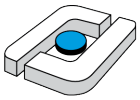

Hochschule Osnabrück
University of Applied Sciences

Parallelisierung mit OpenMP

Martin Helmich (martin.helmich@hs-osnabrueck.de)
Oliver Erxleben (oliver.erxleben@hs-osnabrueck.de)

Hochschule Osnabrück
Ingenieurswissenschaften und Informatik
Informatik - Mobile und Verteilte Anwendungen

7. Januar 2013

Inhaltsverzeichnis

1. OpenMP

Open Multi-Processing (kurz: OpenMP) ist eine Programmierschnittstelle für die Sprachen C/C++ und Fortran, welche seit 1997 von unterschiedlichen Hardware- und Compilerherstellern entwickelt wird. Ziel von OpenMP, welches mittlerweile den Versionsstand 3.1 erreicht hat, ist es ein portables und zugleich paralleles Programmiermodell für Shared-Memory-Architekturen[1] zur Verfügung zu stellen. Es setzt sich aus Compilerdirektiven, Bibliotheksfunktionen und Umgebungsvariablen zusammen. Anders als viele alternative Ansätze zur Parallelisierung von Programmen sind nur wenige Änderungen an sequenziellen Programmen notwendig um parallele Abläufe zu implementieren. Auch die Lesbarkeit des parallelisierten Quelltextes wird im Vergleich zu anderen alternativen Ansätzen stark verbessert.[2]

Ein weiteres Plus für den Einsatz von OpenMP ist die weite Verbreitung von Multi-Core-Rechnern und die Implementierung in vielen verbreiteten Compilern. So ist OpenMP im GCC seit der Version 4.2, im Visual Studio C/C++ Compiler seit der Version 2005 oder im Intel C/C++-Compiler seit Version 8 verfügbar. Compiler die keine OpenMP-Unterstützung bieten, ignorieren aufgrund der Pragma-Compilerdirektiven die Ausführung als parallelisierte OpenMP-Implementierung, was zu einer guten Portabilität führt.

Ursprünglich wurde OpenMP für den Bereich High Performance Computing entwickelt, wo Programmcode meist viele Schleifen enthält. Demnach ist die Parallelisierung von Schleifen die Hauptaufgabe von OpenMP und die meisten Vorteile können an dieser Stelle von OpenMP aufgezeigt werden.

1.1. Merkmale von OpenMP

Mit OpenMP wird ein hoher **Abstraktionsgrad** erreicht, da Threads nicht durch einen Programmierer initialisiert, gestartet oder beenden werden müssen. Wie bereits in der Einführung erwähnt, beitet OpenMP eine gute **Portabilität** des Programms an, zum einen ist OpenMP in vielen Mainstream-Compilern implementiert, zum anderen können Compiler ohne OpenMP-Unterstützung das Programm ohne Veränderung kompilieren, da die Compilerdirektiven durch Pragmas eingesetzt und ignoriert werden können. Auch bleibt der sequenzielle Programmcode vollständig erhalten, wenn nicht mittels OpenMP kompiliert wird. Der Quelltext kann somit **Schrittweise parallelisiert** werden.

1.2. Ausführungsmodell

Die parallele Ausführung erfolgt durch Threads auf dem Fork-/Join[3]-Ausführungsmodell. Zu Beginn eines Programms ist nur ein Thread aktiv, der sog. *Master Thread*. Sobald

[1] Shared-Memory-Architektur:
[2] Vgl. **?**,
[3] Fork/Join:

bei der Programmausführung die Direktive *#pragma omp parallel { ... }* erreicht wird, gabelt sich die Ausführung in Threads auf. Die erstellten Threads werden als *team of threads* bezeichnet[4]. Für OpenMP, bzw. der Entwicklung mit OpenMP stellen Threads einen Kontrollfluss mit gemeinsamen Adressraum dar.

Die schließende geschweifte Klammer ist zudem ein Synchronisationspunkt, andem das team of threads auf alle Teammitglieder wartet. Die Abbildung **??** stellt einen möglichen Ablauf mit OpenMP dar.

Sofern nicht anders angegeben verwenden alle Threads eines Thread-Teams einen gemeinsam genutzten Adressraum und können somit auf alle Variablen eines parallen Codeabschnitts zugreifen, über die auch die Kommunikation zwischen den Threads ermöglicht wird. Folgende Klauseln für Variablen können eingesetzt werden:

- *shared*/*private*: Threads können explizit als gemeinsame Variable oder für jeden Thread private Variable deklariert werden.
- *firstprivate*/*lastprivate*: Diese Klauseln erlauben die Initialisierung, bzw. Finalisierung der Variable beim Ein- und Austritt in oder aus einem parallen Bereich.
- *default*: Das Standardverhalten kann verändert werden.
- *reduction*: Eine spezielle gemeinsam genutzte Variable über die mehrere Threads Wrte zusammentragen können.

