Cloud Parallelizer

A High-Performance Backend for Distributed Task Parallelization

Technische Dokumentation

VERSION: 6.06

Betreuer

Prof. Dr. Luc Bläser

Studenten

Christoph Amrein

Timothy Markiewicz

Abstract

Im Zuge eines Forschungsprojekts des Instituts für Software der Hochschule Rapperswil wurde ein Framework zur .NET Task-Parallelisierung in der Cloud entwickelt. Das entwickelte Framework ermöglicht es, Tasks aus der .NET Task Parallel Library automatisch über einen Web-Dienst auf entfernte Rechenkerne zu verteilen und so die Anwendung zu beschleunigen. Hinter dem Dienst wurde bis anhin ein HPC Cluster benutzt, welcher jedoch in verschiedener Hinsicht schwerfällig und ineffizient ist. Einerseits muss der Cluster Eingaben und Ausgaben über Dateien austauschen und andererseits müssen Anfragen auf Jobs mit fixer Anzahl Cores abgebildet werden, was zu einer schlechten Lastverteilung führt.

Ziel dieser Arbeit war es daher, ein neues Backend zu entwickeln für eine mögliche Ablösung des bestehenden. Dazu sollte eine native Backend-Architektur ohne Cluster-Middleware entwickelt werden, welche die Last von Anfragen automatisch optimal auf eine beliebige Anzahl heterogener Rechnerknoten hinter dem Service verteilt. Zudem sollen zur Laufzeit neue Knoten hinzugefügt bzw. entfernt werden können.

Entwickelt wurde ein komplett neues Backendsystem bestehend aus zwei Komponenten: Einem Master-Service, mit welchem die .NET Anwendungen als Clients kommunizieren und welcher deren auszuführende Tasks verwaltet. Einem Worker-Service pro beteiligtem Rechen-Knoten, welcher vom Master-Service Tasks bezieht und auf den Cores des Knotens ausführt. Das Konzept entspricht einem verteilten Thread Pool. Aufgrund des Pull-Prinzips der Worker wird automatisch eine optimale Auslastung bei genügender Anzahl Tasks auf eine dynamische Anzahl heterogener Rechnerknoten erreicht. Zudem entfällt eine unnötige Serialisierung in Files. Die HTTPS-Kommunikation mit dem Master-Service basiert beim Warten auf einem Polling-Prinzip mit adaptiven Frequenzen.

Die entwickelte Lösung wurde auf einer Umgebung mit 24 Rechnern zu je 4 Cores (insgesamt 96 Cores) experimentell evaluiert und mit dem HSR Cluster verglichen. Dabei konnte nachgewiesen werden, dass das neue System gegenüber dem Cluster performanter ist, gleichförmiger skaliert und deutlich kleinere Overheads aufweist. Das neue Backend eignet sich somit als idealer Ersatz für das bestehende Cluster-Backend.

Abstract (English)

A recent research project at the University of Applied Sciences Rapperswil implemented a framework for distributed .NET task parallelization in the cloud. This framework enables distributed computation of tasks by sending them to remote computing nodes over a web service. Until now, a HPC Cluster was used behind the web service. The use of a HPC Cluster suffers from several drawbacks concerning efficiency and flexibility. First of all, exchanging input and output data requires the usage of files. Furthermore job requests need to specify the desired amount of cores beforehand, which leads to unfavorable balancing of workload.

As a consequence, the goal of this project was to replace the existing backend with a new one. Therefore, a native backend architecture without cluster middleware needed to be developed. This new backend should be able to optimally balance the amount of workload onto an arbitrary amount of heterogeneous computing nodes which lie behind the service. Another requirement was that the new system should support adding and removing computing nodes at runtime.

Developed was a completely new backend system consisting of two components. A Master-Service which .NET applications communicate with as clients and also manages the tasks. A Worker-Service, for computing nodes, which continuously requests tasks for execution on the local cores. The concept corresponds to a distributed thread pool. Due to the pull principle of workers, optimal workload balancing is automatically achieved, given that there are enough tasks available. Moreover, expensive serialization into files is no longer required. The HTTPS communication with the master service is based on a polling principle with adaptive frequencies.

The developed solution was experimentally evaluated in an environment with 24 machines each having 4 Cores (resulting in 96 cores) and compared to the HSR HPC Cluster. It was proven that the new system is more efficient, has a steadier scale-up and has considerable less overhead. Hence, the new backend is an ideal substitution for the current cluster backend.

Management Summary

EINFÜHRUNG

In der aktuellen Zeit mit dem Trend zu Big Data, SaaS und Real Time Calculation ist es unerlässlich, komplizierte Rechenoperationen schnellstmöglich ausführen zu können. Da ein einzelner Rechner komplexe Rechenoperationen nicht schnell genug ausführen kann, müssen Berechnungen in Echtzeit auf verschiedene Rechner verteilt werden.

Die bisher vorhandene Software zur Parallelisierung von Prozessen auf verschiedenen Rechnern wird den heutigen Ansprüchen nicht mehr gerecht. Neben der komplizierten und schwerfälligen Konfiguration ist die Software aufgrund ihrer Architektur starr, nicht performant genug und lässt sich nicht beliebig skalieren. Da die eingesetzte HPC Cluster-Technologie hohe Betriebskosten verursacht, bieten sich durch eine Ablösung – neben der Vereinfachung und Performancesteigerung – auch finanzielle Vorteile an.

Aus diesen Gründen wurde ein neues, effizienteres, schlankeres und vereinfachtes System entwickelt.

ARCHITEKTUR

Bei der neuen Architektur der Lösung handelt es sich um einen verteilten Thread-Pool. Mit dem Ziel einer hohen Skalier- und Wartbarkeit wurde dieser in drei Kernkomponenten unterteilt: Client, Master, Worker. Der Client möchte eine komplexe Rechenoperation auslagern. Dazu übermittelt er eine Sammlung von Tasks in einem Task-Package an den Master. Dieser wiederum verteilt die Tasks für die Berechnung an beliebig viele Worker-Instanzen. Nach Abschluss eines Tasks wird das Ergebnis vom Worker an den Master übermittelt und schliesslich vom Client entgegengenommen. Für den Transfer zwischen den einzelnen Komponenten wird ein Polling Mechanismus per HTTP(S) verwendet.

BENCHMARKS

Die mit der neu entwickelten Software durchgeführten Benchmarks zeigen auf, dass die automatische Skalierung des Systems einwandfrei funktioniert und mit steigender Anzahl Prozessoren die Ausführungszeit stark abnimmt. Das Minimum wird dann erreicht, wenn die Anzahl Cores die Anzahl auszuführender Tasks überschreitet. Ebenfalls wurde ermittelt, dass komplexe Rechenoperationen mit der neueren, schlankeren Lösung bis zu 50% schneller durchgeführt werden können, als dies mit der bisherigen Software möglich war. Die Ziele der neu entwickelten Lösung wurden gänzlich erfüllt. Eine sofortige Ablösung der alten Software ist sowohl aus finanzieller, als auch technologischer Sicht empfehlenswert.

Inhalt

1. Einführung 7
2. Architektur 8
  2.1 Systemübersicht 9
  2.2 Benutzer Verwaltung 10
    2.2.1 Autorisierung 10
    2.2.2 Quotas 10
  2.3 Client 11
  2.4 Master 11
    2.4.1 Task Queue 11
    2.4.2 Service Schnittstellen 12
    2.4.3 Thread Synchronisation 16
  2.5 Worker 17
    2.5.1 Worker Thread 17
    2.5.2 Kommunikation 17
    2.5.3 Graceful Shutdown 18
    2.5.4 Dynamic Code Memory Leak 18
  2.6 Fehlerfälle 19
3. Benchmarks 20
  3.1 Versuchsaufbau 21
    3.1.1 Neues Backend System 21
    3.1.2 Microsoft HPC 22
  3.2 Benchmark Ergebnisse 23
  3.3 Direktvergleich 24
    3.3.1 Skalierung 24
    3.3.2 Performance 26
    3.3.3 Overhead 27
    3.3.4 Optimierung der Rechenzeit 29
    3.3.5 Fazit 30
4. Mögliche Verbesserungen 32
  4.1 Code Verarbeitung 32
  4.2 Eigener Netzwerk-Stack für Task-Verteilung 32
  4.3 Work Stealing Thread Pool 32
5. Glossar 34
6. Abbildungsverzeichnis 35
7. Codeverzeichnis 35

8. Tabellenverzeichnis 35
9. Literaturverzeichnis 36

1. Einführung

Zurzeit besteht eine Lösung, welche zur verteilten Task-Ausführung ein Microsoft HPC Cluster als Backend verwendet. Bei dieser Lösung werden in .NET entwickelte Tasks transparent an einen WebService übermittelt. Der WebService verteilt diese Tasks schliesslich unter den verwendeten Nodes.

Durch den Push-Mechanismus ergibt sich der Nachteil, dass im vornherein die Anzahl an Nodes und Cores angegeben werden muss. Dabei dürfen wiederum nicht zu viele Nodes bzw. Cores angegeben werden, da die Ausführung erst genau dann gestartet wird, wenn die geforderte Anzahl vorhanden ist. Im schlimmsten Fall kann es dazu führen, dass z.B. 100 Cores verlangt werden, aber nur 99 verfügbar sind und somit die ganze Ausführung wartet. Des Weiteren ist der HPC Cluster File und Batch basiert. Aus diesem Grund müssen sämtliche Tasks vor deren Ausführung serialisiert, in Dateien geschrieben und wiederrum gelesen und deserialisiert werden. Am Ende muss die ganze Prozesskette erneut durchlaufen werden. Diese Umwandlung führt zu einem vermeidbaren Zeitverlust.

Um die erläuterten Nachteile zu beheben, soll eine neue Backend-Architektur entworfen und implementiert werden. Anstelle eines Dateiaustausches soll der Datentransfer über eine Netzwerkverbindung geschehen. Des Weiteren ist die optimale Verteilung der Rechenleistung ein wichtiges Ziel. So sollen Nodes dynamisch dem System angehängt und sofort verwendet werden können, ohne dass der Benutzer die Anzahl an verwendeter Nodes und Cores angeben muss.

2. Architektur

Bei der Architektur handelt es sich um einen verteilten Thread Pool [1], der ohne Cluster-Middleware operiert. In Abbildung 1 wird gezeigt, dass der verteilte Thread Pool wie ein lokaler Thread Pool aufgebaut ist, nur dass die einzelnen Komponenten auf verschiedenen Maschinen ausgeführt werden.

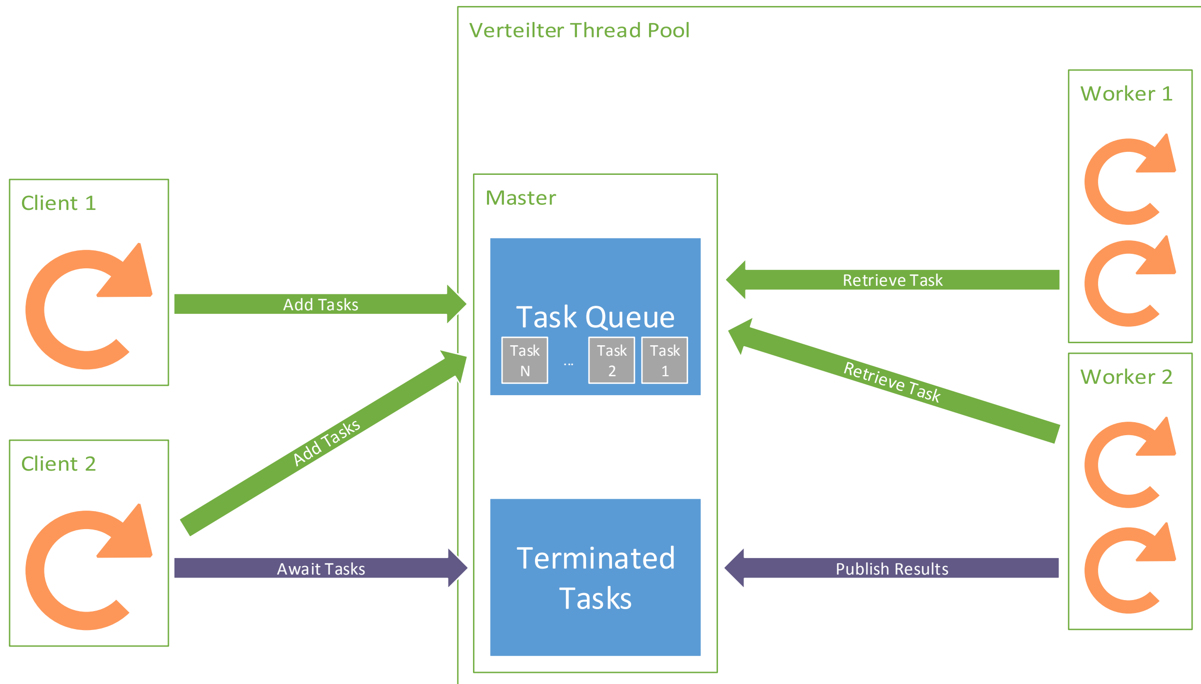

Abbildung 1 Verteilter Thread Pool

Ein Mapping der Komponenten eines lokalen Thread Pools auf einen verteilten Thread Pool wurde folgendermassen realisiert:

	Lokaler Thread Pool	**Verteilter Thread Pool**
Task Ersteller / Starter	Application Thread	Client
Task Verteiler	Thread Pool Interface	Master
Task Ausführer	Worker Thread	Worker

Tabelle 1 Mapping der Thread Pool Komponenten

Ein Task ist eine Kombination von Programmcode (*Portable Task Code*) und den zugehörigen Daten (*Portable Task Data*). Ein solcher Task ist die ausführbare Einheit im System und gehört zu einem Task Package. Ein Task Package ist eine Menge von zusammenhängenden Tasks. Sind alle Tasks eines Task Packages ausgeführt, so ist dieses abgeschlossen. Ein Task erhält eine – im Task Package eindeutige – aufsteigende Nummer, um diesen identifizieren zu können. Da die Resultate in einem Task Package entsprechend der Task-Order geordnet sein müssen, wird die Nummer zur Identifikation gleichzeitig zur Ordnung verwendet.

Ein vollständiger Ablauf für einen einzelnen Task sieht folgendermassen aus:

1. Der Client sendet den Task an den Client Service.
2. Der Client Service trägt den Task in die Task Queue ein.
3. Der Worker fragt beim Worker Service nach einem neuen Task.
4. Der Worker Service holt den Task aus der Task Queue.
5. Der Worker führt den empfangen Task aus.

