und Prozesse*. 2011. URL: http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_14_001.htm (besucht am 28. 10. 2020) (zitiert auf Seite 8).

[14] C. Ullenboom. *14.4 Der Ausführer (Executor) kommt*. 2011. URL: http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_14_004.htm (besucht am 28. 10. 2020) (zitiert auf Seite 9).

[15] Oracle. *Interface BlockingQueue<E>*. 24. Juni 2020. URL: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html (besucht am 17. 10. 2020) (zitiert auf Seite 9).

[16] S. Josefsson. *The Base16, Base32, and Base64 Data Encodings*. 1. Okt. 2006. URL: https://tools.ietf.org/html/rfc4648 (besucht am 20. 10. 2020) (zitiert auf Seite 12).

[17] Freed und Borenstein. *6.8. Base64 Content-Transfer-Encoding*. 1. Nov. 1996. URL: https://tools.ietf.org/html/rfc2045 (besucht am 20. 10. 2020) (zitiert auf Seite 12).

[18] D. E. Knuth. *The art of computer programming: Fundamental algorithms 3re Edition*. Addison Wesley Longman, 1997. ISBN: 0-201-89683-4 (zitiert auf Seite 12).

[19] Google. *Handler*. 30. Sep. 2020. URL: https://developer.android.com/reference/android/os/Handler (besucht am 20. 10. 2020) (zitiert auf Seite 13).

[20] Björn und B. Petri. *Iteration und Rekursion*. 2010. URL: https://www.java-tutorial.org/iteration_und_rekursion.html (besucht am 24. 10. 2020) (zitiert auf Seite 14).

[21] T. Ottmann und P. Widmayer. *Algorithmen und Datenstrukturen*. Spektrum Akademischer Verlag, 1996. ISBN: 978-3827401106 (zitiert auf Seite 15).

[22] J. W. von Gudenberg. *Algorithmen Datenstrukturen Funktionale Programmierung Eine praktische Einführung mit Caml Light*. AddisonWesley Longman Verlag GmbH, 1996. ISBN: 3-8273-1056-3 (zitiert auf Seite 15).

[23] Oracle. *Class ForkJoinPool*. 9. Juli 2020. URL: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html (besucht am 26. 10. 2020) (zitiert auf Seite 18).

[24] J. Jenkov. *Fork and Join Explained*. 3. Feb. 2015. URL: http://tutorials.jenkov.com/java-util-concurrent/java-fork-and-join-forkjoinpool.html (besucht am 26. 10. 2020) (zitiert auf Seite 18).

[25] Google. *Android's Kotlin-first approach*. 23. Sep. 2020. URL: https://developer.android.com/kotlin/first (besucht am 10. 10. 2020) (zitiert auf Seite 20).

[26] D. Gillengerten. *Samsung Galaxy A7 (2018) Daten*. 19. Nov. 2018. URL: https://www.inside-digital.de/handys/samsung-galaxy-a7-2018 (besucht am 13. 10. 2020) (zitiert auf Seite 24).

[27] K. Hinum. *Samsung Exynos 7885 SoC - Benchmarks und Specs*. 19. Mai 2020. URL: https://www.notebookcheck.com/Samsung-Exynos-7885-SoC-Benchmarks-und-Specs.285086.0.html (besucht am 13. 10. 2020) (zitiert auf Seite 24).

[28] I. Docker. *Docker overview*. 12. Okt. 2020. URL: https://docs.docker.com/get-started/overview/ (besucht am 13. 10. 2020) (zitiert auf Seite 25).

[29] Google. *What is Kubernetes?* 5. Aug. 2020. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (besucht am 13. 10. 2020) (zitiert auf Seite 25).

[30] Google. *Profile battery usage with Batterystats and Battery Historian*. 15. Jan. 2020. URL: https://developer.android.com/topic/performance/power/setup-battery-historian (besucht am 13. 10. 2020) (zitiert auf Seite 25).