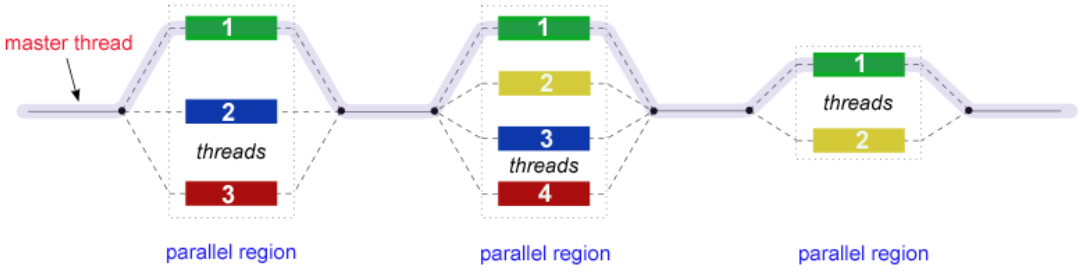

Abbildung 1: Fork-Join-Prinzip
Entnommen aus OpenMP Tutorial (siehe **?**,)

1.3. Parallelisierung von Schleifen

Nach der Klassifikation von Flynn kann OpenMP als Single Programm Multiple Data (SPMD) beschrieben werden. Demnach führt jeder Thread der gestartet wurde den

[4]Die Implementierung von OpenMP entscheidet über die Art der Threads die erstellt werden. Es könnten Threds auf Basis der PThreads-Bibliothek oder aber auch als vollwertige Shared-Memory-Prozesse umgesetzt sein.

Schleifenblock aus. Dabei hat jeder Thread seine eigene Iteration und Teilmenge von Daten.*(Vgl. siehe* **?**, *, Kapitel 3.1: Parallelität auf Schleifenebene)*

Ein team of threads wird mittels *#pragma omp parallel* gestartet. Zur Veranschaulichung sei das Listing 1 gegeben.

Listing 1: pragma omp parallel - Beispiel

```
1 #include <stdio.h>
2 #include <omp.h>
3
4 int main(int argc, char *argv[]) {
5   #pragma omp parallel
6   {
7     int i;
8     for (i = 0; i < 10; i++) { /* some block code */ }
9   }
10  return 0;
11 }
```

Die For-Schleife in Zeile 8 wird von allen vom System zur Verfügung stehenden Threads ausgeführt. Allerdings teilen sich die Threads nicht die Arbeit, sondern jeder Thread führt die Schleife komplett für sich aus. Um eine Schleife parallel auszuführen, muss über eine weitere Direktive, wie in Listing 2 zu sehen, eingefügt werden.

Listing 2: pragma omp parallel for - Beispiel

```
1 #include <stdio.h>
2 #include <omp.h>
3
4 int main(int argc, char *argv[]) {
5
6   int tid;
7   #pragma omp parallel private(tid)
8   {
9     int i;
10    #pragma omp for
11      for (i = 0; i < 10; i++) { tid = omp_get_thread_num(); printf("%
            i, from %i\n", i, tid);}
12  }
13  return 0;
14 }
```

In diesem Beispiel wird der zu parallelisierende Bereich eine Variable *tid*, die threadId, übergeben. Jeder Thread erhält diese Variable, die nicht gemeinsam genutzt werden kann (private). Mittels *#pragma omp for* wird dem Compiler mitgeteilt das die darauffolgende Schleife parallel durch das team of threads im parallelen Bereich ausgeführt werden soll.

Als Vorraussetzung für parallelisierbare Schleifen fordert OpenMP, dass Schleifen in kanonischer Form geschrieben sein müssen. Die kanonische Form liegt wenn folgende Bedingungen erfüllt sind:

- Anzahl der Schleifendurchläufe muss berechnbar sein und darf sich innerhalb des Schleifenblocks nicht ändern. Ebenso dürfen sich Start und Endwert während des Schleifendurchlaufs nicht ändern.
- Es sind nur boolsche Vergleichoperator im Schleifenkopf erlaubt.
- Das Inkrement im Schleifenkopf darf nur aus Addition oder Subtraktion inkrementiert oder dekrementiert werden.
- **break**-Anweisungen dürfen nicht im Schleifenrumpf verwendet werden. Allerdings kann eine Iteration durch eine **continue**-Anweisung übergangen werden oder das gesamte Programm mittel **exit** verlassen werden.