6. Der Worker sendet die berechneten Daten an den Worker Service.
7. Der Worker Service trägt die Daten bei den Terminated Tasks ein.
8. Der Client Fragt beim Client Service nach den Resultaten.
9. Der Client Service holt die Resultate von den Terminated Tasks.

Der erwähnte Ablauf wird in Abbildung 2 nochmal bildlich dargestellt.

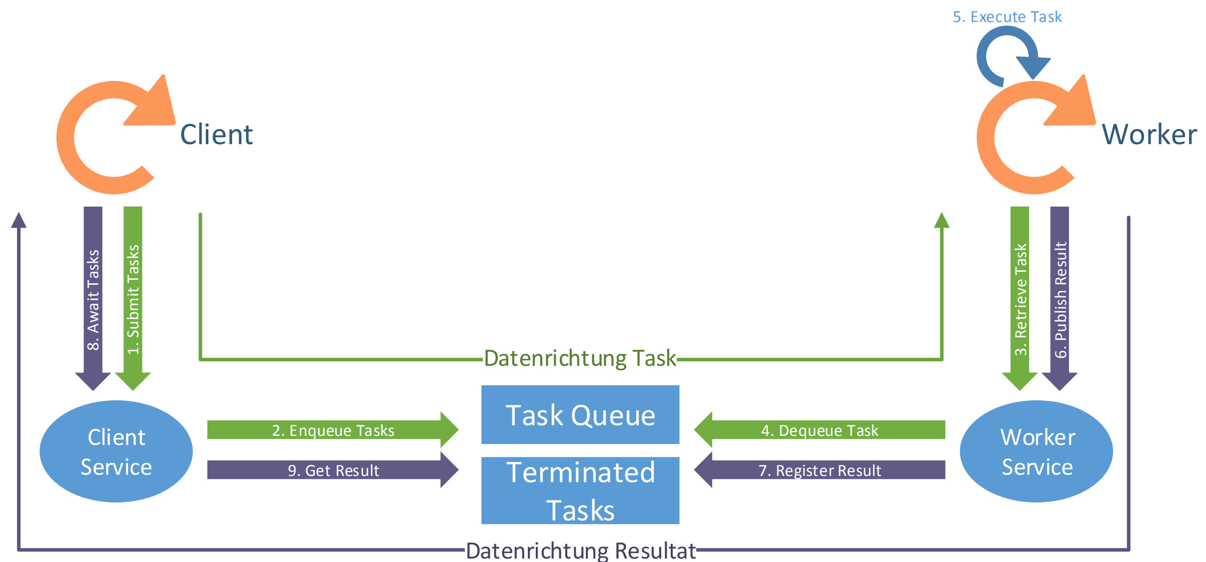

Abbildung 2 Ablauf der Taskausführung im Pool

Um einen Datenaustausch zwischen den Komponenten zu ermöglichen, werden sämtliche Tasks, deren zugehörigen Daten und auch die berechneten Resultate jeweils serialisiert und über das Netzwerk ausgetauscht. Da ausserhalb der Prozesse bzw. über das Netzwerk keine konkreten Objekt-Referenzen gehalten werden können, werden Referenzen intern verwaltet. Das heisst jedem Task werden eindeutige IDs zugewiesen, welche dann intern gemappt werden.

2.1 Systemübersicht

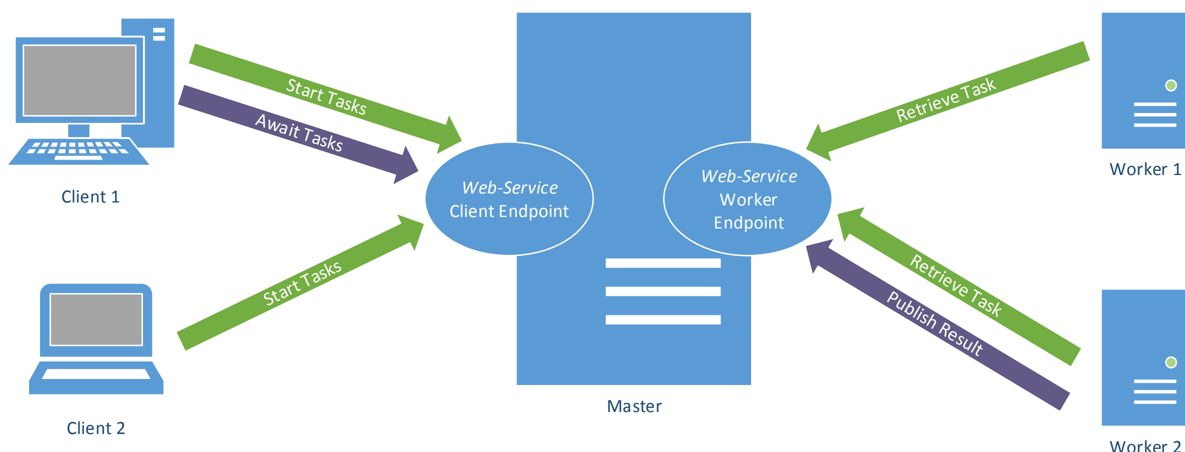

Abbildung 3 Systemübersicht

Durch die Adaption des Thread-Pool Konzepts auf das angestrebte System ergibt sich ein Aufbau, welcher grundsätzlich aus drei Komponenten besteht:

- Ein Client, welcher die Tasks an den Pool übermittelt.

- Ein Master, welcher die Rolle der *Task Queue* und der *Terminated Tasks Dictionaries* übernimmt. Der Master stellt zudem die Web-Services bereit.
- Ein Worker, welcher für die Ausführung der Tasks zuständig ist.

2.2 Benutzer Verwaltung

Für die Autorisierung der Clients und die Überwachung ihrer Rechenzeit wird eine Benutzer Verwaltung implementiert.

Einerseits dürfen keine unberechtigten Clients Zugriff erhalten, andererseits dürfen diese auch nicht mehr Rechenzeit verwenden als zugewiesen.

Mit einem solchen Benutzer Management ist es auch einfach möglich, das System als kommerziellen Dienst anzubieten. Die Benutzer können sich dabei Rechenzeit kaufen und danach für ihre Tasks verwenden.

2.2.1 Autorisierung

Damit keine unberechtigten Benutzer Zugriff auf den verteilten Thread-Pool erhalten, ist eine einfache Autorisierung notwendig. Um diesen Mechanismus möglichst simpel zu halten, wird mit einfachen Autorisierungs-Tokens gearbeitet.

Wird einem Benutzer Zugriff gewährt, so wird diesem ein Token zur Verfügung gestellt, mit welchem er sich autorisieren kann. Dieser Token wird bei jedem Service-Request mitgegeben und auf Server-Seite gegen die registrierten Tokens geprüft. Ist der Token ungültig, wird ein Fehler zurückgegeben.

2.2.2 Quotas

Für die Begrenzung der Rechenzeit wird ein Quota-System verwendet, welches vom Master verwaltet wird. Der Benutzer erhält bei der Token-Vergabe eine vorgegebene Summe an Rechenzeit, welche frei verwendet werden kann. Der Wert verringert sich immer, wenn ein Resultat beim Master eingetroffen ist. Der Worker liefert bei jedem Resultat immer die benötigte Ausführungszeit mit, welche der Master von der verbleibenden Rechenzeit abzieht. Dabei ist es irrelevant, ob der Task erfolgreich ausgeführt wurde oder nicht.

PERSISTIERUNG

Die Quotas werden in einer MS SQL Datenbank persistiert und über die Token des Clients identifiziert. Als ORM wurde das Entity Framework von Microsoft verwendet. Bei jeder Änderung der Quotas wird der aktuelle Wert in die Datenbank geschrieben. Somit ist immer nur ein Wert pro Client in der Datenbank hinterlegt. Dadurch wird verhindert, dass die Datenbank-Grösse mit jedem Update zunimmt. Die Aktualisierung der Datenbank wird vom Master durchgeführt. Um den Overhead zu verringern, werden die Quotas bei Systemstart einmalig geladen und danach nur noch aktualisiert.

TERMINIERUNG

Versucht ein Benutzer ein neues Task-Package zu starten, ohne dass diesem noch Rechenzeit zur Verfügung steht, wird direkt ein Fehler zurückgegeben. Neue Tasks werden in solchen Fällen erst gar nicht in die Task-Queue eingereiht.

Sollte ein Resultat dazu führen, dass die verbleibende Quota unter 0 fällt, wird das Task Package als fehlerhaft markiert und die verbleibenden Tasks werden ohne Ausführung aus der Task-Queue entfernt.

Während der Ausführung eines Tasks überwacht der Worker die Ausführungszeit des Tasks und bricht die Ausführung ab, sollte die Zeit die verbleibende Quota überschreiten. In diesem Fall wird ein mit

Error markiertes Resultat dem Master übermittelt und die Weiterverarbeitung des Task Packages abgebrochen. Die bisher aufgewendete Rechenzeit wird auch in diesem Fall dem Client abgezogen.

2.3 Client

Der Client ist der Anwender des Systems. Mithilfe der bestehenden Client Runtime werden der Code und die dazugehörigen Daten des Tasks an den Master übermittelt. Nach Beendigung des Tasks werden die Resultate wiederum entgegengenommen.

Da der Client kein Bestandteil dieser Arbeit ist, wird dieser nicht detaillierter beschrieben [2].

2.4 Master

Der Master fungiert als Bindeglied zwischen Client und Worker. Er stellt jeweils passende Web-Services zur Verfügung, mit welchen die Clients die Tasks eintragen und die Worker neue Tasks entgegen nehmen können.

2.4.1 Task Queue

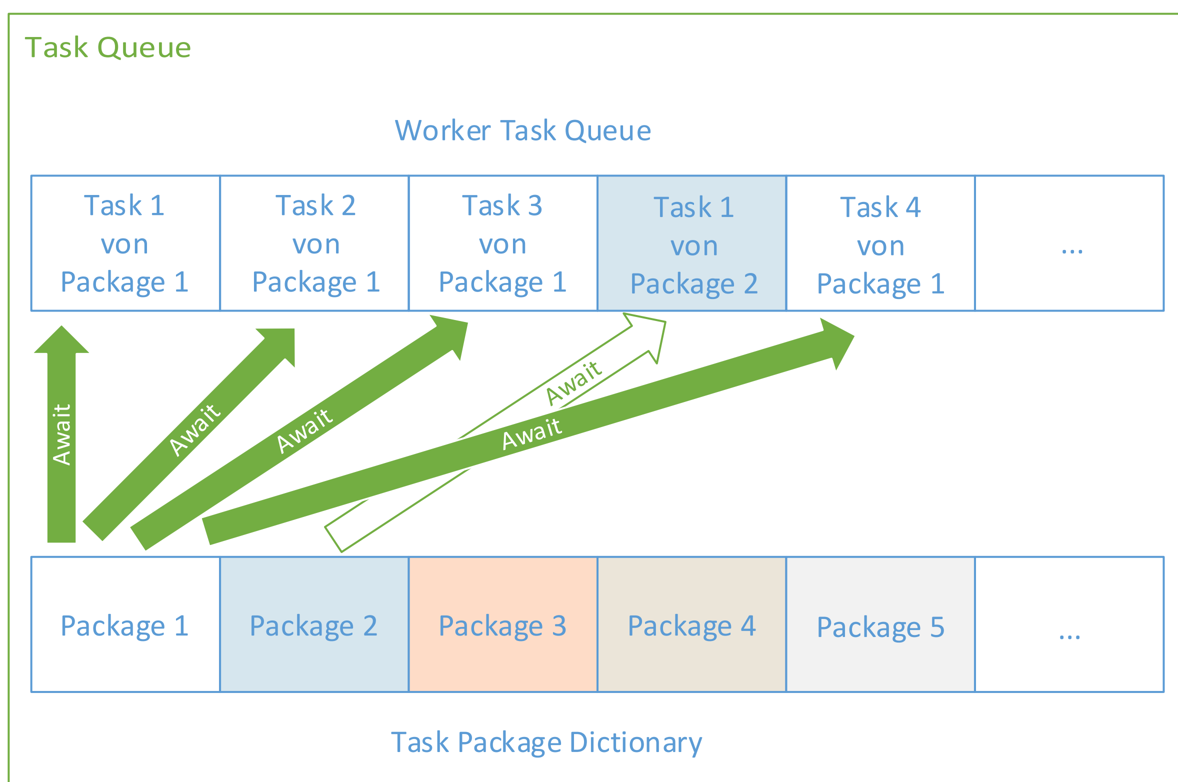

Abbildung 4 Aufbau der Task Queue

Intern ist die Task Queue in zwei Collections aufgeteilt. Empfängt der Master ein Task Package werden die beinhalteten Tasks in Worker Tasks umgewandelt und in eine Worker Queue eingereiht. Das Task Package wird in ein Dictionary abgelegt und der Client wartet auf die Fertigstellung der assoziierten Tasks aus der Worker Queue. Übermittelt ein Worker ein Resultat (Task Result), wird dieses im Task Package abgelegt. Weiter wird geprüft, ob nun alle Tasks des Task Packages ausgeführt wurden. Ist dies der Fall, wird das Task Package aus der entsprechenden Queue entnommen und als Resultat dem Client zur Abfrage bereitgestellt.

Aufgrund der einfachen Worker Task Queue funktioniert das System nach dem *First-Come-First-Served* Prinzip. So ergeben sich keine Fairness Probleme, insbesondere keine Starvation. Da Tasks Pull-Based geholt werden, skaliert das System linear mit der Anzahl Worker. Zudem können Worker welche

schneller sind, öfters neue Tasks aus der Queue holen als langsame und werden so auch maximal ausgelastet (Best Effort). So wird schliesslich auch keine homogene Infrastruktur vorausgesetzt.

2.4.2 Service Schnittstellen

Für die Kommunikation mit den Clients und den Workern wird auf WCF [3] gesetzt. Dadurch ist es sehr einfach möglich, den Service als WebService [4] anzubieten. Durch die HTTP(S) Unterstützung sind keine speziellen Firewall Ausnahmen notwendig.

