Hydra Parallel Programming System

Stefan Zwicklbauer

1. Juli 2009

Inhaltsverzeichnis

1 Einführung

1.1 Entstehung von Multicore CPU's

Hohe Programmiersprachen wie C++ (in den 80er Jahren als Erweiterung der Programmiersprache C von Bjarne Stroustrup bei AT&T entwickelt) oder Java (1996 wurde Version 1.0 von Sun veröffentlicht) wurden schon eingesetzt da konnte man sich effiziente Parallelität in Softwaresystemen nur auf dem Papier vorstellen. Herb Sutter, Vorsitzender des ISO-C++-Standardisierungskomitees und Software-Architekt bei Microsoft wandte sich in einer Rede größenteils an Hardwareentwickler beim In-Stat/MDR's Fall Processor Forum und stellte als Vertreter der Softwareentwicklung unverkenntlich dar, dass durch die Einführung von parallelfähiger Technologien wie Hyperthreading beziehungsweise Dual-Core Prozessoren sich eine Tür in Sachen Performance geöffnet hat. Nicht nur Intel zeigte mit mit seiner gesonderten Abteilung IntelResearch Prototypen von multicorefähigen Prozessoren, sondern auch Firmen wie VIA Technologies produzierten schon dazugehörige Mainboards. Doch ein effizienter Einsatz erfordert neben der entsprechenden Hardware auch die richtige Programmierung der Softwareteile.

1.2 Warum eine neue Sprache?

Hohe Programmiersprachen erlauben zwar den Einsatz von Threads doch heisst dies nicht dass die Anwendung trotz mehrerer CPUs schneller arbeitet. Bei der Programmierung gilt es auch Deadlocks und Livelocks zu verhindern. Eine genaue Planung der Threads ist erforderlich, da durch einen ungeschickten Einsatz genau das Gegenteil erreicht wird. 1996 wurde die OpenMP Schnittstelle eingeführt, ein heutzutage de-facto Standard zur Shared-Memory Parallelisierung. Davon gibt es Implementierung von Fortran, C und C++. Es gibt jedoch keine offizielle Spezifikation von OpenMP für Java! Zur Zeit gibt es nur zwei Implementierungen von OpenMP in Java: JOMP und Jackal. Von JOMP steht die Entwicklung momentan still, da es Probleme mit neuen Java Versionen ab Java 1.4 gibt.
Bekannt ist dass Threads auf keinen Fall immer optimale Performance liefern. Diese ist ohnehin nur schwer möglich da jedes Programm anders aufgebaut ist. Doch wenn man spätere Vergleiche zwischen Hydra und Java Thread Ausführung betrachtet, kann man gut erkennen dass die Performance der Standard Java API unter dem Möglichen liegt! Auch ein Performancevergleich zu JOMP wird später in dieser Arbeit noch aufgezeigt!
Und dann gibts es von Franklin E. Powers, Jr eine Zusatzbibliothek welche im Jahre 2005 und 2006 entwickelt wurde und sich heute Hydra Parallel Programming System nennt. Genau diese Bibliothek wird in diesem Beitrag genauer analysiert.
Programme die in Hydra geschrieben wurden haben den Vorteil, dass sie bei paralleler Hardwarearchitektur auch parallel ausgeführt werden können. Ist keine entsprechende Architektur vorhanden, wird der Code sequenziell ausgeführt

ohne dass der Anwender entsprechendes Fehlverhalten mitbekommen würde, da jegliche Erweiterungen von Hydra ignoriert werden. Somit kann bei Software, bei welcher schon in der Planung feststeht dass Parallelität wahrscheinlich eine Rolle spielen wird (Und das wird sie wohl bei fast jeder neuen Software), bedenkenlos Hydra verwendet werden!

2 Die Sprache Hydra

Ein Java Programm besteht aus Klassen und ihren dazugehörigen Methoden. Die Sprache Hydra verwendet statt Methoden sogenannte „Events“, welche immer atomar in ihrer Ausführung sind!

Anmerkung: **Atomar** bedeutet dass ein Programmabschnitt nur von einem Thread abgearbeitet wird und somit unkritischer ist, was die Synchronisierung angeht.

