Dissertation

Multithread Plattformen für Divide und Conquer und inkrementelle Algorithmen und Anwendungen in der Computational Geometry

Daniel Schmitt *

17. Januar 2013

Universität Trier, Deutschland
Fachbereich IV, Informatik

Prof. Dr. Stefan Näher
Prof. Dr. Peter Sturm

*dschmitt@informatik.uni-trier.de

Zusammenfassung

In dieser Arbeit präsentieren wir einen systematischen Ansatz zur Multithread Implementierung von Divide und Conquer sowie inkrementellen Algorithmen. Wir stellen für beide Kategorien von Algorithmen ein Framework vor. Beide Frameworks sollen die Behandlung von Threads erleichtern, die Implementierung von parallelen Algorithmen beschleunigen und die Rechenzeit verringern. Mit Hilfe der Frameworks parallelisieren wir beispielhaft zahlreiche Algorithmen insbesondere aus dem Bereich Computational Geometry, unter anderem: Sortieralgorithmen, konvexe Hülle Algorithmen und Triangulierungen. Der Programmcode zu diese Arbeit ist in `C++` unter Verwendung templatisierter Klassen implementiert und basiert auf der LEDA Bibliothek.

Inhaltsverzeichnis

1. Einleitung

In der Hardwareentwicklung werden entscheidende Leistungszuwächse nicht mehr länger durch die Erhöhung der CPU Taktrate erzielt. Stattdessen kommen in modernen Rechnern Mehrkern-CPUs zum Einsatz die mit einem gemeinsamen Speicher und eigenem Cache ausgestattet sind. Um von dieser Entwicklung zu profitieren, muss die Softwareentwicklung Methoden zur Parallelisierung entwickeln und verfeinern.

Wir betreiben die Parallelisierung mit dem Ziel alle vorhandenen Prozessorkerne eines Rechners voll auszulasten und die größtmögliche Beschleunigung von Algorithmen auf einem Rechner zu erzielen. Eine verteilte Berechnung über Systemgrenzen hinweg, innerhalb von Netzwerken, streben wir nicht an. Aus dieser Voraussetzung und aus Effizienzvorteilen für Threads befasst sich diese Arbeit ausschließlich mit Multithreading an Stelle von Multiprocessing.

Unser algorithmisches Interesse gilt dabei insbesondere der Computational Geometry. Um die Wirksamkeit unserer Parallelisierungslösungen zu demonstrieren, werden wir in dieser Arbeit unter anderem verschiedene Sortieralgorithmen, konvexe Hülle Algorithmen und Triangulierungsalgorithmen diskutieren.

In einem systematischen Ansatz und teilen die Algorithmen in zwei Kategorien ein: Divide und Conquer Algorithmen und Inkrementelle Algorithmen. Erstes Ziel der Arbeit ist die Entwicklung effizienter Frameworks zur einfachen Parallelisierung von Algorithmen beider Kategorien. Die Frameworks haben im einzelnen folgende Designziele:

- Sie sollen die Threads effizient verwalten,
- die Anzahl der Threads gut skalieren,
- die Handhabung von Threads benutzerfreundlich machen,
- die Implementierung paralleler Algorithmen erleichtern und
- die Wiederverwendbarkeit von sequentiellem und parallelem Code ermöglichen.

Neben den Designaspekten von parallelem Code, ist der harte Leistungsgewinn durch Parallelisierung - im Besonderen von geometrischen Algorithmen - das zweite Ziel dieser Arbeit. Die Leistung einer Implementierung ist der Quotient aus Arbeit (der Problemlösung durch den Algorithmus) und Rechenzeit. Die Beschleunigung eines Algorithmus durch parallele Ausführung bzw. allgemein das Einsparen von Rechenzeit ist demnach eine Leistungssteigerung. Den Gewinn durch die Parallelisierung drücken wir als Beschleunigungsfaktor zwischen der sequentiellen und der parallelen Laufzeit eines Algorithmus aus. Die Beschleunigungsfaktorn werden wir in Laufzeitexperimenten ermitteln.