[31] *Battery Historian*. 20. Mai 2017. URL: https://github.com/google/battery-historian (besucht am 13. 10. 2020) (zitiert auf Seite 25).

[32] T. Franke. „Erweiterte Konzepte in C++: Multithreading“. Magisterarb. Technische Universität Darmstadt, 2005 (zitiert auf Seite 27).

[33] W. Leupold. *Mathematik - ein Studienbuch für Ingenieure Band 1: Algebra - Geometrie - Analysis für eine Variable*. Fachbuchverlag Leipzig im Carl Hanser Verlag, 2004. ISBN: 978-3-446-22583-1 (zitiert auf Seite 32).

Selbstständigkeitserklärung

Hiermit erkläre ich, Niklas Kluge, dass die von mir an der *Hochschule für Telekommunikation Leipzig (FH)* eingereichte Abschlussarbeit zum Thema

„Untersuchung des Stromverbrauchs von verschiedenen Algorithmen und deren Implementierung auf mobilen Geräten“

selbstständig verfasst wurde und von mir keine anderen als die angegebenen Quellen und Hilfsmittel verwendet wurden.

Magdeburg, den 2. November 2020

Niklas Kluge

A Anhang

A.1 Messwerte der parallelen Base64-Kodierung

Tabelle A.1: Base64-Kodierung mit einem Threads

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
0	0,498474675	119,61	4167,5	51,625943
30	2,94325486	717,08	4104,5	73,585476
60	1,96244354	478,12	4104,5	73,576857
90	2,94268023	716,94	4104,5	73,419077
120	1,9519249	478,12	4082,5	73,20147
150	2,928173125	717,25	4082,5	73,200246
180	1,95184325	478,1	4082,5	51,231905
210	1,463617075	358,51	4082,5	65,863993
240	2,9273158	717,04	4082,5	73,182487
270	1,95151665	478,02	4082,5	73,194122
300	2,928091475	717,23	4082,5	73,199021
330	1,95184325	478,1	4082,5	73,1782
360	2,926703425	716,89	4082,5	73,1782
390	1,95184325	478,1	4082,5	51,236804
420	1,463943675	358,59	4082,5	65,618469
450	2,910620905	716,99	4059,5	72,766538
480	1,940481595	478,01	4059,5	72,773845
510	2,911108045	717,11	4059,5	72,775671

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
540	1,94060338	478,04	4059,5	101,86706
570	4,85053417	1194,86	4059,5	94,594063
600	1,4557367	358,6	4059,5	65,493537
630	2,91049912	716,96	4059,5	72,763493
660	1,940400405	477,99	4059,5	72,779934
690	2,911595185	717,23	4059,5	72,784196
720	1,94068457	478,06	4059,5	50,938403
750	1,455208965	358,47	4059,5	72,780543
780	3,39682722	836,76	4059,5	72,781152
810	1,45524956	358,48	4059,5	65,509978
840	2,912082325	717,35	4059,5	72,783587
870	1,940156835	477,93	4059,5	73,074451
900	2,93147325	717,18	4087,5	73,284992
930	1,954192875	478,09	4087,5	73,293576
960	2,9320455	717,32	4087,5	58,633144
990	0,97683075	238,98	4087,5	

Tabelle A.2: Base64-Kodierung mit zwei Threads

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
0	1,4542958	358,51	4056,5	71,62824
30	3,3209202	836,82	3968,5	120,9694
60	4,7437068	1195,34	3968,5	113,92106
90	2,8510306	717,06	3976	92,665054
120	3,3266397	836,68	3976	92,708077
150	2,8538988	717,06	3980	142,70807
180	6,6599728	1673,36	3980	142,68416