1.4. Synchronisation

Wie im Kapitel 1.2 bereits erläutert wird ein team of threads durch ihren gemeinsamen Adressraum synchronisiert. Dabei ist die Reihenfolge der Bearbeitung der Arbeit nicht "vorhersehbar". Um einen kritischen Adressraum innerhalb eines parallelen Abschnitts zu kennzeichnen, bietet OpenMP ein weitere Direktive an: *#pragma omp critical { ... }*. Der von der Klammer umschlossene Code wird demnach von nur einem Thread ausgeführt.

2. Untersuchung des Laufzeitverhaltens von OpenMP im Vergleich zur Intel TBB

2.1. Zielsetzung und Vorgehen

Zielsetzung dieses Abschnittes ist es, die Möglichkeiten zur Parallelisierung eines sequentiellen Programms mit OpenMP sowie die Leistungsfähigkeit und Skalierbarkeit von OpenMP im Vergleich zu Parallelisierungslösungen anderer Hersteller zu untersuchen.

Als Beispiel dient dazu unser Beitrag zum von Intel ausgerichteten *Accelerate Your Code*-Programmierwettbewerb vom November 2012. Der Wettbewerbsbeitrag verwendet die Intel *Threading Building Blocks*-Bibliothek (kurz TBB). Für diese Arbeit wurde der Beitrag nach OpenMP portiert und anschließend das Laufzeitverhalten beider Programme untersucht und verglichen.

Die Aufgabenstellung des Programmierwettbewerbs liegt in Anhang A bei. Die Quelltexte beider Programme liegen dieser Arbeit auf Datenträger bei.

2.2. Auftretende Probleme

Bei der Portierung des Programms von TBB auf OpenMP fiel vor allem der geringere Sprachumfang der OpenMP-Erweiterung negativ auf.

Beispielsweise bietet die TBB eine (wenn auch nicht sonderlich effiziente)[5] parallele While-Schleife in Form der `tbb::parallel_do`-Methode. Ein vergleichbares OpenMP-Konstrukt existiert nicht. Das Problem konnte gelöst werden, indem der iterative, mit einem Stack arbeitende, Algorithmus zurück in eine rekursive Form überführt wurde. Diese konnte anschließend durch Task-Parallelität ohne Probleme parallelisiert werden (Task-Parallelität wird sowohl von der TBB als auch von OpenMP unterstützt).

Ein weiteres Problem fiel bei der Implementierung des task-parallelen Algorithmus auf: Task-Parallelität wurde erst in der Version 3.0 des OpenMP-Standards eingeführt.[6] Obwohl diese Version bereits 2008 veröffentlicht wurde,[7] wird sie noch nicht von allen Compilern unterstützt. So verwendet MacOS X beispielsweise auch 2012 noch die GCC-Version 4.2 von 2007 als Standard-Compiler. *Visual C++* von Microsoft unterstützt sogar in der aktuellen Version nur die Version 2.0 [!] des Standards.[8]

Darüber hinaus fiel auf, dass die TBB zahlreiche hochentwickelte threadsichere Datenstrukturen anbietet (insbesondere von der `tbb::concurrent_hash_map` wurde in der eingereichten Lösung des Intel-Wettbewerbs intensiver Gebrauch gemacht). OpenMP hingegen bietet selbst keine entsprechenden Datenstrukturen an. Da die TBB jedoch explizit

[5] Vgl. hierzu **?**, : Laut Handbuch stellt sich ein Parallelisierungsnutzen erst ab mehreren tausend Instruktionen pro Schleifendurchlauf ein.
[6] Vgl. **?**, und **?**, .
[7] Vgl. **?**, .
[8] Vgl. **?**, und **?**, .

einen Parallelbetrieb mit OpenMP gestattet,[9] konnte das Problem gelöst werden, indem auch in der OpenMP-Version des Programms die threadsicheren Datenstrukturen der TBB weiterverwendet wurden.

2.3. Ergebnisse

Um die Leistungsfähigkeit der TBB- und OpenMP-Programme zu testen, wurde eine Eingabedatei mit 5 Millionen Flügen verwendet. Diese Eingabedatei wurde nicht von den Veranstaltern des Wettbewerbs zur Verfügung gestellt, sondern wurde im offiziellen Forum des Wettbewerbs von einem Teilnehmer angeboten. Die Verwendung einer Eingabedatei dieser Größe war jedoch notwendig, um überhaupt Unterschiede im Laufzeitverhalten beider Lösungen messen zu können.