Vor allem können Beziehungen zwischen Events hergestellt werden, indem man sie in sogannte „Eventgruppen“ anordnet. Insgesamt gibt es 4 verschiedene Arten von Eventgruppen welche alle unterschiedliche Ausführungseigenschaften besitzen und weiter unten genauer beschrieben werden. Die Ausführing der einzelnen Events geschieht mit einem Timeslot Mechanismus. Wenn Events somit gleichzeitig ausgeführt werden sollen, stehen sie bei der Zeitleiste auf der identischen Position. Mit einer unterschiedlichen Positionierung kann man Synchronisierungsprobleme oder ähnliche Probleme vermeiden.
Die Sprache an sich benutzt „Annotations“ um Events und Locks (werden verwendet damit ein Event keine Daten falsch überschreibt) festzulegen. Zusätzlich werden leere Klassen und Methoden zur Gestaltung von Eventsgruppen benutzt.

Vorteil zu OpenMP: Bei einer OpenMP Implementation sind die Extensions normlerweise als Kommentare gekennzeichnet und deswegen in der Regel nicht im Bytecode enthalten. Ein weiterer Punkt ist dass Hydra den Bytecode nicht modifiziert und auch abwärtskompatibel ist. Ein wichtiger Punkt für Anwendungsentwickler ist, dass Hydra mit den meisten Standard IDE's, Compilern und Runtimes arbeitet.

2.1 Events

Der Unterschied zwischen einer normalen Java Methode und einem Event in Hydra liegt darin dass ein Event bei seinem Aufruf nicht sofort ausgeführt wird. Der Programmierer legt mit der Verwendung von Eventgruppen und Locks die Ausführungszeit(Timeslot) fest. Bei dem Aufruf eines Events wird eine Anfrage an den Scheduler gestellt und dieser entscheidet je nachdem ob alle anderen vorherigen Events im Timeslot bereits abgearbeitet wurden, ob der Code ausgeführt werden kann oder nicht. Um Parallelität zu erreichen müssen sich 2 oder mehrere Events im gleichen Timeslot befinden. Je mehr Prozessoren zur

Verfügung stehen, desto mehr Events können zur gleichen Zeit ausgeführt werden.
Insgesamt gibts es 3 verschiedene Arten von Timeslots: Logical Timeslot, Prescheduled event group timeslots and Runtime timeslots.

- Logical time slot: Durch das Einsetzen von Eventgruppen und Locks werden die einzelnen Events automatisch von Hydra in einer Zeitleiste platziert, welche den Ausführungszeitpunkt in Relation zu anderen Events wiedergeben. Auf dieses Timeslotsystem hat der Programmierer keinen direkten Einfluss!

- Pre-scheduled event group time slot: Beim Verwenden von einer pre-scheduled Eventgruppe ist es dem Programmier direkt möglich Einfluss auf die Zeitleiste zu nehmen, indem er relativ zu anderen Events festlegt wann ein Event ausgeführt werden soll.

- Runtime time slot: Es handelt sich hier um ein Mapping der obigen beiden Timeslots. Mit dem Einbinden der jeweiligen Lockobjekte entscheidet die Laufzeitumgebung intern wann die Events schließlich ausgeführt werden sollen. Der Benutzer hat keinen Einfluss darauf.

2.2 Eventgruppen

Einfach lässt sich sagen dass sich durch die Platzierung verschiedener Events in einer Gruppe die Ausführungsreihenfolge festlegen lässt. Auch eine Platzierung von mehreren Eventgruppen in einer Anderen ist möglich. 4 unterschiedliche Typen an Eventgruppen gibt es in Hydra:

1. Parallele Eventgruppen: In einer mit „parallel“ gekennzeichneten Gruppe können alle Events parallel ausgeführt werden. Allerdings ist es möglich durch Locks anzugeben dass bestimmte Events nicht willkürlich ausgeführt werden können, was aber aufgrund der Lockperformance verhindert werden sollte. Ein anderes Eventgruppendesign ist mit hoher Wahrscheinlichkeit schneller in ihrer Auführung!

2. First-In, First-Out: Je nachdem in welcher Ordnung die einzelnen Events zur Eventgruppe hinzugefügt(gescheduled) wurden, so werden die Events auch ausgeführt, da sie auf verschiedenen Timeslots liegen.