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
210	2,8523044	717,2	3977	92,705063
240	3,3280331	836,82	3977	92,699694
270	2,8519465	717,11	3977	92,680008
300	3,3267207	836,49	3977	71,428169
330	1,4351572	358,61	4002	71,775555
360	3,3498798	836,32	4005,5	71,79318
390	1,4363322	358,59	4005,5	50,274633
420	1,9153099	478,17	4005,5	50,270427
450	1,4360519	358,52	4005,5	64,642161
480	2,8734255	717,37	4005,5	93,567547
510	3,3644109	836,71	4021	100,93534
540	3,364612	836,76	4021	95,15596
570	2,9791187	740,89	4021	95,07936
600	3,3595053	835,49	4021	71,367121
630	1,3983028	347,75	4021	

Tabelle A.3: Base64-Kodierung mit drei Threads

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
0	1,6419644	398,39	4121,5	75,060991
30	3,3621017	836,76	4018	115,25385
60	4,3214886	1075,8	4017	115,16531
90	3,3561985	836,54	4012	124,61148
120	4,9512337	1232,57	4017	123,26727
150	3,2665842	813,19	4017	93,312701
180	2,9542625	735,44	4017	114,61164
210	4,6865134	1166,67	4017	143,86168

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
240	4,9042651	1221,79	4014	122,87777
270	3,2875864	819,03	4014	114,09434
300	4,3187027	1075,91	4014	115,01304
330	3,3488336	836,79	4002	101,28922
360	3,403781	850,52	4002	93,422288
390	2,8243715	705,74	4002	93,875879
420	3,4340204	848,43	4047,5	103,02061
450	3,4340204	848,43	4047,5	94,463186
480	2,8635253	707,48	4047,5	72,372799
510	1,961328	477,79	4105	

Tabelle A.4: Base64-Kodierung mit vier Threads

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
0	3,5440048	838,52	4226,5	96,918029
30	2,9171972	716,58	4071	116,00246
60	4,8163002	1195,26	4029,5	144,48659
90	4,816139	1195,22	4029,5	122,81312
120	3,3714021	836,68	4029,5	144,50109
150	6,2620042	1554,04	4029,5	144,50472
180	3,3716438	836,74	4029,5	93,842127
210	2,884498	717,18	4022	115,37268
240	4,807014	1195,18	4022	122,38665
270	3,3520959	836,56	4007	93,709133
300	2,8951796	717,25	4036,5	94,09021
330	3,377501	836,74	4036,5	94,075678
360	2,8942109	717,01	4036,5	94,444106

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
390	3,4020629	836,71	4066	73,181047
420	1,4766736	358,59	4118	44,300209
450	1,4766736	358,59	4118	

Tabelle A.5: Base64-Kodierung mit fünf Threads

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
0	1,48701917	359,14	4140,5	94,242991
30	4,79584692	1194,78	4014	143,91153
60	4,79825532	1195,38	4014	122,34552
90	3,3581124	836,6	4014	122,33107
120	4,79729196	1195,14	4014	115,13356
150	2,87827884	717,06	4014	115,30456
180	4,8086919	1195,3	4023	122,69369
210	3,37088766	836,76	4028,5	122,8056
240	4,81615232	1195,52	4028,5	116,68611
270	2,962921465	735,49	4028,5	116,53141
300	4,80583936	1192,96	4028,5	121,51547
330	3,295192145	817,97	4028,5	94,349911
360	2,99480194	731,51	4094	73,704896
390	1,9188578	468,7	4094	51,134879
420	1,49013412	363,98	4094	

Tabelle A.6: Base64-Kodierung mit sechs Threads

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
0	1,7253043	418,56	4122	97,568271

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
30	4,7792471	1195,26	3998,5	143,36351
60	4,77832	1194,58	4000	114,7068
90	2,8688	717,2	4000	114,81146
120	4,7852974	1194,98	4004,5	121,9241
150	3,3429761	836,79	3995	122,32258
180	4,8118623	1194,9	4027	115,50342
210	2,8883658	717,25	4027	115,53242
240	4,8137953	1195,38	4027	122,73511
270	3,3685452	836,49	4027	122,74992
300	4,8147826	1195,18	4028,5	115,5323
330	2,8873704	717,27	4025,5	93,81146
360	3,3667269	836,35	4025,5	72,495113
390	1,4662806	358,46	4090,5	