2.4.2.1 Client Service

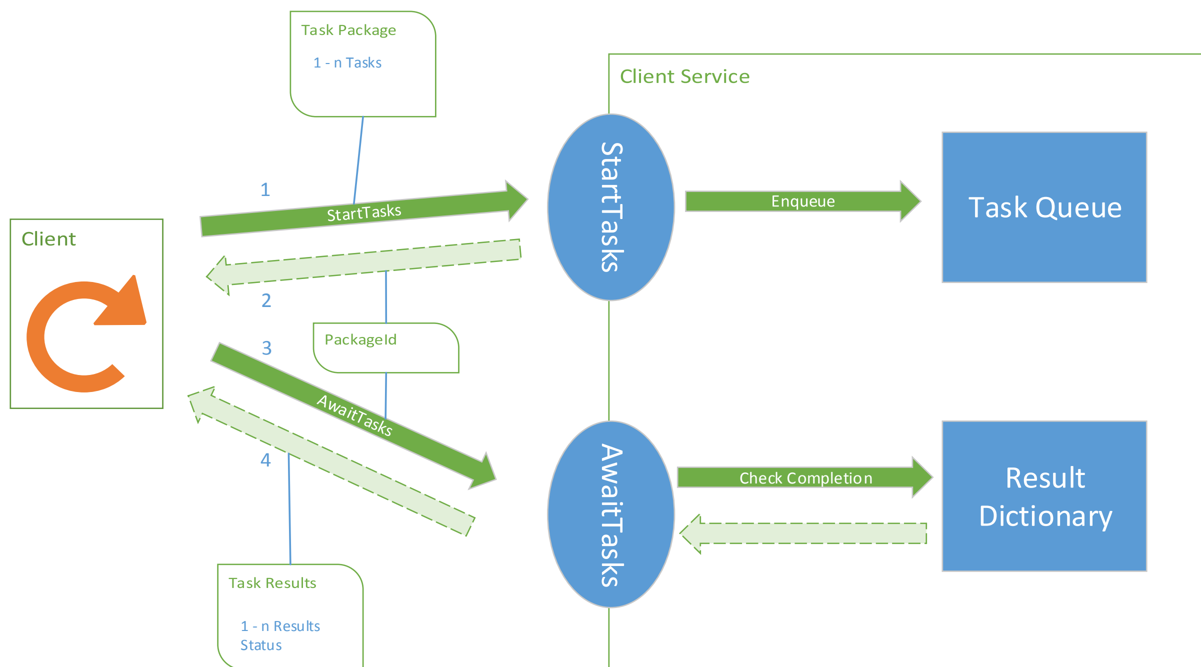

Abbildung 5 Client Service

Der Client Service des Masters bietet zwei Service-Methoden an. *StartTasks* nimmt ein Task Package entgegen und legt dessen Tasks in der internen Task Queue ab. Das Task Package wird zusätzlich in das *Package Dictionary* abgelegt. Diese Zwischenspeicherung dient dazu, sämtliche Resultate der Worker zu sammeln und aufzubereiten.

Da nicht einfach eine Referenz auf das *Task Package* zurückgegeben werden kann, wird eine *Package Id* erstellt, welche das *Task Package* in dem *Package Dictionary* eindeutig identifiziert.

Die zweite Service-Methode ist *AwaitTasks*. Diese nimmt die von *StartTasks* zurückgegebene *Package Id* entgegen. Sollten noch nicht sämtliche Tasks des Packages abgeschlossen worden sein, wird der Status *AwaitTasksRunning* mit einem Poll-Delay zurückgegeben. Dieser Status teilt dem Client mit, dass der Service nach dem gegebenen Poll-Delay erneut aufgerufen werden muss. Sobald ein Task erfolgreich abgeschlossen wurde, wird der Status *AwaitTasksCompleted* inklusive sämtlicher Resultate zurückgegeben. Ist ein Task während der Ausführung fehlgeschlagen, wird der Status *AwaitTasksError* zurückgegeben.

Aus Entwicklersicht sehen die beiden Client-Services wie folgt aus.

```
Interface ClientService {
      StartTasks(TaskPackage, AuthorizationToken)
            returns PackageId;
      AwaitTasks(PackageId, AuthorizationToken)
            returns AwaitTasksStatus;
```

```
}
```

Code 1 Client Service Interface

2.4.2.2 Worker Service

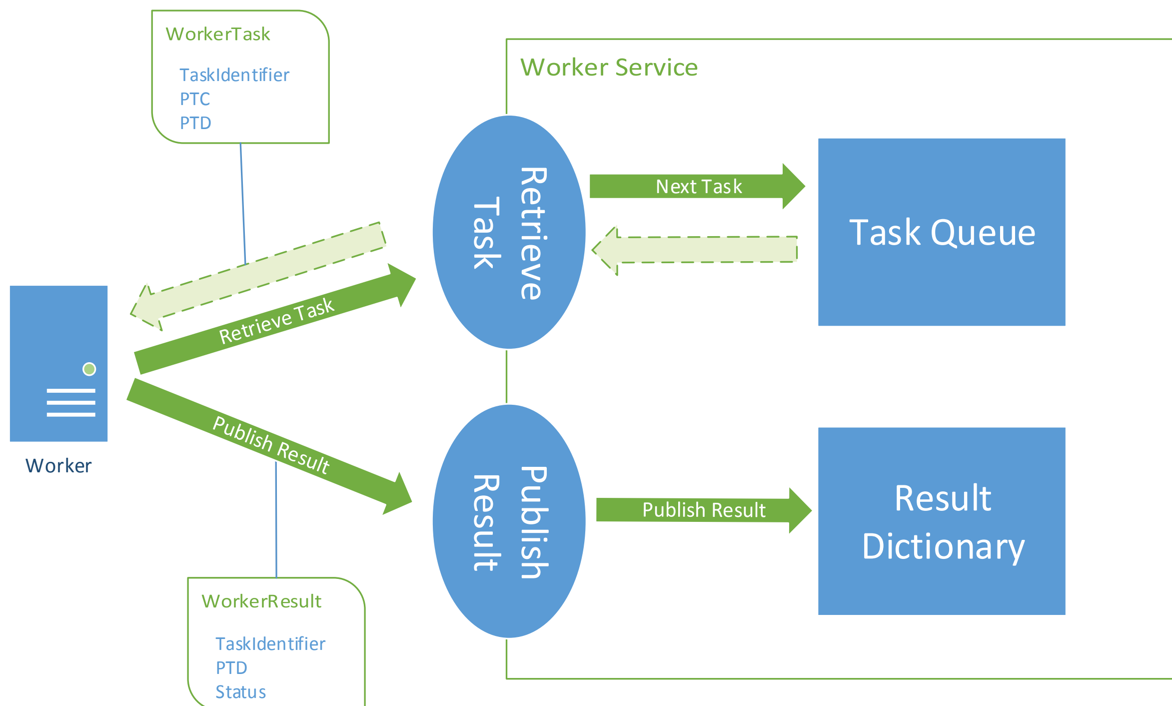

Abbildung 6 Worker Service Schnittstelle

Worker erhalten beim Aufruf von *RetrieveTask* einen *RetrieveTaskStatus*. Steht kein Task zur Ausführung bereit, wird der Status *RetrieveTaskRetry* zurückgegeben. Ansonsten wird der Status *RetrieveTaskAvailable* mit einem *WorkerTask* übermittelt. Der *WorkerTask* enthält den *Portable Task Code* sowie *Portable Task Data*. Des Weiteren enthält ein *WorkerTask* einen *TaskIdentifier,* welcher den Task eindeutig identifiziert.

Das Resultat eines *Workers* ist ein *TaskResult*. Dieses beinhaltet nur noch die aus der Ausführung resultierende *Portable Task Data (PTD)* sowie den *TaskIdentifier,* mit welchem das Resultat eindeutig mit einem Task assoziiert werden kann. Das *TaskResult* wird dann über die *PublishResult* Methode des Services an den Master übermittelt.

Aus Entwicklersicht sehen die beiden Worker-Services wie folgt aus.

```
Interface WorkerService {
      RetrieveTask(AuthorizationToken)
            returns RetrieveTaskStatus;
      PublishResult(TaskResult, AuthorizationToken);
}
```

Code 2 Worker Service Interface

Zu beachten ist, dass die Service-Methode *RetrieveTasks* Tasks überspringt, welche von Clients stammen, die keine verbleibende Quota mehr haben. Wird vom WorkerService ein solcher Task erkannt, wird das Task Package als *Error* markiert und der Client erhält eine Fehlermeldung. Durch das Überspringen der restlichen Tasks des Task Packages wird die Taskqueue von den nicht mehr ausführbaren Tasks bereinigt.

2.4.2.3 Adaptive Request Delays

Client und Worker Fragen beim Master nach dem Polling Prinzip an. Dabei wird bei jedem ergebnislosen Request der Delay zwischen den Requests erhöht. Die Delays überschreiten dabei nicht ein vorgegebenes Maximum, um dennoch eine gewisse Reaktionszeiten beibehalten zu können. Zu tiefe Polling-Zeiten würden – bei grosser Worker-Menge – einen ähnlichen Effekt erzeugen wie eine DDoS-Attacke.

Durch solche, sich wiederholenden Requests werden Spezial-Konfigurationen und Verhalten entgegen dem HTTP-Standard vermieden. So ist es möglich, auch nur mit einem vorhandenen Request-Thread die gesamte Tätigkeit durchzuführen, wenn auch nur noch synchron. Das heisst, mehr Threads dienen primär der Durchsatzsteigerung und nicht als Voraussetzung, damit das System einwandfrei funktioniert. Durch diese Eigenschaft verhält sich der verteilte Thread-Pool gleich einem lokalen Thread Pool.

Der verteilte Thread-Pool ist darauf ausgerichtet, grosse und aufwändige Tasks auszuführen. Dadurch wird die verlorene Zeit vom Einreihen des ersten Tasks bis zu dessen Ausführung als weniger problematisch betrachtet. Dennoch gilt zu beachten, dass im schlechtesten Fall direkt nach einem Request ein Task (für Worker) oder das Resultat (für Client) verfügbar wird und somit nochmal der gesamte Delay gewartet wird.

2.4.2.4 Keine blockierenden Requests

Es wird darauf verzichtet, die Requests zu blockieren. Zwar bringen blockierende Requests eine sehr tiefe Reaktionszeit und es werden Idle-Loops vermieden, allerdings stört dies die Skalierung des Systems.

Für jeden offenen Request wird ein Thread auf Master Seite benötigt. Bei einem blockierenden *RetrieveTask* Service wächst die Anzahl an benötigter Threads mit jedem zusätzlichen Worker gravierend. In Abbildung 7 wird das auftretende Problem an einem Beispiel mit drei Request Threads dargestellt.

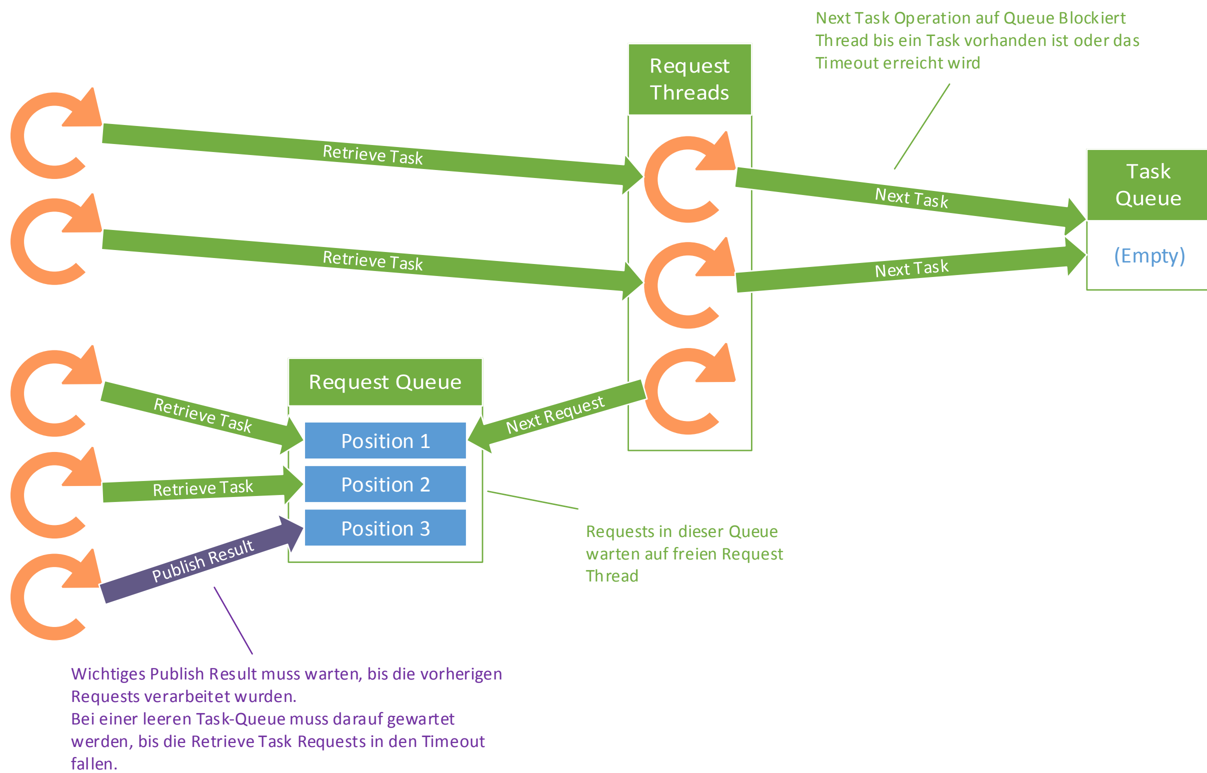

Abbildung 7 Skalierung mit blockierenden Requests

Gibt es nun acht Worker, welche je acht Threads starten, werden 64 Threads alleine für den *RetrieveTask* Service benötigt. Zusätzlich muss beachtet werden, dass wartende Requests nicht solche blockieren, welche eine wichtigere Aufgabe übernehmen (speziell das Senden von Resultaten). Abbildung 8 zeigt, wie *RetrieveTask* Requests einen *PublishResult* Request bei einem Timeout von zehn Sekunden mit zwei Request-Threads verzögern, auch wenn diese innerhalb kurzer Zeit nacheinander beim Master eintreffen.