3. Sequenziell: Der sequenzielle Typ ist von den 4 möglichen Kritieren die Standardtyp. Hier erfolgt die Abarbeitung genauso als wenn das Programm mit normalen Methoden geschrieben worden und die Methoden in sequenzieller Reihenfolge aufgerufen worden wären. Auch hier gibt es keine Parallelität.

4. Pre-scheduled: Im Gegensatz zu den anderen Methoden kann hier der Programmierer direkt festlegen wann die einzelnen Events in Relation zu Anderen aufgerufen werden sollen. Es kann als eine Kombination zwischen

dem parallelen und dem FIFO Typ angesehen werden. Parallele Eventgruppen welche von anderen parallenen Eventgruppen abhängig sind, werden auf verschiedene Timeslots gesetzt. Der Programmierer kann auf der Timeslot Leiste praktisch nach vorne und nach hinten wandern um Events auszulösen. Der Vorteil ist hier, dass man sich aufgrund der komplexen Timeslot Strategie im Hintergrund nicht um Lock Objekte kümmern muss, da diese automatisch entsprechend gesichert sind.

2.3 Das Verwenden von Locks

Wie bei normalen Java Methoden, gibt es auch für Events Parameter! Diese können mit 4 verschiedenen Locktypen versehen werden. Gekennzeichnet werden diese mit einer @-Annotation mit dem Namen des Locktyps.
Es gibt zahlreiche Lockeigenschaften von Hydra, welche hier den Rahmen allerdings sprengen würden. Trotzdem sind folgende Eigenschaften besonders relevant:

- Zuerst wird eine Anfrage gestellt, ob ein Lockobjekt gerade verfügbar ist. Je nachdem wird der entsprechende Event vom Scheduluer richtig auf den Timeslots zur Ausführung platziert.
- Locks können nicht während der Ausführung des Events verändert werden.
- Alle Locks werden nach der Ausführung des Events automatisch zurückgestellt.
- Die Laufzeitumgebung kann die vom Programmierer gesetzten Locks bei der Ausführung noch optimieren.

Als Programmierer hat man 4 verschiedene Lockvarianten welche man auf Objekte setzen kann:

1. Pass: Pass heisst nur dass der Parameter für das Event freigegeben ist und für andere Events nicht Lese/Schreibgesperrt werden muss.
2. Read: Sperrt den Parameter für Lesezugriff. Das heisst in anderen Worten dass Dieser in einem anderen (parallelen) Event nicht beschrieben werden darf.
3. Write: Sperrt den Parameter für andere Events, da ansonsten zur gleichen Zeit 2 Events diesem Parameter einen Wert zuweisen könnten. Stellt sicher dass nur 1 Schreiboperation zur gleichen Zeit gemacht werden kann.
4. ReadWrite: Sperrt den Parameter für Read und Write Operationen. Meistens entspricht Read Write dem normalen Write-Lock. Die Laufzeitumgebung kann unter Umständen die Locks optimieren.

Normalerweise ist die richtige Anordnung der Eventgruppen die effizienteste Form der Synchronisierung, es kann aber unter Umständen vorkommen dass

Programmabschnitte nicht komplett von unabhängigen Eventgruppen ausgedrückt werden können. In diesem Falle muss der Programmierer mit dem Benutzen von Locks selbst Hand anlegen.
Falls ein Objekt mit einem Lock angefordert wird(zum Beispiel beim Aufruf eines Events), prüft die Laufzeitumgebung ob Konflikte auftreten können und im besten Fall kann das Objekt für Parallelität im selben Timeslot freigegeben werden. Sonst wird die Ausführung des Events auf den nächsten Timeslot verlegt, bei dem das entsprechende Objekt wieder frei ist.
Die Laufzeitumgebung benutzt folgenden Algorithmus ob ein Event auf dem gleichen Timeslot wie ein Anderer gesetzt werden kann oder ob dieser eben Aufgrund von einem Lock auf einen nachfolgenden Timeslot gelegt werden muss:

```
ScheduleFor = 0

FOREACH Object IN ParametersOf(Event)
    IF PreviousLockFor(Object) == READ THEN
        IF RequestedLockFor(Object) == READ THEN
            Available = PreviousTimeSlotFor(Object)
        ELSEIF RequestedLockFor(Object) == WRITE THEN
            Available = PreviousTimeSlotFor(Object) + 1
        ELSEIF RequestedLockFor(Object) == READWRITE THEN
            Available = PreviousTimeSlotFor(Object) + 1
        ENDIF
    ELSEIF PreviousLockFor(Object) == WRITE THEN
        Available = PreviousTimeSlotFor(Object) + 1
    ELSEIF PreviousLockFor(Object) == READWRITE THEN
        Available = PreviousTimeSlotFor(Object) + 1
    ENDIF
    IF Available > ScheduleFor THEN
        ScheduleFor = Available
    ENDIF
NEXT Object

FOREACH Object IN ParametersOf(Event)
    SetPreviousLockFor(Object, RequestedLockFor(Object))
    SetPreviousTimeSlotFor(Object, ScheduleFor))
NEXT Object
```