Zu allen Algorithmen, die wir in dieser Arbeit vorstellen und mit Hilfe unserer Frameworks parallelisieren, stellen wir unsere experimentellen Ergebnisse vor. In den Experimenten werden wir die Leistungsfähigkeit unserer Frameworks belegen und die Vor- und Nachteile verschiedener Implentierungen und Techniken sichtbar machen.

Mit dieser Arbeit betreiben wir Algorithm Engineering. Algorithm Engineering wird in der Arbeit von P. Sanders [12] als zyklischer Prozess aus Algorithmenentwurf, Algorithmenanalyse, Implementierung und experimenteller Auswertung beschrieben. Diese Arbeit ist das Ergebnis mehrerer Durchläufe dieses Prozesses. Ein großer Teil unseres Aufwandes ist dabei in Implementierung und experimentelle Auswertung geflossen. Das Debugging, Profiling und Tunning paralleler Algorithmen ist mit einem höheren Schwierigkeitsgrad als gewohnt verbunden. Die speziellen Herausforderungen des Algorithm Engineering für parallele Algorithmen werden von Bader et al. [11] beschrieben. Ziel des Algorithm Engineering sind Implementierung von Algorithmen mit hoher Effizienz und Zuverlässigkeit und leichter Bedienbarkeit für praktische Anwendungen.

Entsprechend der Zielsetzung benötigen wir zuverlässige Werkzeuge, die den Anforderungen des Algorithm Engineering genügen. Die LEDA Bibliothek (Library of Efficient Data types and Algorithms) [6] wurde unter den Aspekten des Algorithm Engineering entwickelt und bildet das Rückgrat dieser Arbeit. Die LEDA Bibliothek enthält die geometrischen Objekte und Prädikate, derer wir uns bedienen, und eine Reihe sequentieller geometrischer Algorithmen, auf die wir zurückgreifen. Da wir über den LEDA Quellcode verfügen sind wir in der Lage Debugging und Profilling auch hierfür durchzuführen, was für andere Bibliotheken nicht der Fall ist.

Wie auch LEDA, ist diese Arbeit in C++ verfasst und verwendet templatisierte Klassen. Die Programmiersprache C++ ist nicht nur wegen der Kompatibilität zur LEDA Bibliothek, sondern vor allen Dingen wegen ihrer Effizienz unsere Präferenz. Entwickelt wurde die Arbeit auf einem Linux Ubuntu System. Wir verwenden daher den GNU gcc Compiler [1] und die Threadbibliothek NPTL (Native POSIX Thread Library) für Linux [3]. Letztere steuern wir über die `pthread` API an [16]. Weitere Details zum verwendeten System sind bei der Präsentation der Laufzeitexperimente zu finden.

Mit der Implementierung paralleler Algorithmen geht die Entwicklung synchronisierter Datenstrukturen einher. Wir beginnen mit der Diskussion technischer Grundlagen und den Synchronisierungsmethoden gegenseitiger Ausschluss und lockfreie Synchronisation, gehen auf die jeweiligen Vor- und Nachteile ein und präsentieren Lösungen, Kapitel 2. Danach besprechen wir ausgiebig das Framework für Divide und Conquer Algorithmen inklusive zahlreicher Implementierungen und Codebeispiele, Kapitel 3. Ebenso verfahren wir mit dem Framework für inkrementelle Algorithmen, Kapitel 4. Unsere Implementierungen haben wir in zahlreichen Experimenten auf Ihren Leistungsgewinn hin untersucht, Kapitel 5. Im Anhang A ist ein Auszug aus der Dokumentation des zu dieser Arbeit gehörenden Softwarepakets zu finden.

1.1. Verwandte Arbeiten/Themen

Wir verwenden in dieser Arbeit Multithreading zur Parallelisierung. Die Alternative ist die Benutzung von Multitprocessing. Die MPI API (Message Passing Interface) definiert einen umfangreichen Satz von Funktionen zur Synchronisation und Kommunikation zwischen Prozessen. Informationen zum MPI Standard sind auf der Homepage [13] oder der Webseite des MPI Forums [14] zu finden.