Gemessen wurden Laufzeitverhalten und CPU-Auslastung des Programms in Abhängigkeit der zur Verfügung stehenden Threads (beginnend bei einem Thread bis hin zu 24 Threads). Das Laufzeitverhalten ist in Abbildung 2 auf Seite 7 grafisch aufgetragen (jeder Datenpunkt ist das arithmetische Mittel aus je zehn Einzelmessungen). Die Laufzeit des TBB-Programms konvergiert hier gegen 21,33 Sekunden, die des OMP-Programms gegen 19,56 Sekunden.

Abbildung 3 auf Seite 7 zeigt außerdem die Beschleunigung oder engl. *Speed-Up* (nach **?**, (**?**,) definiert als $S_p(n) = \frac{T^*(n)}{T_p(n)}$).[10] Der Speed-Up des TBB-Programms konvergiert gegen einen Faktor von 3,91 und der des OMP-Programms gegen 4,29.

Die Effizienz beider Programme (definiert als $E_p(n) = \frac{S_p(n)}{p} = \frac{T^*(n)}{p \cdot T_p(n)}$)[11] ist zudem in Abbildung 4 auf Seite 8 gezeigt.

In der Auswertung fällt auf, dass die Laufzeit (und der davon abhängige Speed-Up) zügig konvergieren. Einerseits ist dies zweifellos darauf zurück zu führen, dass das Testsystem über 12 physikalische Rechenkerne verfügt; bei mehr als 12 Threads kann daher nur wenig zusätzlicher Nutzen durch weitere Threads erwartet werden. Andererseits fällt auf, dass die gemessene Laufzeit auch bei weniger als 12 Threads deutlich unter der idealen Laufzeit (hier angenommen als $T_{I,p}(n) = \frac{T^*(n)}{p}$) liegt.

Der geringere Speed-Up ist vermutlich auf sequentielle Anteile des Programms sowie erhöhten Synchronisationsbedarf durch Zugriff auf gemeinsame Datenstrukturen zurück zu führen. Diese Einflussfaktoren begrenzen nach dem *Amdahlschen Gesetz* den maximal erreichbaren Speed-Up.[12]

[9] Vgl. **?**, .
[10] Vgl. **?**, .
[11] Vgl. **?**, .
[12] Vgl. **?**,

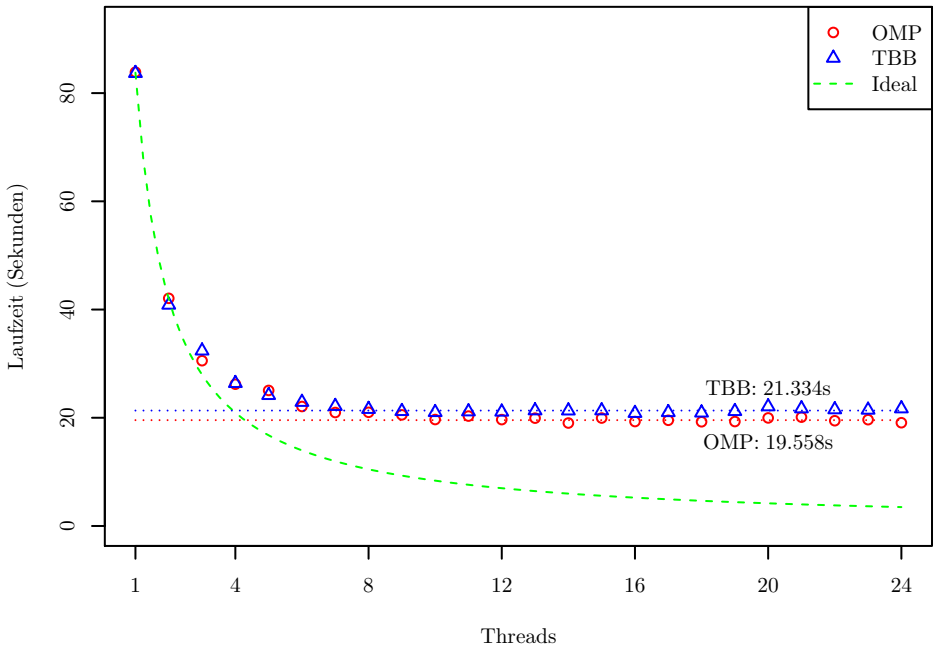

Abbildung 2: Laufzeitverhalten, bei 5 Mio. Flügen und 1-24 Threads.