2.4 Befehle und Zusammenfassung

Grundlegend lässt sich sagen, dass es 3 verschiedene Hauptaufgaben für den Programmierer gibt:

- Das Erstellen einer Eventgruppe: Um eine neue Eventgruppe zu erstellen, wird die entsprechende statische Methode (z.B.: FIFO()) aus der Klasse EventGroup aufgerufen. Diese liefert die entsprechende Gruppe zurück welche der Programmierer uneingeschränkt nutzen kann. Wenn Hydra gestartet wird, wird eine Default-Eventgruppe vom sequenziellen Typ erstellt.

- Die Anpassung einer Eventgruppe mit den entsprechenden Events: Mit den verschiedenen Methodenaufrufen kann man als Programmierer Events an Eventgruppen oder auch Eventgruppen untereinander verknüpfen.
- Das Verwenden von Locks bei Parametern zur sicheren parallelen Ausführung von Events: Die entsprechende Annotation wird direkt vor dem Typbezeichner im Methodenkopf geschrieben.

Nachfolgend eine kleine Befehlsübersicht welche dem Programmierer bei der Erstellung von Events und Eventgruppen zur Verfügung steht

Event group creation	Hydra command
FIFO	EventGroup.FIFO()
Parallel	EventGroup.Parallel()
Pre-scheduled	EventGroup.Prescheduled()
Sequential	EventGroup.Sequential()
Event group manipulation	**Hydra command**
Add event group *group* to the event group that events are currently being added to.	*group*.AddAsChildEventGroup()
Add event group *a* to event group *b*	*a*.AddAsChildEventGroupTo(*b*)
Add events to the event group *group*	*group*.AddEventsToThis()
Retrieve the event group that events are being added to.	EventGroup.GetAddingTo()
Retrieve the event group that the currently executing event belongs to.	EventGroup.GetCurrent()
Locking parameters	**Hydra command**
Pass	@pass
Read	@read
Write	@write
ReadWrite	@readwrite

3 Kompilierung und Laufzeitumgebung

Die Spache Java folgt dem Motto „Write once, run everywhere“. Das Gleiche gilt auch für Hydra. Deren Laufzeitumgebung sorgt dafür dass der Programmcode auch auf der installierten API ohne Probleme läuft.

3.1 Der Unterschied zwischen der Kompilierung und Laufzeitumgebung bei der Standardvariante und Hydra

Grundsätzlich wird ein Programm welches für parallele Ausführung entwickelt wurde, zuerst vom Compiler der entsprechenden Javaextension kompiliert. Anschließend werden diese Javadateien vom normalen Javacompiler in normale Classdateien übersetzt. Als gutes Beispiel kann man in diesem Falle JOMP verwenden. Hier wird zuerst der JOMP Code in gewöhnlichen Javacode übersetzt. Falls dies vom Programmierer nicht explizit gemacht wird, werden alle JOMP Erweiterungen ignoriert und der Code normal ausgeführt.
Das Problem ist dass wenn der Programmierer Programmcode füer verschiedene Laufzeitumgebungen entwickelt, er den Programmcode immer neu für alle verschiedenen Umgebungen kompilieren muss. Zusätzlich benötigt man ein kleines Programm welches die aktuelle Umgebung des Benutzers herausfindet und zur entsprechenden Version schaltet. Anders läuft das Ganze bei Hydra ab. Hier