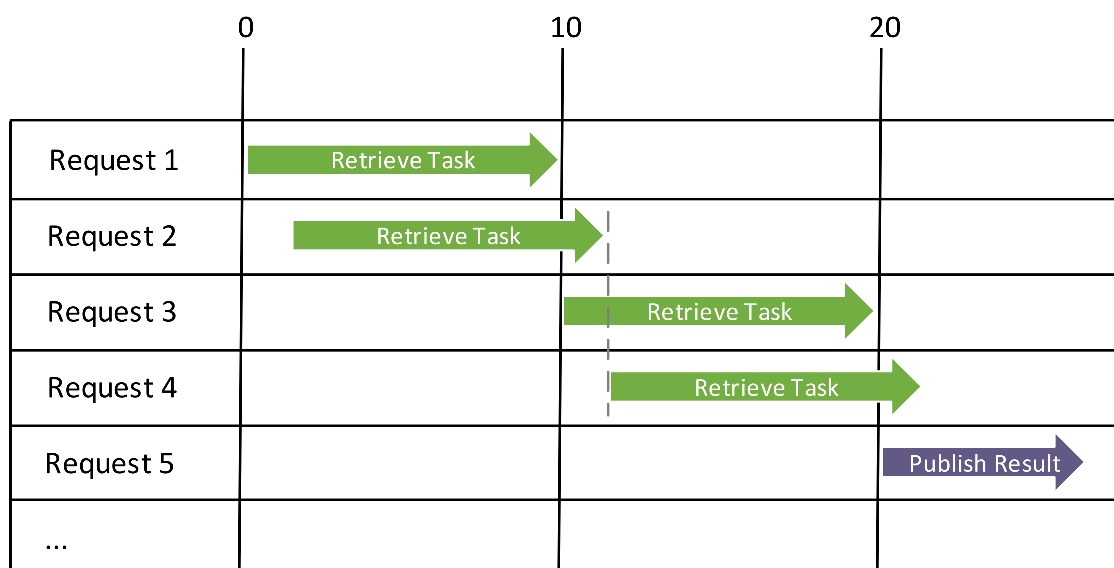

Abbildung 8 Zeitliches Verhalten mit blockierenden Requests

Um ein System mit blockierenden Requests auf WCF- bzw. HTTP-Basis zu unterstützen, sind mehrere Konfigurationen notwendig.

- Es muss erlaubt werden, dass mehrere gleichzeitige Requests von und zu einem Host möglich sind [5].
- Das WCF-Service Throttling muss so konfiguriert werden, dass Requests nicht gequeued werden [6].
- In der *machine.config* der .NET Runtime muss die Anzahl Threads erhöht werden [7]. Diese hängen wiederrum von der Anzahl Worker und Clients ab.

All diese Konfigurationen zeigen, dass die Technologie nicht dazu ausgelegt ist, viele offene Requests zu haben, auch wenn diese keine Aktivität ausführen. Zwar erlaubt WCF die Unterscheidung zwischen den WebServices, leider ist diese Unterscheidung aber nicht auf Ebene des ASP.NET Containers möglich.

Um ein System mit blockierenden Requests bzw. ohne Idle-Loops zu implementieren, würde es sich empfehlen, auf WCF und die .NET HTTP Schnittstellen zu verzichten. Vorstellbar wäre ein eigener Netzwerk-Stack, welcher für einen solchen Mechanismus ausgelegt ist. Im Rahmen dieses Projekts wurde diese Variante allerdings nicht weiter verfolgt.

2.4.3 Thread Synchronisation

Für die Synchronisation der Task Queue und der Client Quotas wird ein Monitor verwendet. Abbildung 9 zeigt, dass es dabei nicht zu einem Dead-Lock kommen kann. Der Monitor wurde auf Ebene der Ressourcen realisiert (hier: Client Quota und Task Package). So wird ein Request-Thread eines Services niemals zwei Ressourcen zur selben Zeit belegen (nur sequentiell). Ebenfalls ist zu erkennen, dass es keine hierarchische Verschachtelung der Locks gibt.

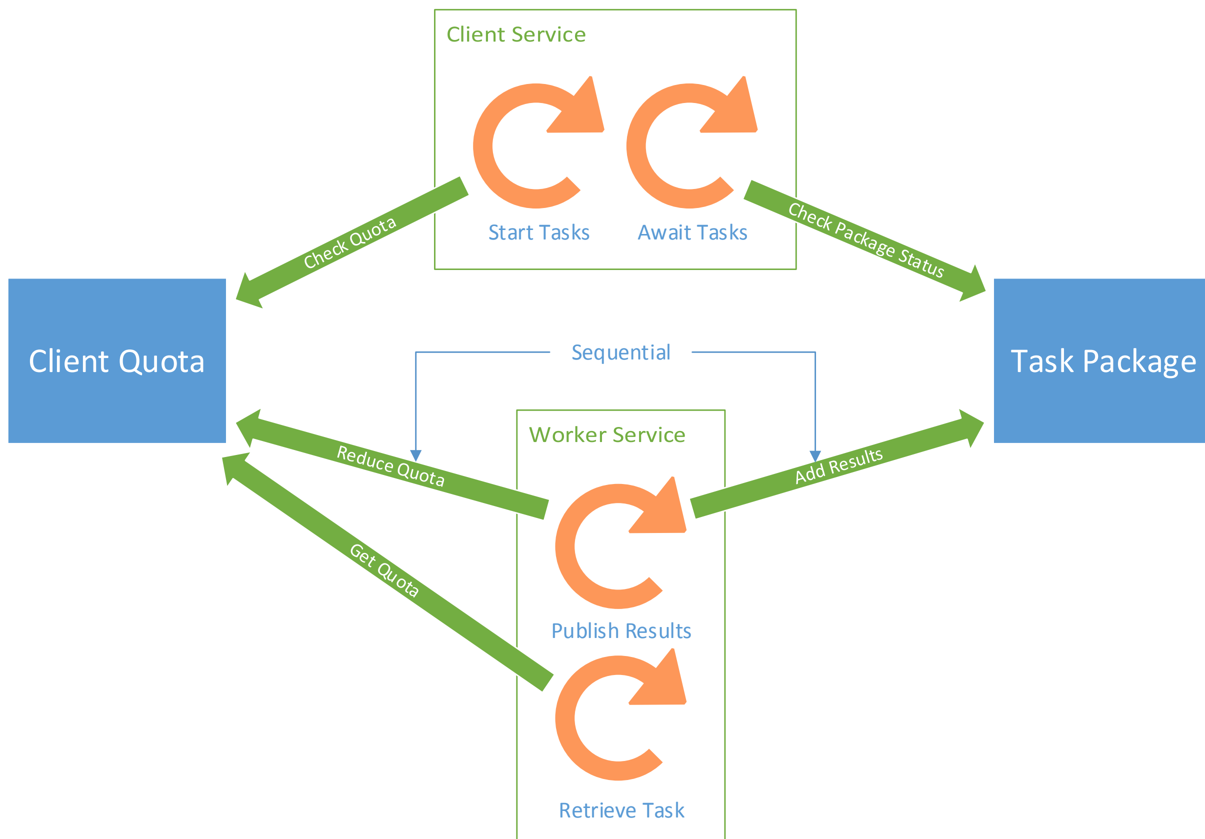

Abbildung 9 Thread-Synchronisation mit Monitor

2.5 Worker

Ein Worker stellt dem System Worker Threads zur Verfügung und ist intern als Thread Pool aufgebaut. Innerhalb des Worker Prozesses laufen ein oder mehrere Worker Threads, welche an der verteilten Task Ausführung beteiligt sind.

Ziel ist es, den Host des Worker Prozesses möglichst hoch auszulasten. Da Task-Code keine blockierenden IO Aufrufe enthält ist es ausreichend, so viele Worker Threads wie Anzahl logischer Cores zur Verfügung zu stellen.

Sind alle Worker Threads mit der Ausführung von Task-Code beschäftigt, so sind die verfügbaren CPU Ressourcen optimal ausgelastet.

2.5.1 Worker Thread

Ein Worker Thread führt kontinuierlich einen Execution Loop aus, in welchem die Tasks verarbeitet werden. Der Execution Loop kann nur durch einen Abbruch des Worker Threads gestoppt werden.

Code 3 ist eine vereinfachte Illustration des Ablaufs eines Worker-Threads. Der Worker-Thread kann zwei verschiedene Responses vom Master erhalten. Ist ein Task verfügbar, erhält er einen ausführbaren Task, ansonsten einen Delay. Erhält der Worker-Thread einen Delay, so pausiert er seine Ausführung solange, wie es der Wert des Delays vorgibt. Danach versucht der Worker-Thread wieder, einen Task vom Master zu holen. Sobald der Worker-Thread einen ausführbaren Task erhält, wird dieser direkt vom Worker-Thread ausgeführt. Sollte bei der Ausführung eines Tasks eine Exception auftreten, wird das Ergebnis einfach mit *Error* markiert. Konnte der Task erfolgreich ausgeführt werden, wird das Resultat mit *Completed* markiert und die resultierenden Daten mitgeliefert.

```
Execution Loop {
	task = master.RetrieveTask();
	if(task is Delay) {
		Sleep(delay);
	} else {
		result = Execute(task);
		master.PublishResult(result);
	}
}
```

Code 3 Worker Thread Execution Loop

2.5.2 Kommunikation

Für die Kommunikation mit dem Service wird für jeden Request eine neue Instanz des WCF-Service Clients *ClientBase* erstellt. Beobachtungen und mehrere Tests haben zwar gezeigt, dass mehrere Requests intern synchron ausgeführt werden, allerdings wird dieses Verhalten in der MSDN Dokumentation [8] nicht angegeben. Da die Konstruktion allerdings zeitintensiv ist [9]:

- ContractDescription Tree erstellen
- alle CLR Typen reflektieren
- Channel Stack erstellen
- Ressourcen freigeben

wird die *CacheSetting* der Factory auf *AlwaysOn* gesetzt.

2.5.3 Graceful Shutdown

Beim Herunterfahren eines Workers werden alle laufenden Tasks noch zu Ende verarbeitet und die Resultate übermittelt. Somit gehen beim Beenden des Workers keine Tasks beziehungsweise Resultate verloren und der Worker kann ohne negativen Nebeneffekte heruntergefahren werden.

2.5.4 Dynamic Code Memory Leak

Es wurde ein Memory Leak bei der Ausführung von vielen Tasks beobachtet. Mit folgendem Code wurde geprüft, ob es sich um ein Problem mit vielen Daten oder um eines mit viel Code handelt.

```
byte[] input = new byte[dataSize];
distribution.ParallelFor(0, taskCount, row => {
  transferred = input.Length;
});
```

Code 4 Task zur Memory-Leak Reproduktion

Der benötigte Speicher des Workers beträgt nach dem vollständigen starten 3 MB.

Durch das Senden eines Tasks mit 50 MB Daten konnte im TaskManager beobachtet werden, dass durch den Aufruf des Garbage Collectors der Memory Footprint von rund 300 MB wieder auf 3 MB gefallen ist.

Hingegen veränderte sich der Memory Footprint bei 5‘000 Tasks mit je 0 Byte Daten nur gering. Maximal waren es 142 MB und nach dem Aufruf des Garbage Collectors immerhin noch 132 MB.

Dieser Memory Leak konnte durch das Setzen von *AssemblyBuilderAccess.RunAndCollect* anstelle von *AssemblyBuilderAccess.Run* bei der Erstellung der Dynamic Assembly behoben werden. Collectible Assemblies wurden mit .NET 4.0 eingeführt [10].

2.6 Fehlerfälle

Zu beachten gibt es noch einige zu erwartende Fehlerfälle, welche in der aktuellen Lösung aber nicht behandelt werden. Die Behandlung dieser Fehlerfälle hätte den Zeitrahmen des Projektes gesprengt.

TASK PACKAGE MEMORY LEAK

Der Client trägt Tasks ein, fragt aber deren Resultate nicht ab.

Führt zu einem Memory Leak, da Tasks erst nach der erfolgreichen Rückgabe des *AwaitTasks* Services gelöscht werden.

Eine mögliche Lösung wäre Packages, welche mit Status *Completed* oder *Error* auf dem Server liegen, nach einer gewissen Zeit zu entfernen.

INKOMPLETTE TASKS

Ein Worker stürzt während der Ausführung ab oder wird nicht gracefully beendet.

Einmal abgeholte Tasks werden aus der Task-Queue gelöscht. Durch den hier fehlenden *PublishResult* Service-Aufruf wird das Task-Package nie vollständig abgeschlossen.

Durch das nicht direkte Löschen der Tasks, sondern einer speziellen Markierung (oder Ablage auf einer Backup-Queue), könnte ein Task erneut ausgeführt werden.

MASTER CRASH

Der Master stürzt ab.

Sämtliche eingetragenen Tasks werden nur In-Memory abgelegt und gehen bei einem solchen Absturz verloren.

Die sicherste Lösung bei einem solchen Fall wäre eine Persistierung. Eine andere Möglichkeit wäre, dass der Client das Package erneut auf dem Master einträgt.

ENDLOSE TASKS

Der Client erstellt Tasks mit Endlos Loops.

Das System verlangt zwar formell, dass keine Endlosschlaufen in den Tasks sein dürfen, allerdings kann dies nicht geprüft werden (NP-Vollständig, Halteproblem). Sendet ein Client ein Packet voller Endlosschlaufen, so legt er das System lahm.

Durch das Einführen eines absoluten Maximums an Ausführungszeit pro Task kann das Problem zwar nicht gänzlich gelöst, aber entschärft werden.

3. Benchmarks

Die Benchmarks zeigen auf, wie sich die neue Lösung gegen die bestehende HPC-Lösung bewährt. Dabei wird auf zwei Punkte besonderen Wert gelegt. Die Skalierung mit steigender Anzahl von zur Verfügung stehenden Cores sowie das Performanceverhalten bei steigender Anzahl der Tasks. Ausserdem werden die anfallenden Overheads analysiert und bewertet.

Aufgrund der unterschiedlichen Architekturen der beiden Lösungen sowie begrenzten Hardwareressourcen für die Benchmarks ist es schwierig, exakte Vergleiche zur HPC-Lösung ziehen zu können. Die neue Lösung kann nicht auf denselben Maschinen getestet werden, wie es mit dem Einsatz des HSR HPC Clusters der Fall ist. Dabei ist die neue Lösung allerdings eher im Nachteil, da die Rechner im Cluster mehr Kerne besitzen und ausserdem über InfiniBand im Cluster vernetzt sind. Zusätzlich besteht bei HPC kaum Kontrolle darüber, wie die Verteilung und Partitionierung der Tasks tatsächlich geschieht.

Das Ziel der neuen Lösung ist, mit einfacheren Mitteln und weniger Ressourcen ähnlich gute Resultate zu erzielen wie mit der bisherigen. Somit zeigen die Benchmarks unter diesen Bedingungen auch, ob dieses Ziel erreicht werden konnte.