Weiter haben wir uns bei der Verwendung von Threads konkret auf die pthread API [16] festgelegt. Eine weitere Programmierschnittstelle für Threads ist OpenMP [15]. OpenMP definiert Compiler Direktiven. Mit diesen Direktiven werden Codebereich zur parallelen Ausführung markiert. So kann man beliebige Funktionen durch einige einfache Anweisungen parallel starten. Das Standardbeispiel ist die parallele Abarbeitung von For-Schleifen, bei der die einzelnen Eingabeelemente unabhängig voneinander sind.

Das Programmierschnittstelle von OpenMP ist einfach zu bedienen. Jedoch geht mit der Benutzung von OpenMP ein Kontrollverlust einher. Das Starten und Beenden von Threads ist vollständig maskiert und lässt sich nur über Umgebungsvariablen steuern. Das Design von OpenMP zielt darauf ab, unabhängige Codeteile parallel auszuführen. Anstelle von Speicherzugriffen synchronisiert OpenMP die Ausführung von Codeabschnitten. Datenstrukturen mit einer hohen Zahl konkurrierender Zugriffe sind so schwierig effizient zu implementieren. Die pthread API ist hingegen systemnah. Sie erlaubt exakte Formulierungen von Threadstarts und -stops, sowie die Benutzung, Erzeugung und Manipulation von Mutex Variablen. Dadurch ist mit pthread potenziell eine effizientere Programmierung möglich. Tests haben gezeigt, dass Pthread und OpenMP nicht kompatibel sind.

Das Projekt MCSTL (Multi-Core Standard Template Library) hat sich der Parallelisierung der C++ Standard Template Library verschrieben. Die MCSTL Bibliothek wird vorgestellt in der Arbeit von Peter Sanders et al. [9]. Weitere Informationen sind auf der Webseite des Projekts zu finden [10]. Die MCSTL Bibliothek enthält eine Reihe von Basisalgorithmen, wie Sortier- und Partitionierungsalgorithmen.

Die Bibliothek Thread Building Blocks (TBB) ist eine templatisierte C++ Bibliothek von Intel. Sie bietet nicht nur Klassen und Funktionen zur Organisation der Threads, zur Synchronisation und threadsichere Datenstrukturen, sondern auch weiterführende Konzept zur Parallelisierung von Algorithmen. Einige der Funktionalitäten der TBB sind:

Tasks/Task Scheduling: Parallel abzuarbeitende Aufgaben (Tasks) können von Task Schedulern verwaltet werden. Der Scheduler verteilt die Aufgaben an Threads und implementiert Work Stealing Algorithmen zur Ausbalancierung der Threads. Aufgaben können rekursiv angelegt und in Abhängigkeit voneinander abgearbeitet werden.

Synchronisation: Es werden sowohl blockierende als auch lockfreie Synchronisationsmethoden unterstützt. Die vorhandenen Mutex Variablen implementieren unter anderem rekursives Sperren und einen Fairnessalgorithmus für konkurrierende Threads.

Datenstrukturen: Von der Bibliothek werden verschiedene threadsichere Datenstrukturen angeboten, beispielsweise eine Queue, ein Vector, eine ungeordnete Menge und weitere.

Templatisierte Algorithmen: Es stehen verschiedene Templates zur Ausnutzung von Datenparallelität zu Verfügung, unter anderem ein Template, das parallel über eine Eingabe iteriert (parallel_for), ein Template, das mit gegebener Funktion parallel einen Wert über eine Eingabemenge berechnet (parallel_reduce) und das Range Konzept, das zur parallelen Partitionierung von Mengen dient.

Flow Graph: Ein Flow Graph modelliert die Programmlogik als Graph. Die Knoten repräsentieren Funktionen, Kanten repräsentieren einen Kanal über den Daten weitergeleitet werden. Der Flow Graph kann zyklische wie azyklische Graphen abbilden und ist eine frei programmierbare Schnittstelle für Parallelisierung durch Pipelining.