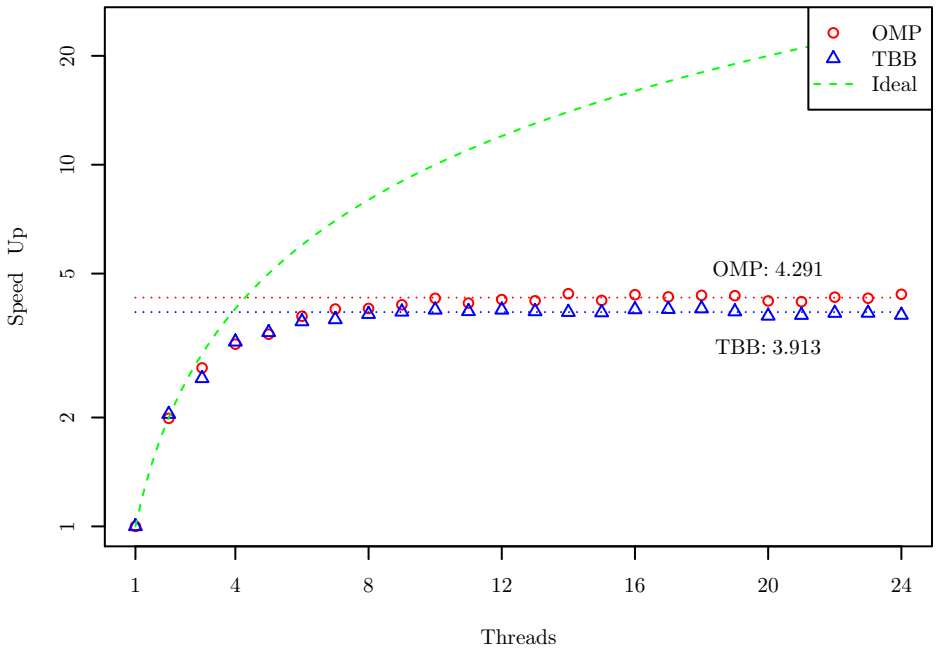

Abbildung 3: Parallelisierungsgewinn, bei 5 Mio. Flügen und 1-24 Threads.

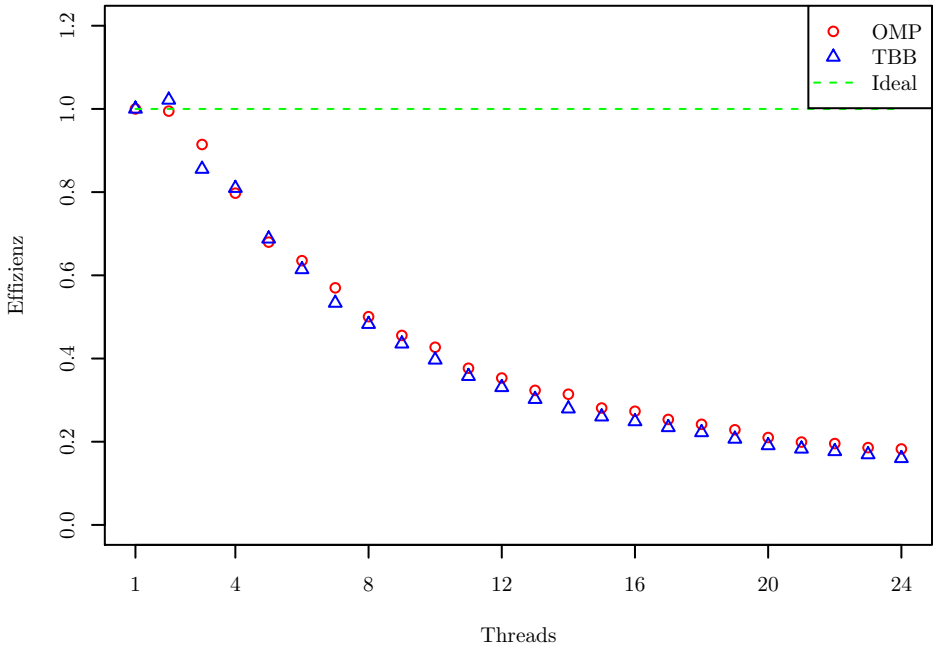

Abbildung 4: Effizienz, bei 5 Mio. Flügen und 1-24 Threads.