Gelöst werden bei allen Benchmarks dieselben Aufgaben. Jede Aufgabe ist dabei eine unabhängige Faktorisierung von zufällig generierten Produkten von Primzahlen. Ein Task entspricht dabei einer solchen Aufgabe.

```
Factorize {
	for (k = 2; k * k <= number; k++) {
		if(number % k == 0) { return k; }
	}
	return number;
}
```

Code 5 Verteilte Aufgabe

Die Aufgabe wurde gewählt, weil sie sehr arbeitsintensiv ist und das System dadurch maximal ausgelastet wird.

3.1 Versuchsaufbau

Für die Durchführung der Benchmarks werden zwei Umgebungen benötigt. Eine, um das neue System zu testen und eine für die Durchführung der Tests mit einem HPC Cluster.

3.1.1 Neues Backend System

MESSPUNKTE

Für die Erfassung der aufgewendeten Rechenzeiten wurden an den notwendigen Stellen Messpunkte eingebaut, um diese festzuhalten.

Nummer	Messung	Ort	Granularität
1	Serializing Time	Client	Pro Task Package
2	Task Retrieve Time	Worker	Pro Task
3	Task Execution Time	Worker	Pro Task
4	Publish Time	Worker	Pro Task
5	Deserializing Time	Client	Pro Task Package
6	Total await Time	Client	Pro Task Package

Tabelle 2 Messpunkte Distributed Task Pool

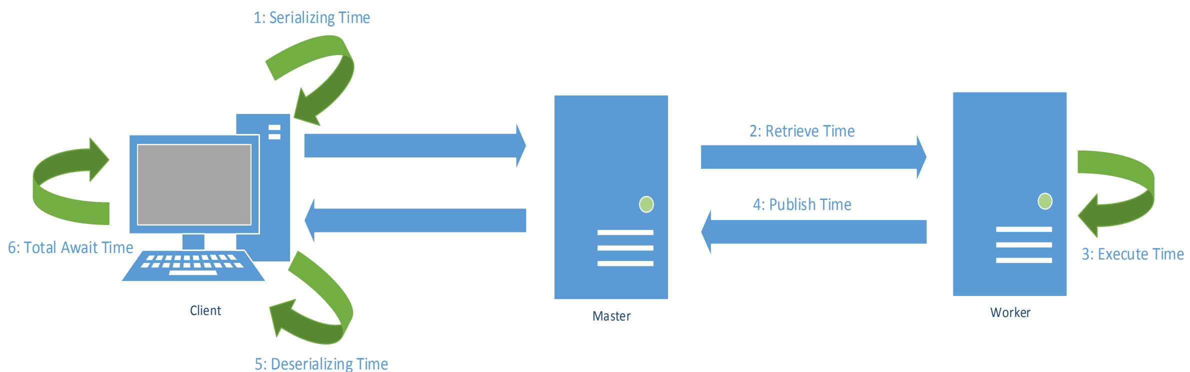

Abbildung 10 Übersicht Messpunkte

Die Resultate der Benchmarks werden aggregiert und ergeben ein zuverlässiges Abbild der benötigten Zeiten pro Task beziehungsweise pro Task Package.

Intern werden die Zeiten mithilfe von Stopwatches [11] erfasst. Da diese Art der Zeiterfassung nicht hochpräzise ist, sind die Werte nicht als absolut zu verstehen, sondern als gute Approximation. Ausserdem werden durch die variierenden Pollingzeiten und der nichtdeterministischen Auslastung der Worker – einer könnte sich gleich alle Schnappen, aber bei einem weiteren Versuch sind mehrere beschäftigt – bei kleineren Taskmengen unterschiedliche Ausführungszeiten erreicht.

Der Netzwerkoverhead ist ebenfalls Schwankungen unterlegen, da die Benchmarks in einem anderweitig bereits gut ausgelasteten Netzwerk durchgeführt wurden. Die Abweichungen bewegen sich allerdings im Millisekunden Bereich und haben auf die Aussagekraft der Benchmarks keinen Einfluss.

Die Erfassung der Benchmarks selbst ist ebenfalls mit einem Overhead verbunden, allerdings wurde versucht, dieser so gering wie möglich zu halten. Während des gesamten Benchmarks wird kein File IO generiert. Erst nach Beendigung der Durchläufe werden die Ergebnisse aus dem Memory in ein entsprechendes File geschrieben.

Es wurden jeweils 3 Läufe durchgeführt und jeweils das Beste Ergebnis übernommen.

UMGEBUNG

Für die Durchführung der Benchmarks werden von der HSR zur Verfügung gestellten Rechner verwendet. Als Master dient ein virtueller Server, welcher für die Semesterarbeit zur Verfügung gestellt wurde. Für die Worker werden Rechner aus einem Übungszimmer verwendet. Beim Client handelt es sich um ein privates Notebook. Der genaue Aufbau ist Abbildung 11 zu entnehmen.

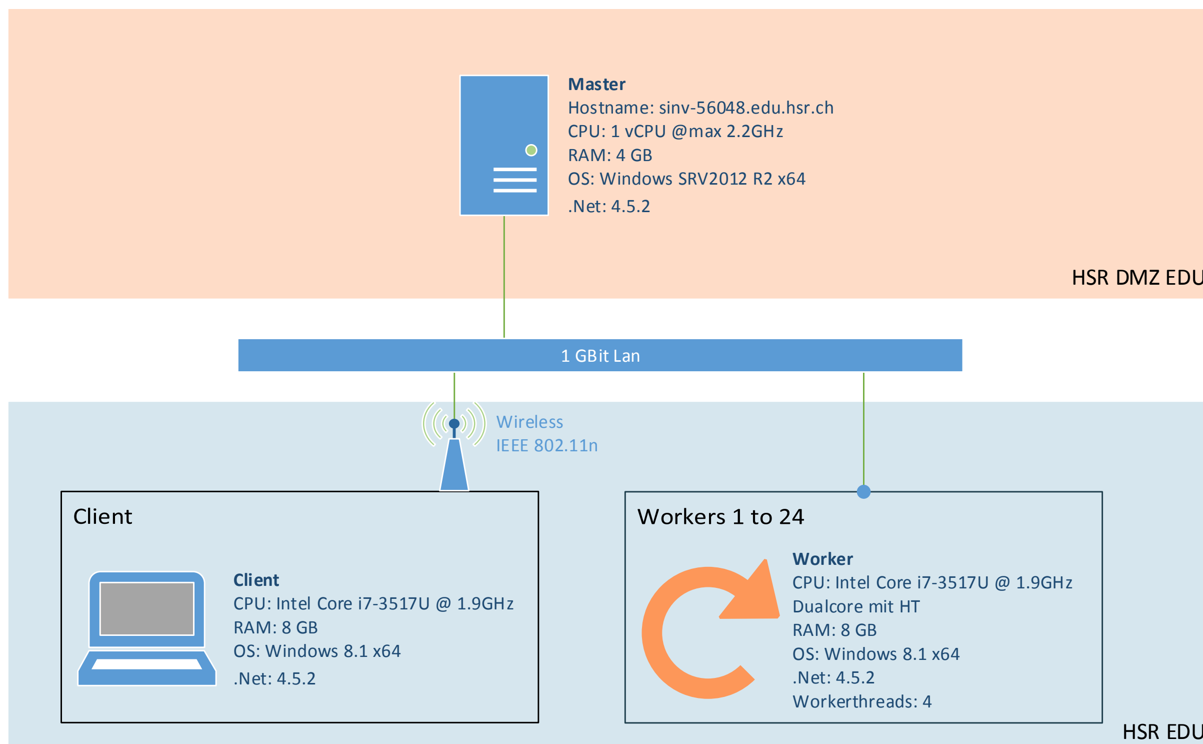

Abbildung 11 Übersicht Distributed Thread-Pool Testaufbau

Zu beachten ist, dass die Kommunikation über zwei getrennte Subnetze geführt wird, was den Netzwerkoverhead beeinflussen kann.

Für den Versuch wird der Log-Level für den Master auf *Error* und für die Worker auf *Warn* heruntergestuft. Dadurch wird vermieden, dass häufige Konsolenausgaben – speziell im Debug Level – den Versuch nicht negativ beeinflussen. Ebenfalls werden nur Release-Builds (Assemblys mit Compiler Optimierungen) verwendet.

3.1.2 Microsoft HPC

Mangels genauer Kontrolle über die Verteilung und Partitionierung der Tasks auf dem Microsoft HPC Cluster sind die gewonnenen Zahlen nicht direkt vergleichbar. Da die dadurch entstehenden Differenzen allerdings gering sind, bleibt die Aussagekraft der Vergleiche erhalten.

MESSPUNKTE

Aufgrund der komplexeren Architektur der bisherigen Lösung ist es schwieriger, Messpunkte einzufügen, weshalb es auch weniger sind als beim neuen System.

Nummer	Messung	Ort	Granularität
1	Deserialisierung des Taskpackage	Node	Pro Partition des Taskpackage
2	Internalisierung des Taskpackage	Node	Pro Task
3	Externalisierung der Resultate	Node	Pro Task
4	Serialisierung der Resultate	Node	Pro Partition des Taskpackage

Tabelle 3 Messpunkte HPC

UMGEBUNG

Für die Durchführung der Benchmarks mit einem HPC Cluster, wird auf den von der HSR zur Verfügung gestellten HPC zurückgegriffen [12].

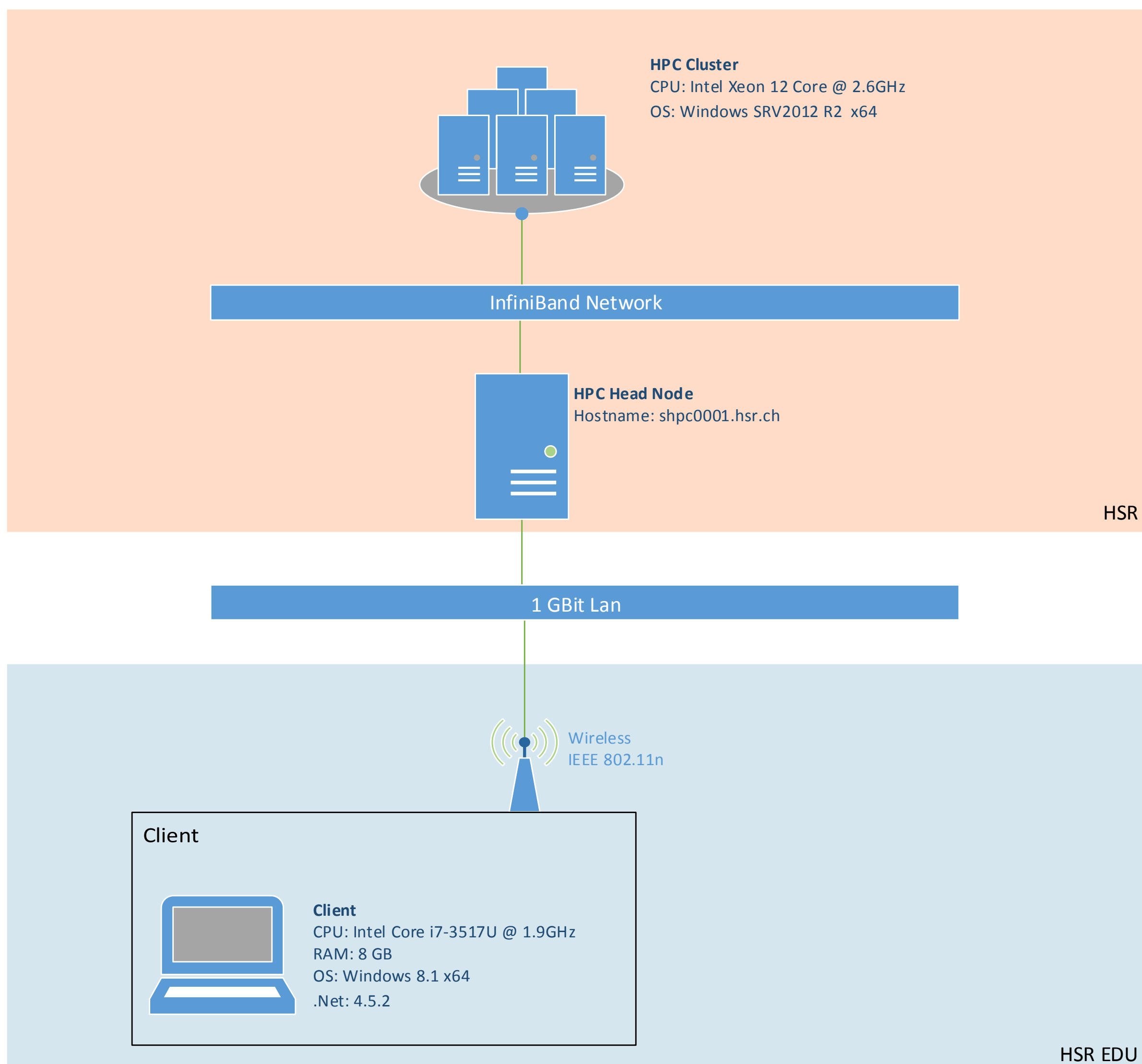

Abbildung 12 Übersicht HPC Cluster Testaufbau

3.2 Benchmark Ergebnisse

In den folgenden Benchmarks werden jeweils Vergleiche zwischen folgenden Setups durchgeführt:

- Lokale, sequentielle Ausführung

- Lokale, parallele Ausführung auf 4 Cores mittels TPL
- Ausführung mit HPC auf 1 Node mit 12 Cores bis zu einem Total von 8 Nodes mit 96 Cores.
- Ausführung mit dem verteilten Thread Pool auf 1 Node mit 4 Cores bis zu einem Total von 24 Nodes mit 96 Cores.

3.3 Direktvergleich

Der Direktvergleich wurde mit 100 Faktorisierungen durchgeführt. Der Vergleich zeigt, wie die Laufzeit der verschiedenen Ausführungsvarianten ist. Abbildung 13 zeigt, dass die neue Lösung um etwa Faktor 2 schneller als die bestehende ist. Als Referenz sind zusätzlich die lokal sequentielle Ausführung und die parallele Ausführung dargestellt, welche beide essenziell höhere Laufzeiten ausweisen.