Die Thread Building Blocks Bibliothek implementiert viele Automatismen zur Aufgaben- und Datenparallelität und kapselt die Steuerung der Threads vollständig. Weitere Informationen zur Bibliothek können auf der Homepage [21] und in der Dokumentation von Intel [22] gefunden werden.

Die Boost C++ Bibliothek ist ein Sammlung von über 100 portierbaren Teilbibliotheken. Die Anwendungsgebiete von Boost sind vielfältig und durch die Verwendung von Templates ist Boost flexibel einsetzbar. Die Teilbibliotheken Threadßtellt Klassen und Funktionen zur Multithreadprogrammierung zur Verfügung. Die Programmierschnittstelle für Threads ist komfortabel, bietet aber nicht mehr als Basisfunktionen zum Starten und Beenden von Threads und zur Synchronisation. Eine Dokumentation der Bibliothek ist auf der Homepage des Boost Projekts [18] zu finden, detailliertere Ausführungen sind in folgendem Online Buch zu finden [19].

Natürlich stehen auch in anderen Umgebungen Bibliotheken zur Parallelisierung zu Verfügung.

Mit der Veröffentlichung der Version 4 hat Microsoft sein .Net Framework um die Task Parallel Library (TPL) ergänzt. Die TPL Bibliothek bietet verschiedene Konstrukte zur vereinfachten Implementierung von Daten- und Aufgabenparallelität. Zur automatischen Reallisierung der Datenparallelität stehen verschiedene Schleifenkonstrukte zur Verfügung. Die Aufgabenparallelität wird mit Hilfe von Klassen (bspw. ThreadPool oder Taskfactory) implementiert, die Threads und ihre Aufgaben (Tasks) verwalten und organisieren. Unter anderem implementieren sie einen Work Stealing Algorithmus. Atomare Operationen stellt .Net mit der Klasse Ïnterlockedßur Verfügung. Diese wird auch verwendet, um einige der threadsichere Datenstrukturen wie z.B. Queue oder Stack threadsicher zu machen. Andere Datenstrukturen sind blockierend. Die .Net Datenstrukturen enthalten eine Unterstützung zur parallelen Partitionierung ihrer Inhalte. Weitere Informationen sind auf der Homepage des Microsoft Developer Networks [23] zu finden.

In der Java Entwicklungsumgebung der Version 7 gibt es zahlreiche Klassen zur Unterstüzung der Multithread Porgrammierung. Die Java Klassen liefern Basisfunktionalität zur Erzeugung und Organisation von Threads und zur Synchronisation. Die Funktionalität der Mutex Variablen wurde erweitert und verfeinert; so gibt es Mutex Klassen, die

rekursives Sperren und eine Fairnessfunktionalität für wartende Threads implementieren. Einfache Typen wie Integer und Long stehen als Klassen mit atomaren Funktionen zur Verfügung. Weiter sind threadsichere Datenstrukturen wie Stacks und Queues sowohl lockfrei als auch blockierend als Klassen verfügbar. Zu guter Letzt unterstützt Java auch in einfacher Weise die parallele Ausführung rekursiver Funktionen. Weitere Details zur Java API sind in der Online Dokumentation [20] nachzulesen.

2. Multi Thread Programmierung

2.1. Einleitung

Ein Thread ist ein Ausführungsstrang eines Prozesses. Zur Parallelisierung steht uns die Möglichkeit zur Verfügung ein Programm in mehrere Prozesse aufzuteilen und ausführen zu lassen (Multi Processing) oder innerhalb eines Prozesses mehrere Threads zu starten (Multi Threading). Von einem abstrakten Standpunkt aus, gleichen sich beide Verfahren. Die technischen Kontexte sind aber verschieden.