werden zuerst die normalen *.java Dateien in normalen class-Dateien vom Compiler umgewandelt. Die Hydra Annotationen werden wie üblich mit in die class Dateien übernommen. Wenn der Benutzer nun diese Dateien mit einer normalen Java Virtual Machine ausführt, ignoriert Diese die Hydra Anweisungen und das Programm wird ohne Hydra gestartet.
Falls der Benutzer die Classdateien jedoch mit der Hydralaufzeitumgebung ausführt, wird der Code rekompiliert um sich der Hydraarchitektur anzupassen. Folgende Schritte werden dabei ausgeführt:

- Die Laufzeitumgebung geladen (Wird bei Hydra mitgeliefert!)
- Die Laufzeitumgebung wird initialisiert und konfiguriert den Alchemist Compiler (Rekompiliert die class-Dateien)
- Anschliessend rekompiliert Dieser den vorhandenen Programmcode
- Der Alchemist Compiler liefert eine Kopie des Programms zurück
- Schliesslich wird der neue Programmcode ausgeführt

Der Benutzer bekommt von diesem Prozess nichts mit. Das Ganze kann mit folgender Befehlszeile gestartet werden:

java Hydra.Runtime.Start NameDesProgramms

4 Vergleiche zu anderen Sprachen

Zuerst sei gesagt dass jede andere explizite paralle Sprache seine eigenen Stärken und Schwächen hat. Im Vergleich zu JOMP hat Hydra keine sogenannten High-Level-Konstrukte, mit denen man zum Beispiel Schleifenparallelisierung verwirklichen kann. Hier wird einfach nur durch Abhängigkeiten verschiedener Programmteile eine gewisse Reihenfolge in den Ablauf gebracht.

- Das Verbinden zwischen sequenziellen und parallen Programmteilen: Der Programmcode innerhalb von Events ist atomar und wird immer sequenziell ausgeführt. Events jedoch können auch immer parallel ausgeführt werden. Das erbringt den Vorteil, dass während der Ausführung eines Events keine Synchronisierungsprobleme auftreten können, da Hydra keine neuen Events durch ein Anderes startet bevor der komplette Teil ausgeführt worden ist. Jedoch ist die Mischung in anderen Sprachen wie JOMP flexibler. Es lassen sich in einem parallen Programmabschnitt, weitere parallele und sequentielle Abschnitte vermischen.

- Schleifenparallelisierung: Grundsätzlich kann man in Hydra jede Art von Schleife ohne Probleme oder zusätzliche Konstrukte realisieren: for-Schleifen, for-each-Schleifen, while-Schleifen und do-while-Schleifen. Dies kann man auf verschiedene Arten bewerkstelligen. Sequenzielle Schleifen kann man einfach wie im normalen Java nur innerhalb von Events realisieren. Bei gewollter Parallelität werden Schleifendurchläufe auf verschiedene Events aufgeteilt. Der Benutzer hat hier die volle Kontrolle, ist jedoch umständlich.
  Im Gegensatz zu Hydra gibt es bei JOMP wiederrum spezielle Konstrukte, bei denen die Schleife, speziell auf mehrere Threads aufgeteilt, parallel abgearbeitet wird.

- Parallele Aufrufe: In Hydra werden Methoden, welche parallel ausgeführt werden, wie weiter vorne beschrieben durch die Anordnung in Eventgruppen heraus aus ausgeführt. In JOMP werden Aufrufe, welche parallel stattfinden sollen aus sogenannten parallelen Regionen heraus gestartet.

Das nachfolgende Beispiel soll ein kleiner Vergleich zwischen der hier behandelten Sprache Hydra und JOMP sein. Hier geht es nicht um den eigentlichen parallelen Code sondern um dessen Aufruf bzw. Integration:

Hydra version:

```
public @event void EventWithLoop() {
        EventGroup group = EventGroup.Parallel();
        group.AddAsChildEventGroup();
        group.AddEventsToThis();
        for (int a = 0; a < 10; a++)
                DoSomeWork(a);
}

public @event void DoSomeWork(@pass int a) {
        // Insert code for doing work here.
}
```