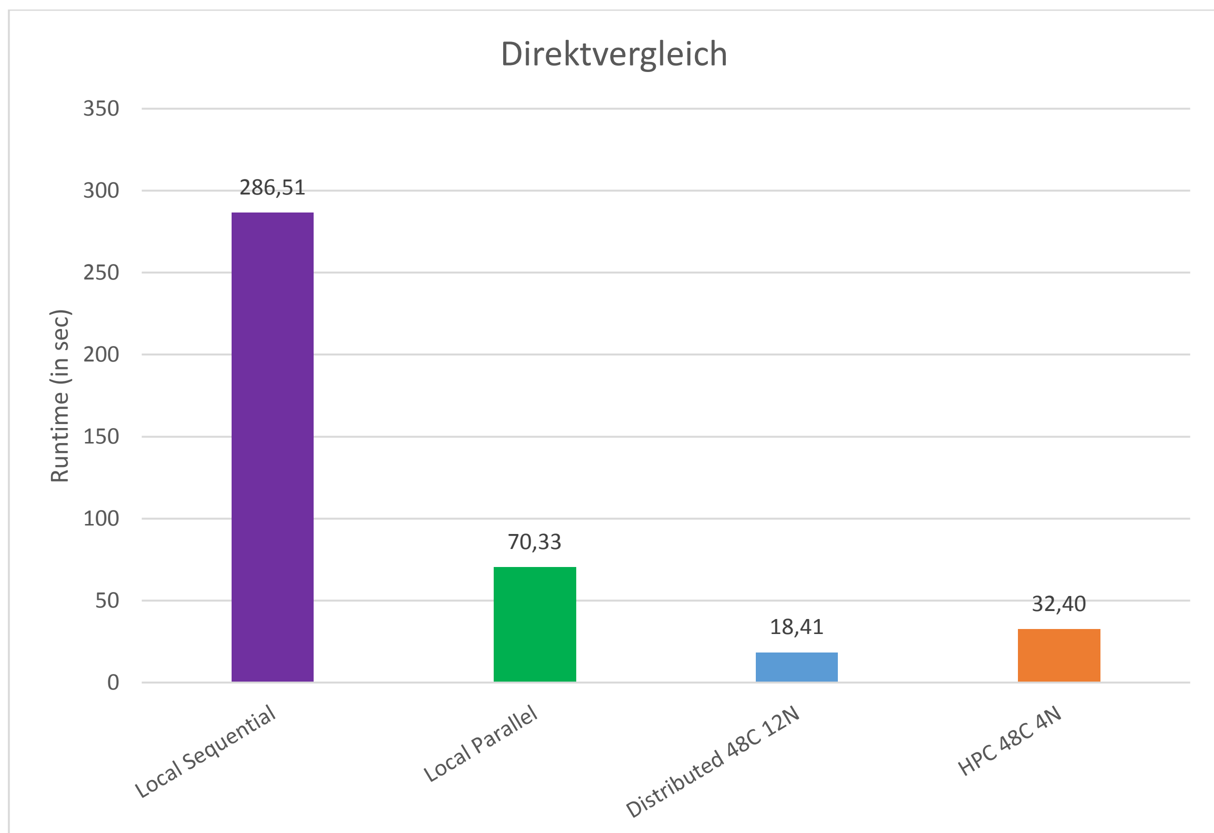

Abbildung 13 Direktvergleich

3.3.1 Skalierung

Die Skalierung stellt dar, wie sich die Laufzeit einer verteilten Aufgabe zur Anzahl zur Verfügung stehender Cores verhält. Je grösser das System wird, desto schneller sollte die Aufgabe ausgeführt werden. Mit jeder hinzugefügten Maschine erhöhte sich die Gesamtzahl an Cores um jeweils vier.

SKALIERUNG BEI STEIGENDER CORE-MENGE

Die Skalierung wurde mit einer Gesamtzahl von 500 Faktorisierungen gemessen. Abbildung 14 zeigt, dass die Ausführungsgeschwindigkeit mit zunehmender Core-Menge stetig steigt. Auch zu erkennen ist, dass die gewonnene Ausführungszeit kontinuierlich abnimmt. Das kommt daher, dass das Verhältnis an gewonnener Rechenleistung mit jeder zusätzlichen Maschine weiter abnimmt.

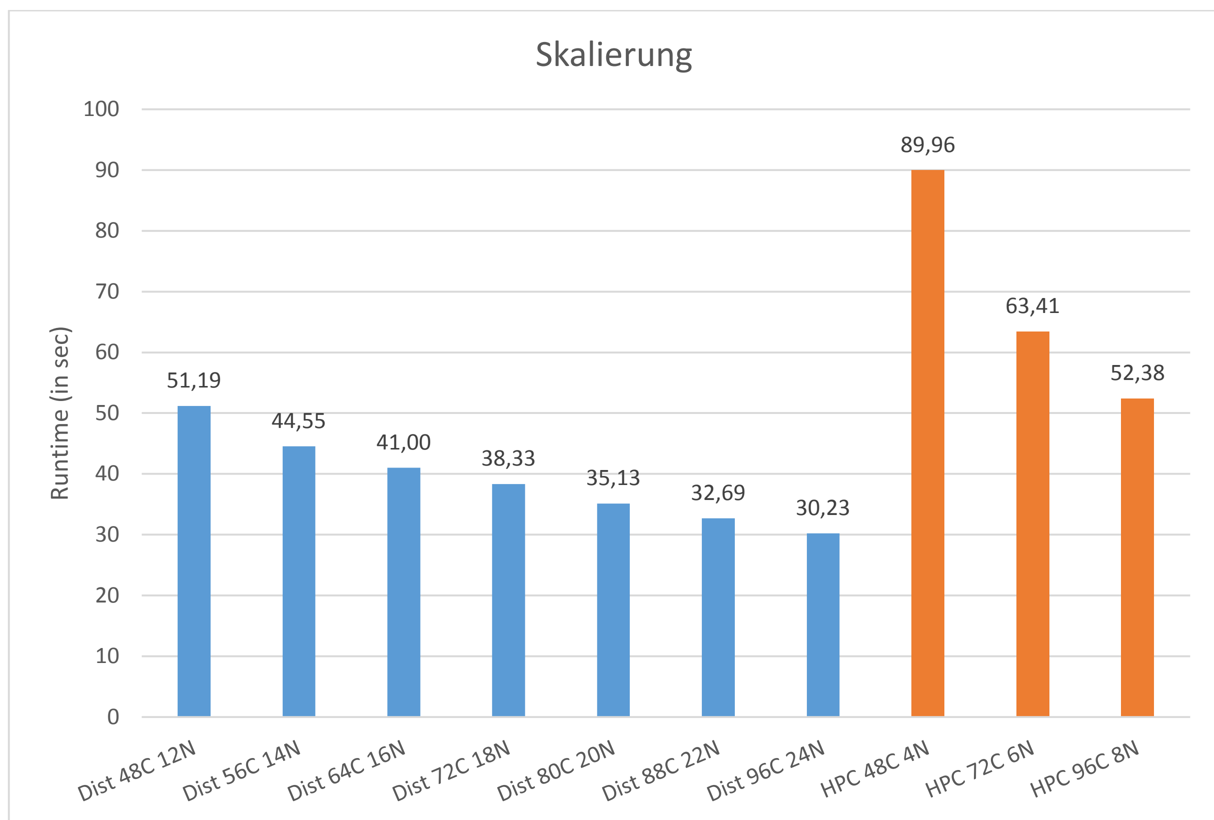

Abbildung 14 Skalierung mit 100 Faktorisierungen

Bemerkenswert ist der Gewinn gegenüber dem HPC Cluster. Dieser ist den Fällen mit jeweils 96 Cores rund 60% schneller beim verteilten Thread Pool. Dieser markante Gewinn rührt daher, dass die Worker im Thread Pool ständig neue Tasks holen, bis die zentrale Task-Queue leer ist. Hingegen werden beim HPC-Backend die Tasks in einzelne Partitionen aufgeteilt und auf die verschiedenen Nodes verteilt. Durch diesen Push-Mechanismus werden Nodes, welche die ihnen zugewiesenen Tasks ausgeführt haben, nicht mehr mit neuen Aufgaben versorgt und sind somit Idle, auch wenn die übrigen Nodes noch beschäftigt sind.

SKALIERUNGSLIMIT

Das absolute Minimum für die Ausführungszeit wird dann erreicht, wenn die Core-Menge die Anzahl Tasks übersteigt. Dadurch ist es nicht möglich, ein gewisses Minimum an Ausführungszeit zu unterschreiten.

Das Minimum im verteilten Thread-Pool wird unvermeidlich grösser sein als das Minimum bei einem lokalen Thread-Pool. Das kommt daher, dass gewisse Faktoren unabhängig der Anzahl Cores sind.

- Poll-Zeiten
- Serialisierung und Deserialisierung der Daten (bei Client und Worker)
- Netzwerk Übertragung

3.3.2 Performance

Mit der Analyse der Performance wird untersucht, wie sich die totale Laufzeit einer verteilten Aufgabe zu ihrer Grösse verhält. Die Benchmarks wurden für die neue Lösung auf 12 Worker Maschinen mit jeweils acht logischen Cores und acht Worker-Threads durchgeführt. Somit stehen 12 Nodes mit insgesamt 96 Worker-Threads zur Verfügung. Das HPC Backend wurde mit 8 Nodes mit jeweils 12 Cores getestet.

Ideal ist, wenn die Laufzeit konstant bleibt, bis die Grösse der Tasks die Anzahl zur Verfügung stehender Worker-Threads, respektive Cores, überschreitet.

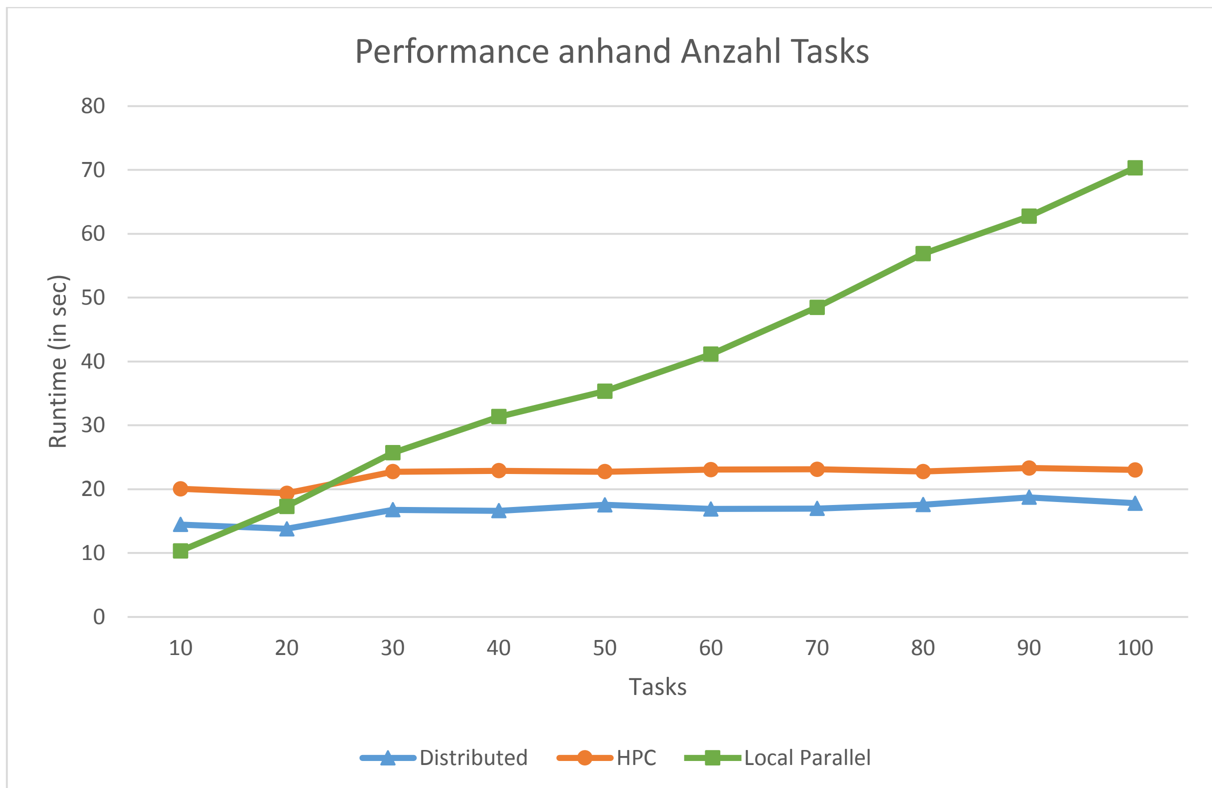

Abbildung 15 Skalierung anhand Task Menge

Abbildung 15 zeigt die gemessenen Ausführungszeiten anhand der Anzahl Tasks. Während die Achse bei der parallelen Ausführung stetig steigt, bleiben die Werte für die verteilten Ausführungen konstant. Die neue Lösung erzielt dabei durchgehend bessere Ergebnisse als das bisherige Backend.

3.3.3 Overhead

Der Overhead berücksichtigt alle zeitlichen Einflussfaktoren, welche in einem lokalen Thread-Pool nicht vorhanden sind. Durch die verteilte Ausführung entsteht ein entsprechender Overhead bei der Übermittlung und Serialisierung der Tasks. Der Serialisierungs-Overhead ist abhängig von der Grösse und der Art des Portable Task Codes sowie des Portable Task Data. Der Client hat den Aufwand der Serialisierung doppelt. Zu Beginn externalisiert er alle Tasks und nach Erhalt des Resultates werden alle Daten wieder internalisiert. Danach unterscheidet sich der nötige Overhead zwischen dem neuen und dem alten Backend.

NEUES BACKEND

Beim neuen Backend entsteht vor allem bei der Kommunikation zwischen Master und Workern Overhead. Jeder Task wird separat von einem Worker-Thread abgeholt, wobei schon einmal Netzwerkoverhead entsteht. Danach wird der Task von dem Worker-Thread internalisiert und zum Schluss externalisiert, wobei jeweils Serialisierungszeit aufgewendet werden muss. Zum Schluss entsteht noch einmal Overhead, wenn das Resultat des Tasks dem Master übermittelt wird.

HPC BACKEND

Beim alten Backend entsteht vor allem Overhead bei der Serialisierung der Daten. Da die Tasks partitioniert und im Bulk an die Nodes übermittelt werden, ist der Netzwerkoverhead allerdings um einiges geringer. Die Tasks werden auf dem Node aus einem File deserialisiert und dann sequentiell internalisiert. Zum Schluss werden die Tasks sequentiell externalisiert und wiederum in ein File serialisiert. Bei den sequentiellen Abschnitten ist der Node dabei nicht optimal ausgelastet.

SKALIERUNG

Bei der Skalierung des Overheads wird untersucht, wie sich der totale Overhead zur Grösse der Aufgabe verhält. Dabei soll der Overhead möglichst gering und stabil gehalten werden.