Tabelle A.7: Base64-Kodierung mit sieben Threads

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
0	3,4421244	834,05	4127	116,60426
30	4,3314929	1075,88	4026	137,10644
60	4,8089362	1194,47	4026	122,52145
90	3,3591606	835,82	4019	121,45647
120	4,7379371	1194,94	3965	113,87274
150	2,8535791	717,25	3978,5	114,67824
180	4,791637	1195,22	4009	122,19191
210	3,3544907	836,74	4009	122,19432
240	4,7917973	1195,26	4009	115,00037
270	2,874894	717,11	4009	93,852742

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
300	3,3819555	836,6	4042,5	72,758018
330	1,468579	358,19	4100	95,495355
360	4,897778	1194,58	4100	

Tabelle A.8: Base64-Kodierung mit acht Threads

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
0	1,9986448	478,66	4175,5	102,08178
30	4,8068071	1194,98	4022,5	143,86909
60	4,7844657	1195,22	4003	115,29143
90	2,9016294	717,16	4046	115,92449
120	4,8266698	1195,46	4037,5	123,85654
150	3,4304327	853,66	4018,5	122,11257
180	4,7104053	1172,18	4018,5	114,48111
210	2,9216687	725,61	4026,5	93,773159
240	3,3298752	826,99	4026,5	93,406207
270	2,8972052	717,13	4040	95,228804
300	3,4513818	838,63	4115,5	103,29823
330	3,4351667	834,69	4115,5	73,653663
360	1,4750775	358,42	4115,5	

Tabelle A.9: Base64-Kodierung mit neun Threads

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
0	1,49514709	359,54	4158,5	95,144258
30	4,84780347	1195,66	4054,5	123,60001
60	3,392197425	836,65	4054,5	123,58132

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
90	4,846557	1195,5	4054	116,27285
120	2,904966015	717,01	4051,5	115,42281
150	4,78988766	1195,38	4007	122,07634
180	3,34853523	836,82	4001,5	94,27973
210	2,936780125	717,25	4094,5	117,45913
240	4,89382829	1195,22	4094,5	124,7979
270	3,42603193	836,74	4094,5	73,413566
300	1,46820581	358,58	4094,5	73,39944
330	3,425090195	836,51	4094,5	73,388385
360	1,4674688	358,4	4094,5	

Tabelle A.10: Base64-Kodierung mit zehn Threads

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
0	1,4831672	358,99	4131,5	94,356503
30	4,8072663	1195,54	4021	144,2011
60	4,8061405	1195,26	4021	122,49566
90	3,360237	836,4	4017,5	122,33737
120	4,7955878	1195,46	4011,5	143,70801
150	4,784946	1195,34	4003	115,36205
180	2,9058573	717,23	4051,5	116,48756
210	4,85998	1195,42	4065,5	123,91217
240	3,4008314	836,51	4065,5	94,889712
270	2,9251494	717,3	4078	95,145713
300	3,4178981	836,49	4086	73,236647
330	1,464545	358,43	4086	43,936349
360	1,464545	358,43	4086	

Tabelle A.11: Base64-Kodierung mit 11 Threads

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
0	1,4939819	358,57	4166,5	53,453075
30	2,0695564	511,57	4045,5	81,261329
60	3,3478655	835,4	4007,5	122,07526
90	4,7904854	1195,38	4007,5	122,13338
120	3,3517401	836,68	4006	114,94202
150	4,3110616	1075,48	4008,5	115,51974
180	3,3902547	836,79	4051,5	144,88576
210	6,2687957	1553,99	4034	144,67154
240	3,3759739	836,88	4034	122,97205
270	4,8221629	1195,38	4034	115,94416
300	2,9074477	717,18	4054	95,005176
330	3,4262307	836,38	4096,5	73,420545
360	1,4684724	358,47	4096,5	