JOMP version:

```
public void MethodWithLoop() {
        //omp parallel for
        for (int a = 0; a < 10; a++)
                DoSomeWork(a);
}

public void DoSomeWork(int a) {
        // Insert code for doing work here.
}
```

Dieser Code soll kurz die verschiedenen Ausführungen deutlich machen. In einem Event wird eine neue parallele Eventgruppe erstellt, welcher wiederum in einer for-Schleife verschiedene Events hinzugefügt werden, die zeitgleich abgearbeitet werden sollen. Was bei diesem Programmteil erwähnenswert ist, ist der Ausführungszeitpunkt der Events. In diesem Beispiel kann es unter Umständen sein, dass die Schleifenevents erst ausgeführt werden nachdem jeglicher Nicht-Hydra-Programmcode vor und nach der Schleife bearbeitet wurde. Dies lässt sich aus diesem Beispiel aber nicht klar erkennen, da man nicht weiss an welche Art von Eventgruppe die parallele Schleifen-Event-Gruppe gehängt wurde.

Die Frage ist warum bei Hydra eigentlich so ein relativ komplexes Verfahren eingesetzt wird. Diese Konstrukte wurden entwickelt um beim Abarbeiten des Codes einen Deadlock zu vermeiden. Durch das alleinige Verwenden von Lockobjekten und Eventgruppen kann dieses Problem umgangen werden.

5 Beispiele

5.1 Ausführungsreihenfolge verschiedener Eventgruppen

Es sei folgender Pseudocode gegeben:

```
ScheduleAllEvents():
        SCHEDULE SayHello()
        SCHEDULE SayGoodBye()
SayHello():
        DISPLAY "Hello"
        SCHEDULE SayHowAreYou()
SayHowAreYou():
        DISPLAY "How Are You?"
SayGoodBye():
        DISPLAY "Good Bye"
```

Das Programm kann je nach Art der Eventgruppe die 3 Sätze in verschiedener Reihenfolge ausgeben!

- Ohne Hydra: Falls der Code mit der normalen Javaruntime aufgerufen wird, wird wie erwartet die richtige Reihenfolge in Form von „Hello“, „How are you?“ und „Goodbye“ aufgerufen, da die einzelnen Programmteile in normaler Reihenfolge abgearbeitet werden.
- Bei Hydra und einer sequenziellen Eventgruppe ist das Ergebnis ähnlich. Aber die Bearbeitung erfolgt anders. Zuerst werden in den ersten beiden Zeilen die Events SayHello() und SayGoodBye() an den Scheduler übergeben. Dieser führt Zeile 1 aus und Zeile 2 wird geschedult aber noch nicht ausgeführt. Bei der Ausführung von Zeile 1 wird „SayHowAreYou“ als nächstes geschedult und auch vor „SayGoodBye“ ausgeführt. So ergibt sich die gleiche Reihenfolge wie oben.
- Bei einer FIFO-Eventgruppe ist es jedoch Anders. Hier werden die einzelnen Events gemäß ihrer FIFO-Ordnung ausgeführt. Dies vertauscht die „GoodBye“ und „How are you “- Ausgabe, da GoodBye früher geschedult wird. (In der 2ten Programmzeile!).
- Bei einer parallelen Ausführung kann man zuvor keine Aussage über die Reihenfolge machen. Diese kann sich bei jedem Programmstart ändern. (Vorhandene Möglichkeiten: „Hello, HowAreYou, GoodBye“, „Hello, GoodBye, HowAreYou“ oder „GoodBye, Hello, HowAreYou“).
- Falls die Eventgruppe die Eigenschaft pre-scheduled hat, passiert hier genau das identische wie bei einer parallelen Eventgruppe, da hier kein Gebrauch von der „forward“ und „backward“ Eigenschaft gebraucht gemacht wurde und somit alle Events auf dem identischen Timeslot liegen.