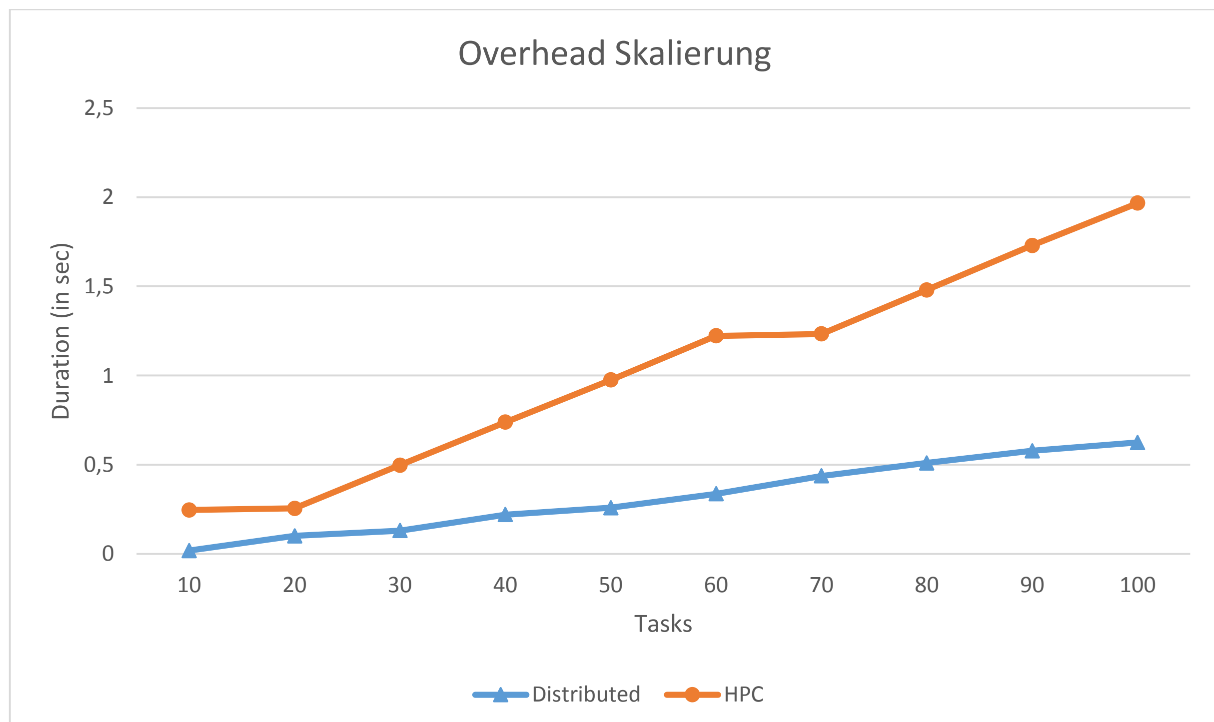

Abbildung 16 Overhead Skalierung

Zu beobachten ist, dass der Overhead deutlich kleiner ist als beim Einsatz des HPC Cluster. Bei einer Menge von 100 Tasks muss das bisherige Backend knapp zwei Sekunden für Overhead aufwenden. Die neue Lösung hingegen lediglich ein wenig mehr als eine halbe Sekunde.

Dies ist – wie zu erwarten war – auf den grossen und nicht optimalen Serialisierungsaufwand des bestehenden Backends zurückzuführen.

Bei der neuen Lösung hingegen wird jeder Task von einem Workerthread serialisiert. Dies bedeutet, dass bereits die Internalisierung und Externalisierung parallel geschieht und somit besser skaliert. Das System ist jederzeit optimal ausgelastet und die Tasks können komplett unabhängig ausgeführt werden.

Wichtigste Erkenntnis ist die Stabilität der Skalierung. Der Overhead erhöht sich in kleinen, linearen Schritten, was es einfacher macht, Overheads abzuschätzen und zu evaluieren, ob sich eine verteilte Ausführung lohnt.

VERTEILUNG

Abbildung 17 zeigt den totalen Overhead bei der Ausführung von 100 Tasks. Ersichtlich ist – wie bei der Skalierung bereits erwähnt – dass das bisherige Backend viel Zeit für die Serialisierung aufwendet. Beim neuen Backend ist der Netzwerk Overhead um etwa Faktor 5 grösser. Jedoch kann dies durch die sehr viel schnelleren Serialisierungszeiten wieder kompensiert werden. Ausserdem verfügte, wie im Testsetup erwähnt, das HPC Backend über eine sehr viel schnellere Netzwerkanbindung.

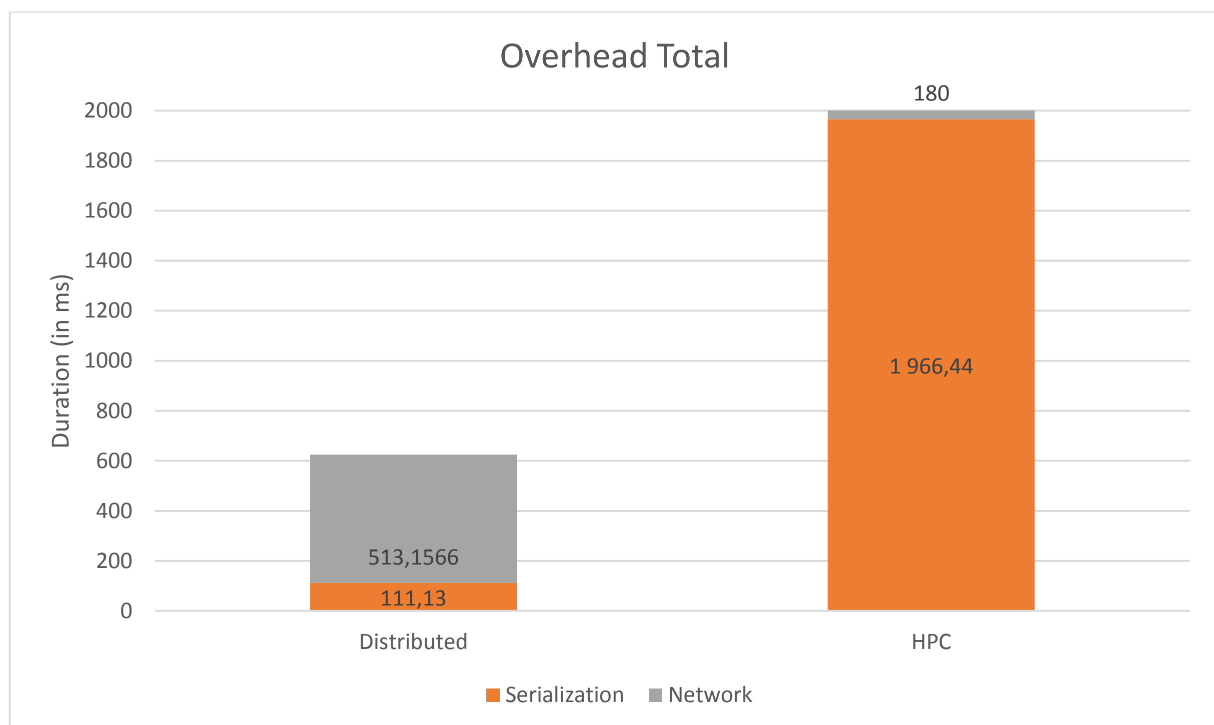

Abbildung 17 Overhead Total

Innerhalb des neuen Backend Systems lassen sich die Overheads noch detaillierter darstellen. Abbildung 18 zeigt, wie sich der Overhead relativ verteilt. Dabei ist ersichtlich, dass das Holen von Tasks und die Übermittlung der Resultate etwa gleich lange dauert. Die Publish- und Retrievezeiten beinhalten auch die benötigte Zeit des Masters. Darin ist die Zeit enthalten, welche der Master aufwenden muss, um die Tasks aus der Wartschlange herauszunehmen, respektive um die Resultate einzuordnen. Somit macht die Übertragung der Daten über das Netzwerk 82% des gesamten Overheads aus.

Bei der Serialisierung fällt die Internalisierung um Faktor 8 stärker ins Gewicht als die anschliessende Externalisierung. Dies liegt daran, dass das Internalisieren an sich bereits aufwändiger ist, da alle Daten initialisiert werden müssen und vor allem der Code noch geladen wird. Bei der Externalisierung muss lediglich das Resultat serialisiert werden. Die Serialisierung macht 18% des gemessenen Overheads aus.

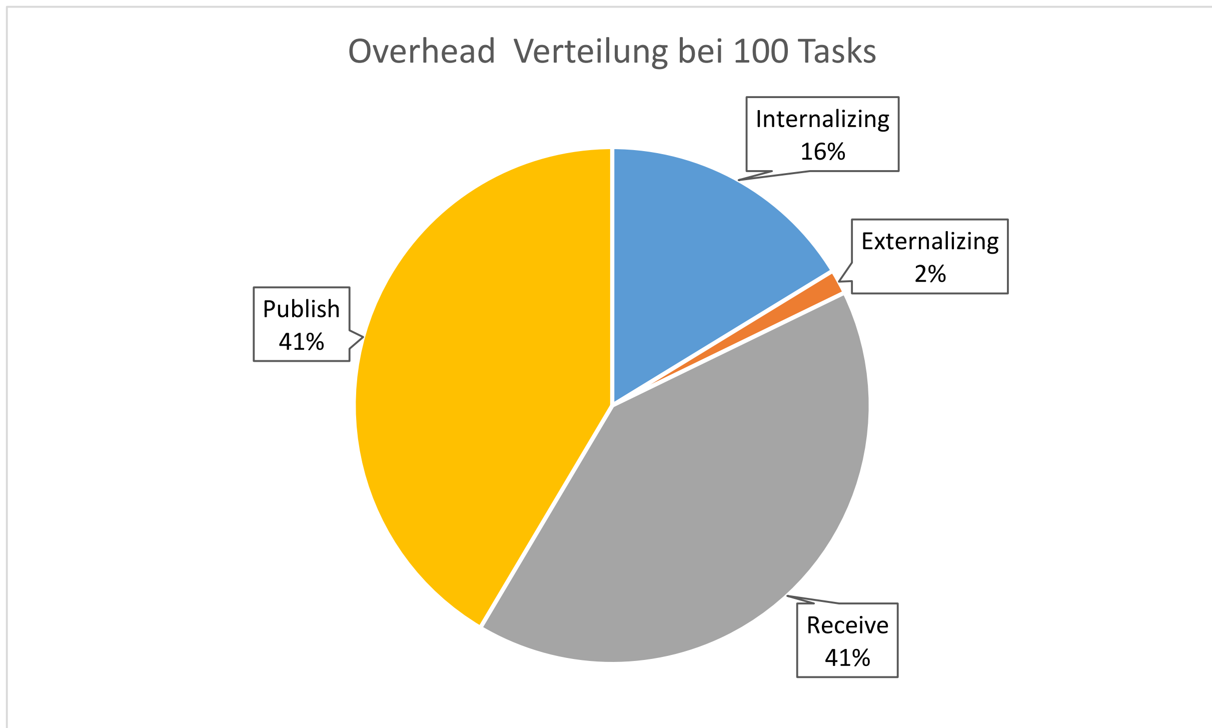

Abbildung 18 Overhead Verteilung

3.3.4 Optimierung der Rechenzeit

Während der Benchmarks ergab sich, dass die reine Ausführungszeit eines Tasks im verteilten Thread Pool gut 80% höher ist, als dessen lokale Ausführung. Es wurden mehrere Versuche mit unterschiedlichen Implementationen des Algorithmus für die Faktorisierung durchgeführt.

Das Ersetzen der Quadratwurzel durch ein Quadrat hat schon eine Leistungssteigerung von 25% gegenüber der Original-Lösung erzielt; verlangsamte allerdings die lokal sequentielle Ausführung geringfügig. So ergab sich der Verdacht, dass der Aufruf der nativen Funktion „Math.Sqrt()“ der Bottleneck ist.

Um den Verdacht der langsamen „Math.Sqrt()“ Funktion auf den Grund zu gehen, wurde diese aus dem Loop extrahiert und das Resultat der Berechnung in eine temporäre Variable gespeichert. Allerdings veränderte sich die Geschwindigkeit nur minim.

Der letzte Versuch, welcher zu markanten Veränderungen der Resultate geführt hat, ist das manuelle Casten von double auf long (bzw. die IL-Funktion conv.i8 [13]) ausserhalb der Schleife. Dadurch wurde einerseits die sequentielle Ausführung – wenn auch nur geringfügig – beschleunigt. Andererseits hat dies aber einen markanten Einfluss auf die Ausführungszeit im verteilten Thread Pool ergeben. So zeigt Abbildung 19, dass durch diese Anpassung die Ausführungszeit um ganze 30% reduziert wurde. So ist diese Variante nur noch 40% langsamer als die lokal sequentielle Ausführung, im Gegensatz zu den ursprünglichen 80%.