Tabelle A.12: Base64-Kodierung mit 12 Threads

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
0	2,1153673	511,02	4139,5	96,987013
30	4,3504336	1076,84	4040	115,95931
60	3,3801872	836,68	4040	101,49974
90	3,3864623	836,68	4047,5	102,12841
120	3,4220987	836,29	4092	95,260681
150	2,9286134	717,27	4083	116,77505
180	4,8563897	1194,39	4066	146,20514
210	4,8906194	1195,02	4092,5	124,42649

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
240	3,4044801	836,79	4068,5	116,5175
270	4,363353	1075,91	4055,5	116,21623
300	3,3843959	834,52	4055,5	72,846117
330	1,472012	358,59	4105	51,815739
360	1,9823707	477,91	4148	59,47112
390	1,9823707	477,91	4148	

Tabelle A.13: Base64-Kodierung mit 13 Threads

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
0	1,9986448	478,66	4175,5	102,08178
30	4,8068071	1194,98	4022,5	143,86909
60	4,7844657	1195,22	4003	115,29143
90	2,9016294	717,16	4046	115,92449
120	4,8266698	1195,46	4037,5	123,85654
150	3,4304327	853,66	4018,5	122,11257
180	4,7104053	1172,18	4018,5	114,48111
210	2,9216687	725,61	4026,5	93,773159
240	3,3298752	826,99	4026,5	93,406207
270	2,8972052	717,13	4040	95,228804
300	3,4513818	838,63	4115,5	103,29823
330	3,4351667	834,69	4115,5	73,653663
360	1,4750775	358,42	4115,5	

Tabelle A.14: Base64-Kodierung mit 14 Threads

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
0	0,3662014	88,54	4136	49,107698
30	2,9076451	716,61	4057,5	116,14957
60	4,8356598	1194,58	4048	123,57061
90	3,4023806	835,35	4073	124,61591
120	4,905347	1195,26	4104	117,25119
150	2,9113989	717,27	4059	145,34967
180	6,7785788	1673,31	4051	145,57678
210	2,9265401	716,85	4082,5	94,606792
240	3,3805794	836,57	4041	93,689882
270	2,8654128	717,16	3995,5	115,22815
300	4,816464	1193,08	4037	101,55225
330	1,9536861	478,2	4085,5	73,741448
360	2,9624104	716,77	4133	74,061707
390	1,9750367	477,87	4133	

Tabelle A.15: Base64-Kodierung mit 15 Threads

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
0	3,0186006	718,8	4199,5	96,349872
30	3,4047242	836,85	4068,5	123,99581
60	4,8616634	1195,54	4066,5	123,93655
90	3,4007733	836,29	4066,5	115,81496
120	4,3202241	1074,95	4019	136,46278
150	4,7772943	1195,22	3997	121,82916
180	3,3446496	836,79	3997	121,99662
210	4,7884581	1195,62	4005	114,5905
240	2,8509089	717,3	3974,5	92,652802

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
270	3,3259446	836,4	3976,5	100,08025
300	3,3460722	836,1	4002	93,525239
330	2,8889438	717,75	4025	94,325619
360	3,3994308	836,68	4063	58,273171
390	0,4854472	119,48	4063	

Tabelle A.16: Base64-Kodierung mit 16 Threads

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
0	0,6952572	167,29	4156	53,901095
30	2,8981491	716,92	4042,5	115,38239
60	4,7940101	1194,62	4013	122,27109
90	3,3573962	836,63	4013	122,26869
120	4,7938495	1194,58	4013	115,08261
150	2,8783243	717,25	4013	115,25181
180	4,8051295	1194,86	4021,5	122,53776
210	3,3640548	835,79	4025	122,62706
240	4,8110825	1195,3	4025	115,45407
270	2,8858554	717,25	4023,5	115,54713
300	4,8172869	1195,06	4031	122,86125
330	3,3734633	836,88	4031	93,947494
360	2,889703	716,87	4031	72,715504
390	1,9579973	477,91	4097	