5.2 Der Merge-Sort-Algorithmus

Der Mergesort Algorithmus ist gut geeignet zum Sortieren von Arrays. Zuerst wird das Array in einzelne Teilarrays aufgeteilt bis die Länge der Subarrays einen bestimmten vorgegeben Thresholdwert erreicht haben. Anschließend werden Diese sortiert und wieder in richtiger Reihenfolge zusammengefügt. Das „splitten“ und „mergen“ geschieht rekursiv.

```
ScheduleMergeSort(Array):
    CREATE NEW Prescheduled Event Group
    ScheduleMergeSort(Array, 0, SizeOf(Array) - 1)

ScheduleMergeSort(Array, A, B):
    BACK
    IF A = B THEN
        RETURN
    Difference = B - A
    Middle = (A + B) / 2
    IF Difference >= G  THEN
        ScheduleMergeSort(Array, A, Middle)
        ScheduleMergeSort(Array, Middle + 1, B)
        SCHEDULE Merge(Array, A, Middle, B)
    ELSE
        SCHEDULE MergeSort(Array, A, Middle, B)
    ENDIF
    FORWARD
```

Der Forward und Backward Befehl wird deswegen eingesetzt, da der „Merge“-Aufruf nicht zur selben Zeit ausgeführt werden kann. Da hier parallele Abläufe vorhanden sind, eignet sich der Algorithmus gut zur Demonstration der Hydra Performance. Diese wird man in einem späteren Kapitel zeigen.

6 Performance und Benchmarktests

6.1 Verfügbare Rechnersysteme

Folgende Rechnersysteme standen dem Testteam um John Wiley & Sons, Ltd zur Verfügung:

1. Standard Desktoprechner: Ein normaler handelsüblicher Rechner indem ein normaler AMD Dual-Core CPU arbeitet. Kann 2 Threads gleichzeitig bearbeiten.

2. Workstation: Intel Xeon 2.8 GHZ mit handelsüblichen 2GB Arbeitsspeicher. Vergleichbar mit heutigen Desktop Pc's. 4 Threads können aufgrund von Hyperthreading parallel ausgeführt werden.

3. Serversystem: Es wird ein Sun Enterprise 450 Server verwendet. Hier wird auf mehrere (4) Kerne anstatt Kerntaktung (400 MHZ/Kern) gesetzt. 4 Threads können auf jeweils einem Kern ausgeführt werden.

6.2 Benchmarking mit BAPS

Im HydraPPS PR 2 wird bereits ein Benchmark mit dem Namen BAPS (Benchmarking and Profiling System) mitgeliefert, welcher nicht speziell für Hydra sondern auch auf andere normale oder parallele Javaarchitekturen anwendbar ist. Der Benchmark wird automatisch mehrere Male ausgeführt um grössere Abweichungen abzurunden und einen Schnitt zu errechnen. Zudem kann der Benutzer per Kommandozeile etwaige Einstellungen am Benchmarking vornehmen.

6.3 Micro & Macro-Benchmarks

Zuerst gilt es die grundsätzlichen Unterschiede der beiden Benchmarks zu analysieren.

- Microbenchmark: Hier wird vorrangig ein bestimmter Ausschnitt eines Programmes, sprich Programmcode, auf Performance getestet. Es wird hier in der Regel das gleiche Gesamtsystem verwendet um herauszufinden welche Algorithmusimplementierung die Effizienteste ist. Bei uns muss aufgrund der verschiedenen Paralleltechnologien jeweils eine andere Implementierung des Mergesortalgorithmus getestet werden. Als Testcode, welcher den Unterschied zwischen normaler Java Runtime und Hydra herauskristallisieren soll, wird hier der vorher besprochene MergeSort Algorithmus verwendet indem ein Array mit 500000 Integer Einträgen sortiert wird:

	Desktop		Workstation		Server	
Test	Time (ms)	Faster	Time (ms)	Faster	Time (ms)	Faster
MergeSort with Java	152.04	× 1.00	155.66	× 1.00	865.20	× 1.00
MergeSort with JOMP	79.17	× 1.92	123.29	× 1.26	458.81	× 1.89
MergeSort with Hydra	92.42	× 1.65	72.82	× 2.14	305.06	× 2.84

- Macrobenchmark: Im Gegensatz zum Microbenchmark wird hier das Gesamtsystem (Rechner, JVMs usw.) getestet. Der eigentliche Algorithmus soll hier identisch bleiben. In unserem Fall kann kein JOMP getestet werden, da hier der Code anders implementiert werden würde. Als Testalgorithmus kommt hier also nicht der MergeSort Algorithmus zum Einsatz sondern der bei Hydra schon enthaltene Hydra Definition Generator!