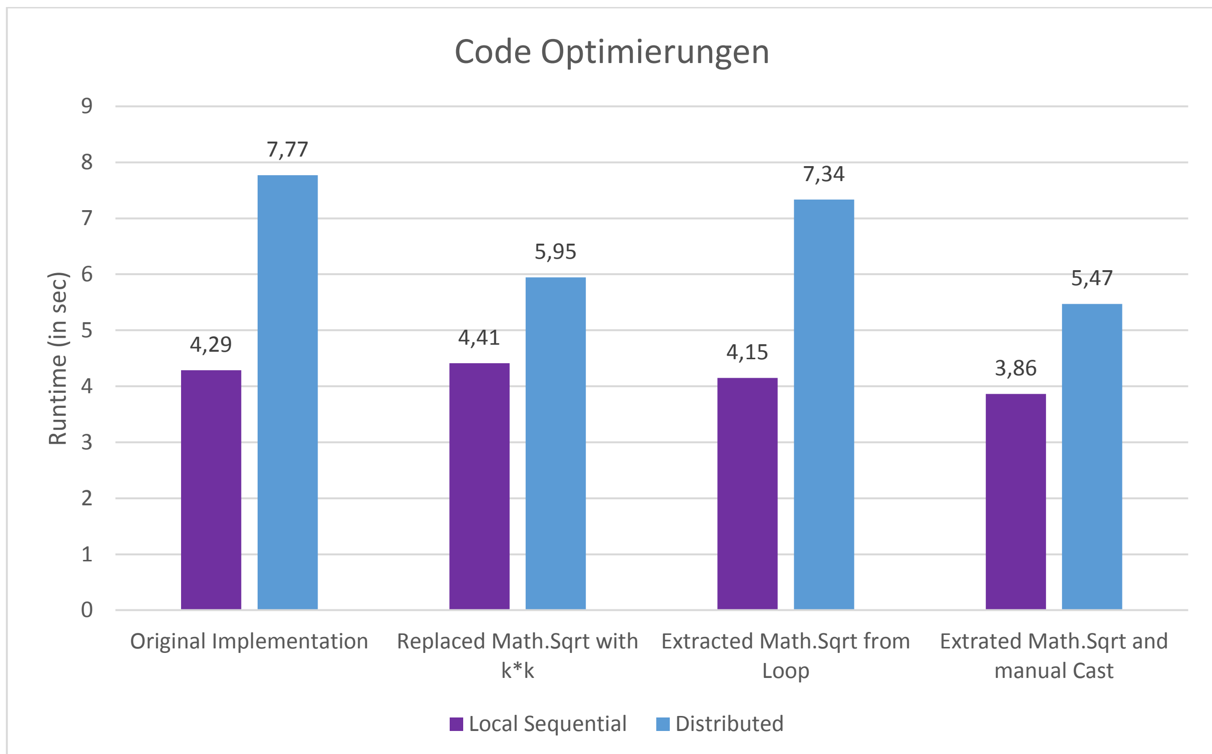

Abbildung 19 Code Optimierungen

Auch wenn die Ausführungszeit verbessert werden konnte, so konnte die Zielgeschwindigkeit der lokalen Ausführung nicht erreicht werden. Weitere Versuche haben gezeigt, dass sowohl das Auslagern als auch das Inlinen von Methoden keinen nennenswerten Einfluss auf die Ausführungszeit hatten.

Verbleibende Vermutungen sind, dass gewisse JIT-Optimierungen bei Dynamic Assemblys nicht durchgeführt werden oder Sicherheitsfunktionen strikter vorgehen. Im Rahmen dieses Projekts wurde diesem Problem nicht weiter auf den Grund gegangen.

Allerdings zeigt diese Beobachtung, dass manuelle Optimierungen durchaus einen starken Einfluss auf das Gesamtergebnis haben. Code 6 zeigt die finale Implementation der Faktorisierungs-Funktion mit den besten Ergebnissen.

```
Factorize {
	long sqrt = (long)Math.Sqrt(number);
	for (k = 2; k * k <= sqrt; k++) {
		if(number % k == 0) { return k; }
	}
	return number;
}
```

Code 6 Optimierte Faktorisierungs-Funktion

Die vorher beschriebenen Benchmarks konnten – aufgrund nicht verfügbarer Testumgebung – nicht erneut mit der angepassten Funktion ausgeführt werden.

3.3.5 Fazit

Die durchgeführten Benchmarks zeigen, dass die neue Lösung deutlich effizienter und performanter als die bisherige Lösung ist.

Das System skaliert stabiler mit steigender Anzahl Cores und die Performance bleibt bei steigender Anzahl von Tasks stabiler und performanter als bei der bisherigen Lösung. Die

Ausführungsgeschwindigkeit des neuen Backends ist – bei grossen Taskmengen – über 50% schneller als beim HPC Cluster Backend.

Der Overhead für die Serialisierung der Daten konnte – durch dessen Parallelisierung – um etwa Faktor 20 minimiert werden. Der Netzwerkoverhead ist bei der neuen Lösung jedoch rund 3 Mal höher. Schlussendlich kann aber die Einsparung bei der Serialisierung diesen erhöhten Netzwerkoverhead kompensieren.

Die Ergebnisse schliessen darauf, dass das neue Backend das Bestehende mit vielen einhergehenden Vorteilen ablösen kann.

4. Mögliche Verbesserungen

Ziel war es, ein stabiles Grundgerüst für die verteilte Task-Ausführung zu implementieren. Dabei sollte das neue Backend so einfach wie möglich bleiben und die Komplexität gering gehalten werden. Aus diesen sowie aus Zeitgründen wurden diverse mögliche Verbesserungen nicht implementiert. Die noch nicht implementierten Verbesserungen werden in diesem Kapitel genauer dokumentiert und gegebenenfalls bereits mit bekannten Vor- und Nachteilen beschrieben.

4.1 Code Verarbeitung

Das Deserialisieren des auszuführenden Programmcodes nimmt einige Zeit in Anspruch. Um diesen Overhead zu minimieren, könnte der Programmcode recycled werden.

Das heisst, der Programmcode wird in diesem Fall nach wie vor mit jedem Task übermittelt. Allerdings prüft der Worker nun, ob bereits Code mit der zugewiesen *Package Id* vorhanden ist. Wenn Code vorhanden ist, wird dieser wieder ausgeführt, ansonsten wird der Code neu geladen.

Der Vorteil dieser Verbesserung ist sicherlich ein Performancegewinn bei der Deserialisierung des Programmcodes. Damit sinkt der anfallende Overhead, wenn Code aus dem Cache geladen werden konnte.

Der Nachteil ist, dass die Einführung eines Caches schnell viel Komplexität einführt. Ausserdem hat das Unterhalten eines Caches nur einen Vorteil, wenn Tasks exakt denselben Code ausführen müssen.

4.2 Eigener Netzwerk-Stack für Task-Verteilung

Im internen Netzwerk könnte auf die Verwendung von Web-Services und WCF verzichtet werden. So müssen womöglich zusätzliche Firewall Rules definiert werden, aber es fallen verschiedene Faktoren weg.

- Kein Overhead von HTTP Daten
- Keine SOAP- bzw. XML-Serialisierung der Service-Aufrufe und der zugehörigen Daten

Da WCF Byte-Arrays Base64 encodiert, würde sowohl das Encodieren und Decodieren entfallen, als auch die Menge an übertragenen Daten minimiert.

Ebenfalls von Vorteil ist das Wegfallen der Notwendigkeit des Einsatzes eines Microsoft IIS als Webserver.

Als Nachteil wird vor allem die erhöhte Komplexität und Wartbarkeit einer eigenen Implementierung angesehen. Durch die eigene Implementierung können viele neue Fehlerfälle eingeführt werden. Ebenfalls kann von weiteren Verbesserungen von WCF durch Microsoft nicht mehr profitiert werden.

4.3 Work Stealing Thread Pool

Um das Netzwerk effizienter zu nutzen und den Overhead pro Task zu minimieren, könnte der Worker „Work Stealing“ betreiben. Dabei holt der Worker immer gleich eine grössere Menge an Tasks auf einmal. Sind keine Tasks mehr auf dem Master vorhanden, so holen sich die Worker von anderen noch beschäftigten Workern Tasks ab, um diese bei sich auszuführen. Des Weiteren wäre es möglich, die Task Results ebenfalls im Bulk dem Master zu übermitteln.

Durch diesen Ansatz wird der Netzwerkoverhead minimiert, da weniger Roundtrips zwischen Worker und Master notwendig sind.

Der Nachteil dieses Ansatzes ist allerdings, dass sich die Worker untereinander kennen müssten, was gegen die Grundidee von autonomen und unabhängigen Workern im System verstösst. Sofort würde eine hohe Komplexität aufgrund des Managementaufwands eingeführt werden.

Ein weiterer Lösungsansatz ist, dass der Master als Vermittler agiert. Somit müssten die Worker nicht untereinander kommunizieren. Allerdings führt auch dieser Ansatz unnötige Komplexität ein, welche als nicht genügend gewinnbringend erachtet wurde, um dessen Anwendung zu rechtfertigen.

5. Glossar

Begriff	Erklärung
CIL	Common Intermediate Language (ehemalig MSIL – Microsoft Intermediate Language)
PackageId	Eine Package Id ist ein eindeutiger Identifier für ein Task Package. Damit kann jederzeit ein Task Package identifiziert werden.
PTC	Portable Task Code: Serialisierter CIL Code eines Tasks.
PTD	Portable Task Data: Serialisierte Kontextdaten für die Ausführung eines Tasks.
Task Package	Ein Task Package wird vom Client verwendet, um dem Master eine Menge an zusammengehöriger Tasks zu senden.
TaskIdentifier	Ein TaskIdentifier enthält nebst einer Package Id, um das Task Package zu identifizieren, auch eine TaskId, um den Task innerhalb des Packages identifizieren zu können. Mittels TaskIdentifier kann jederzeit das Task Package identifiziert werden, zudem der Task gehört sowie seine Position / Index darin.
TPL	Task Parallel Library
WCF	Windows Communication Foundation
Worker Thread	Ein Worker Thread ist Bestandteil eines Workers. Der Worker Thread führt ununterbrochen seine Execution Loop aus. In einem Worker sind für gewöhnlich mehrere Workther Thread tätig.

6. Abbildungsverzeichnis

Abbildung 1 Verteilter Thread Pool 8
Abbildung 2 Ablauf der Taskausführung im Pool 9
Abbildung 3 Systemübersicht 9
Abbildung 4 Aufbau der Task Queue 11
Abbildung 5 Client Service 12
Abbildung 6 Worker Service Schnittstelle 13
Abbildung 7 Skalierung mit blockierenden Requests 15
Abbildung 8 Zeitliches Verhalten mit blockierenden Requests 15
Abbildung 9 Thread-Synchronisation mit Monitor 16
Abbildung 10 Übersicht Messpunkte 21
Abbildung 11 Übersicht Distributed Thread-Pool Testaufbau 22
Abbildung 12 Übersicht HPC Cluster Testaufbau 23
Abbildung 13 Direktvergleich 24
Abbildung 14 Skalierung mit 100 Faktorisierungen 25
Abbildung 15 Skalierung anhand Task Menge 26
Abbildung 16 Overhead Skalierung 27
Abbildung 17 Overhead Total 28
Abbildung 18 Overhead Verteilung 29
Abbildung 19 Code Optimierungen 30

7. Codeverzeichnis

Code 1 Client Service Interface 13
Code 2 Worker Service Interface 13
Code 3 Worker Thread Execution Loop 17
Code 4 Task zur Memory-Leak Reproduktion 18
Code 5 Verteilte Aufgabe 20
Code 6 Optimierte Faktorisierungs-Funktion 30

8. Tabellenverzeichnis

Tabelle 1 Mapping der Thread Pool Komponenten 8
Tabelle 2 Messpunkte Distributed Task Pool 21
Tabelle 3 Messpunkte HPC 23

9. Literaturverzeichnis

[1] R. Fosner, «Scalable Multithreaded Programming with Thread Pools,» Microsoft, 10 2010. [Online]. Available: https://msdn.microsoft.com/en-us/magazine/gg232758.aspx. [Zugriff am 24 03 2015].

[2] L. Bläser, «.NET Task Parallelization in the Cloud: .NET Task Parallelization in the Cloud: Runtime Support for Seamless Distribution of Shared Memory Parallel Tasks. In 4th Workshop on Systems for Future Multicore Architectures (SMFA'14) at Eurosys 2014,» Amsterdam, 2014.

[3] Microsoft, «What Is Windows Communication Foundation?,» 4 2 2011. [Online]. Available: https://msdn.microsoft.com/en-us/library/vstudio/ms731082(v=vs.90).aspx. [Zugriff am 24 3 2014].

[4] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris und D. Orchard, «Web Services Architecture,» W3C, 11 2 2004. [Online]. Available: http://www.w3.org/TR/ws-arch/. [Zugriff am 24 3 2015].

[5] T. Ugurlu, A. Zeitler und A. Kheyrollahi, «Asynchronous HTTP Requests,» in *Pro ASP.NET Web API: HTTP Web Services in ASP.NET (Expert's Voice in .NET)*, Apress, 2013, p. 471.

[6] J. Lowy, «Throttling,» in *Programming WCF Services: Mastering WCF and the Azure AppFabric Service Bus*, O'Reilly Media, 2010, pp. 217-224.

[7] S. Goldshtein, D. Zurbalev und I. Flatow, «Fine-Tuning the ASP.NET Process Model,» in *Pro .NET Performance: Optimize Your C# Applications (Professional Apress) (Expert's Voice in .NET)*, Apress, 2012, pp. 318-319.

[8] Microsoft, «ClientBase<TChannel>-Klasse,» Microsoft, [Online]. Available: https://msdn.microsoft.com/de-de/library/ms576141(v=vs.110).aspx. [Zugriff am 28 04 2015].

[9] Microsoft, «Channel Factory and Caching,» Microsoft, [Online]. Available: https://msdn.microsoft.com/en-us/library/hh314046(v=vs.110).aspx. [Zugriff am 28 04 2015].

[10] Microsoft, «Collectible Assemblies for Dynamic Type Generation,» Microsoft, [Online]. Available: https://msdn.microsoft.com/en-us/library/vstudio/dd554932%28v=vs.100%29.aspx. [Zugriff am 16 05 2015].

[11] Microsoft, «Stopwatch Class,» [Online]. Available: https://msdn.microsoft.com/en-us/library/system.diagnostics.stopwatch%28v=vs.110%29.aspx. [Zugriff am 05 05 2015].

[12] H. IET, «Cluster,» [Online]. Available: http://siet0001.hsr.ch/ietwiki/Pages/Cluster.aspx. [Zugriff am 03 04 2015].

[13] Microsoft, «OpCodes.Conv_I8 Field,» Microsoft, [Online]. Available: https://msdn.microsoft.com/library/system.reflection.emit.opcodes.conv_i8.aspx?f=255&MSPPError=-2147217396. [Zugriff am 23 05 2015].

[14] Microsoft, «BasicHttpBinding Properties,» [Online]. Available: https://msdn.microsoft.com/en-us/library/System.ServiceModel.BasicHttpBinding_properties(v=vs.100).aspx. [Zugriff am 20 03 2015].

[15] Microsoft, «Guid-Struktur,» [Online]. Available: https://msdn.microsoft.com/de-de/library/system.guid%28v=vs.110%29.aspx. [Zugriff am 20 03 2015].

[16] Microsoft, «Monitor Class,» [Online]. Available: https://msdn.microsoft.com/en-us/library/system.threading.monitor(v=vs.110).aspx. [Zugriff am 20 03 2015].

[17] A. Troelsen, «Programmatically Unloading AppDomains,» in *Pro C# 5.0 and the .NET 4.5 Framework (Expert's Voice in .NET)*, Apress, 2012, pp. 644-645.

[18] J. Zander, «Why isn't there an Assembly.Unload method?,» Microsoft, 31 05 2004. [Online]. Available: http://blogs.msdn.com/b/jasonz/archive/2004/05/31/145105.aspx. [Zugriff am 28 04 2015].