3. Fazit

OpenMP stellt eine interessante Alternative zur bereits aus der Veranstaltung bekannten TBB-Bibliothek dar. Die Compiler-Direktiven von OpenMP sind nach eigener Auffassung einfacher handhabbar als die auf *Function Objects* basierende TBB. Dies macht sich vor allem bei der Parallelisierung bereits bestehender, sequentieller Programme bemerkbar.

Negativ fällt im Vergleich zur TBB der etwas geringere Sprachumfang auf: So kennt OpenMP keine parallelen While-Schleifen und die Task-Parallelität, die insbesondere für die Parallelisierung irregulärer Probleme nötig ist,[13] wurde erst spät in den Standard aufgenommen und wird noch nicht von allen Compilern unterstützt.

Einen Vorteil OpenMPs gegenüber der TBB stellt der – zwar geringe, aber dennoch messbare – Geschwindigkeitsvorteil dar. In den durchgeführten Messungen konnte mit OpenMP bei hoher Parallelität ein um $\frac{4{,}291}{3{,}913} - 1 = 9{,}66\%$ höherer Speed-Up erreicht werden.

Zusammenfassend eignet sich OpenMP ausgezeichnet zur Parallelisierung bereits bestehenden Codes. Die Compiler-Direktiven sind vergleichweise einfach handhabbar und erhalten die Lesbarkeit des ursprünglichen Quelltextes. Im Unteschied zur TBB beschränkt sich OpenMP zudem nicht auf C++, sondern ermöglicht auch die Parallelisierung von C- oder Fortran-Code.

[13] Vgl. **?**,

Da OpenMP jedoch keine geeigneten threadsicheren Datenstrukturen mit sich bringt, empfiehlt sich die Verwendung von OpenMP zusammen mit einer entsprechenden Bibliothek (in C++ beispielsweise die TBB- oder die Boost-Bibliothek).

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

A. Aufgabenstellung des Intel Accelerate Your Code-Programmierwettbewerbs

Es folgt ein Auszug aus: `http://www.intel-software-academic-program.com/contests/ayc/2012-11/problem/Intel_AccelerateYourCode_2012-11_problem.pdf`

> Let's imagine you have to go to a software conference in San Francisco (lucky you !), you live in Berlin. Booking the cheapest flight is easy when you want to go from point A to point B and you know the dates. Several websites do that already.
>
> But if you want to pay a little more to extend your flight and do some tourism, it's getting complicated : you are interested in a lot of places (airports in Brazil, Mexico, Colombia, ...), and you are pretty flexible about the dates (before of after your conference). You just want to find a good opportunity : If I can get Berlin-SanFrancisco-SaoPaulo-Berlin for only 100 euros more than just Berlin-SanFrancisco-Berlin, I'll spend a week there on vacation!
>
> But most flights requires a layover (a stop), you have to deal with multiple airline companies, and the price of a single flight is varying depending on the company used for the other flights. If you take all the segments of a flights from a single company, it's 30% cheaper. Plus, companies are forming "alliances" to offer cheaper flights to the customers booking only inside the alliance. If you take all the segments of a flight from a single alliance, it's 20% cheaper.
>
> Even today, this task requires human knowledge and skills because reservation software can't handle the huge combinations. It could be automated. Customers would have more options, spend less time searching on websites, airlines would fill empty planes.
>
> For this contest, you are given a first input file listing all the flights, and a second input file listing the alliances of companies (a company can be in several alliances).
>
> A source file solving the problem, input files and expected outputs are given to help you start:
> `http://www.intel-software-academic-program.com/contests/ayc/2012-11/problem/`
>
> The goal is to optimize and parallelize the software. You can start from scratch if you want, but it's probably safer to optimize and parallelize this reference version.
>
> The goal is to optimize and parallelize for a wide range of use cases (few/many flights, few/ many alliances, short/wide date ranges, ...).

B. Messdaten

Für die Untersuchung des Laufzeitverhaltens wurde jeweils die tatsächliche Laufzeit (*Wall Time*) und die aufgewandte CPU-Zeit gemessen. Die Anzahl der zur Verfügung stehenden Threads wurde von 1 auf 24 erhöht. Um Messfehler und Ausreißer zu vermeiden, wurde jeweils das arithmetische Mittel aus 10 Einzelmessungen gebildet.

Die gemessenen Rohdaten und die R-Skripte zur Auswertung liegen dieser Arbeit auf Datenträger bei.