  Der vom Definition Generator erstellte Code ist als Teil des Alchemist-Compiler anzusehen. Die entstehenden „Definitionen“ sind eine schnell zu benutzende Form von Reflection!

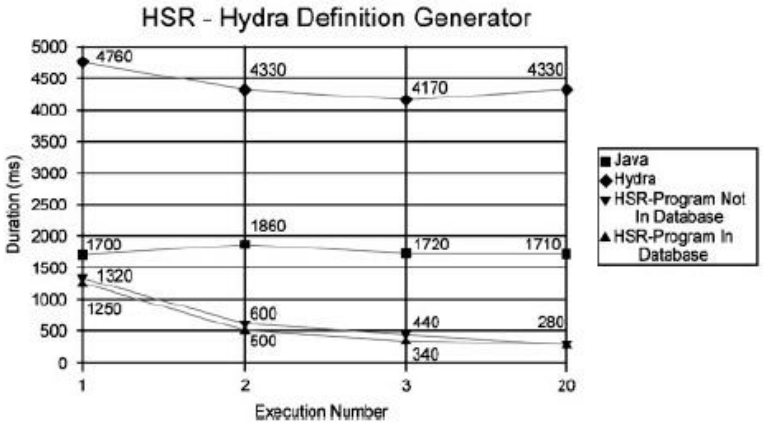
	Desktop		Workstation		Server	
Test	Time (ms)	Faster	Time (ms)	Faster	Time (ms)	Faster
Definition generator with Java	247.27	× 1.00	325.36	× 1.00	1318.82	× 1.00
Definition generator with Hydra	211.96	× 1.17	233.35	× 1.39	877.34	× 1.50

Hier kann man eindeutig erkennen dass der Performanceunterschied nicht wie bei Mergesort deutliche Ausmaße annimmt. Das liegt zum Einen daran dass

durch viel Input/Output der Dateien keine Parallelität erzeugt werden kann. Zudem ist die Java Reflection API eher schlecht für parallele Implementierungen zu verwenden.

7 Runtime Startup Overhead und HSR

Alle im vorherigen Kapitel durchgeführten Benchmarktests wurden mit BAPS erstellt. Aus diesem Grund sind auch die Start und Stopzeiten des Programms nicht in das Ergebnis miteingeflossen. Normalerweise muss beim Starten der Software zuerst die Runtime geladen werden und alle Klassen müssen bei Hydra geladen und kompiliert werden. Dieser Vorgang kann unter Umständen einige Sekunden dauern.
Aus diesem Grund wurde ein Feature mit dem Namen Hydra Stay Resident speziell für Hydra entwickelt um die extrem lange Ladezeit zu umgehen. Dieses erlaubt die Speicherung der einmal ausgeführten Klassen im Arbeitsspeicher des Rechners. Zusätzlich wird die komplette Hydra Laufzeitumgebung im Speicher gehalten. Falls Hydra nun beim Starten des Rechners automatisch ausgeführt wird, muss sich der Benutzer um nichts kümmern.
Als Testbeispiel wird hier eine Javaklasse mit einer leeren Main-Methode verwendet. Ein Script gibt die Zeit vor und nach dem Starten der Main aus. Die normale JVM benötigte ungefähr 170 ms. Bis die Hydra Runtime geladen wurde vergingen 2,36 s. Nachdem Hinzuschalten von HSR konnte unsere einfache Klasse nach der ersten Programmausführung mit einer normalen JVM 17 mal so schnell und mit der HydraRuntime 236 mal so schnell geladen werden.
Als zweites Beispiel verwenden wir den Hydra Definition Generator. Dieser konnte mit HSR ähnliche Geschwindigkeitsschübe zeigen, wie folgendes Diagramm bestätigt:



8 Quellen

http://hydrapps.org/
http://arstechnica.com/old/content/2005/10/5492.ars
http://www.amitysolutions.com.au/documents/Threads-technote.pdf
http://www.angelikalanger.com/Articles/EffectiveJava/21.MicroBenchmarking/21.MicroBenchmarking.html

ACM Symposium on Parallel Algorithms and Architectures
The Hydra Parallel Programming System Paper by John Wiley & Sons, Ltd