A.2 Messwerte der Mergesort-Implementierungen

Tabelle A.17: rekursiver Mergesort

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
0	1,74560854	418,21	4174	52,340027
30	1,7437266	418,21	4169,5	48,3180287
60	1,47747532	358,48	4121,5	51,7176124
90	1,97036551	478,07	4121,5	73,8754146
120	2,95466214	716,89	4121,5	66,4876259
150	1,47784626	358,57	4121,5	51,7151395
180	1,96982971	477,94	4121,5	73,8809786
210	2,95556887	717,11	4121,5	73,8822151
240	1,96991214	477,96	4121,5	51,7108119
270	1,47747532	358,48	4121,5	44,3205503
300	1,47722803	358,42	4121,5	73,6576697
330	3,43328328	836,57	4104	73,5635844
360	1,47095568	358,42	4104	44,13852
390	1,47161232	358,58	4104	51,5004804
420	1,96175304	478,01	4104	73,5598908
450	2,94223968	716,92	4104	73,5629688
480	1,96195824	478,06	4104	51,4930932
510	1,47091464	358,41	4104	73,5592752
540	3,43303704	836,51	4104	73,5660468
570	1,47136608	358,52	4104	44,1403668
600	1,47132504	358,51	4104	73,5586596
630	3,4325856	836,4	4104	73,5586596
660	1,47132504	358,51	4104	44,1403668
690	1,47136608	358,52	4104	51,4955556

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
720	1,96167096	477,99	4104	73,5568128
750	2,94211656	716,89	4104	73,554966
780	1,96154784	477,96	4104	51,6062553
810	1,47886918	358,34	4127	

Tabelle A.18: iterativer Mergesort

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
0	1,53241658	359,09	4267,5	52,8862313
30	1,99333218	477,96	4170,5	52,327472
60	1,49516596	358,51	4170,5	67,0483405
90	2,97472341	717,06	4148,5	74,3662184
120	1,98302449	478,01	4148,5	52,0445919
150	1,48661498	358,35	4148,5	44,6108948
180	1,48744468	358,55	4148,5	74,3487947
210	3,46914164	836,24	4148,5	74,3469279
240	1,48732022	358,52	4148,5	52,0570374
270	1,98314894	478,04	4148,5	74,3680853
300	2,97472341	717,06	4148,5	66,9300322
330	1,48727874	358,51	4148,5	52,0564151
360	1,98314894	478,04	4148,5	51,9387258
390	1,47943278	358,52	4126,5	73,9730833
420	3,45210611	836,57	4126,5	73,9699884
450	1,47922646	358,47	4126,5	44,3743178
480	1,4790614	358,43	4126,5	73,98113
510	3,45301394	836,79	4126,5	73,9842248
540	1,47926772	358,48	4126,5	44,3774126

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
570	1,47922646	358,47	4126,5	51,771069
600	1,97217815	477,93	4126,5	74,235791
630	2,97687459	717,06	4151,5	74,416883
660	1,98425094	477,96	4151,5	52,0859645
690	1,48814669	358,46	4151,5	44,636928
720	1,48764851	358,34	4151,5	

Tabelle A.19: paralleler Mergesort

Δ t in s	P in Watt	I in mA	U in mV	Δ W in Ws
0	3,53301535	838,3	4214,5	97,25161515
30	2,95042566	717,08	4114,5	95,81293905
60	3,43710361	836,38	4109,5	122,3209869
90	4,71762885	1149,94	4102,5	114,9022046
120	2,94251813	717,25	4102,5	95,60773688
150	3,431331	836,4	4102,5	102,7280178
180	3,41720352	836,32	4086	124,1934782
210	4,86236169	1194,39	4071	116,6696351
240	2,91561398	717,16	4065,5	65,9230428
270	1,47925554	358,26	4129	44,3776662
300	1,47925554	358,26	4129	