Parallele Prozesse laufen auf unterschiedlichen CPUs oder Kernen mit einem eigenen Speicherbereich. Der Vorteil von Multi Processing ist die Fähigkeit Prozesse auch über mehrere Systeme hinweg parallel zu betreiben. Dafür ist die Kommunikation zwischen Prozessen aufwendiger und teurer und mit weiteren Problemstellungen verbunden. Auch die Verwaltung von parallelen Prozessen auf einem System ist leistungs- und speicherintensiver im Vergleich zu Threads. Threads hingegen können schnell von Prozessen erzeugt werden. Die Threads eines Prozess teilen sich dessen Speicher. Das macht die Synchronisation zwischen Threads einfacher und den Speicherbedarf geringer. Außerdem unterstützen moderne Prozessoren den schnellen Wechsel zwischen Threads (Context Switch) eines Prozesses.

Innerhalb eines Systems ist die Parallelisierung mit Thread also leistungsfähiger und einfacher zu handhaben. Ausdrückliches Ziel dieser Arbeit ist das Ausschöpfen aller Ressourcen eines modernen Systems mit Multicore Prozessor(en). Das Auslasten von Computerfarmen gehört nicht dazu. Die Wahl der Parallelisierungstechnik fällt daher auf Multithreading. Wie bereits erwähnt implementieren wir die Algorithmen in `C++` unter Verwendung der `pthread` API [16], die wir im nächsten Abschnitt 2.2 einführen. Da unsere Frameworks die pthread Aufrufe vollständig maskieren, fällt unsere Diskussion dazu sehr kurz aus.

Parallelität bringt die Notwendigkeit der Synchronisierung mit sich. Wir behandeln zwei Synchronisationsmethoden: den gegenseitigen Ausschluss und die lockfreie Synchronisation. In Abschnitt 2.3 besprechen wir die Synchronisation durch gegenseitigen Ausschluss. Wir werden damit zusammenhängende Probleme, Einschränkungen und Lösungsstrategien diskutieren. Im Anschluss daran diskutieren wir die lockfreie Synchronisation und deren Vor- und Nachteile in Abschnitt 2.4.

Die Parallelisierung von Algorithmen geht mit der Entwicklung von threadsicheren Datenstrukturen einher. Anhand der vorgestellten Synchronisationsmethoden werden wir in Abschnitt 2.5 einige einfache threadsicher Datenstrukturen implementieren.

Außerdem sind wir bei der Implementierung verschiedener Algorithmen immer wiederkehrenden Basisfunktionen begegnet. Einige davon haben wir als Klassen parallelisiert, siehe Abschnitt 2.6.

2.2. Die `pthread` API

Der Name pthread ist die Kurzform für POSIX Threads. Die pthread API dient der Erzeugung, Verwaltung und Synchronisation von Threads. Sie ist im POSIX Standard IEEE Std 1003.1c-1995 definiert [17].

Die Implementierung der API ist für viele Unix ähnliche Systeme verfügbar. Die native Posix Thread Library [3] hat sich als Implementierung auf Linux Systemen durchgesetzt.

Die pthread API definiert über 100 Funktionen zur Programmierung und Synchronisierung von Threads. Die beiden wichtigsten Funktionen sind:

`pthread_create()` erzeugt einen Thread. Die Funktion erhält folgende Argumente:

- Ein Thread Handle, d.h. die Identität des Threads in einer pthread Variable.
- Die Optionen des Thread, zusammengefasst in einer pthread-Struktur.
- Einen Funktionspointer auf die Startfunktion des Threads.
- Einen Pointer auf das Argument der Startfunktion.

`pthread_join()` bereinigt einen terminierten Thread vom System. Die Argumente sind:

- Das Handle des Threads der bereinigt wird.
- Der Status des terminierten Threads.

Um einen Thread zu starten benötigt man also einen Startfunktion mit einer gegebenen Signatur und die Funktionsargumente in einer gekapselten Datenstruktur. Die Struktur eine Programms sieht im einfachen Fall so aus, dass ein Thread, genannt Master, eine beliebige Zahl weiterer Threads startet. Sowohl diese Threads, als auch der Master Thread führen parallele Berechnungen durch. Im Anschluss müssen die Threads vom Master Thread wieder eingesammelt werden. Der Master Thread führt die `join` Operation nacheinander auf alle Threads aus. Gegebenenfalls wartet der Master Thread auf deren Terminierung. Die Operation `pthread_join()` ist damit auch die einfachste Form einer Synchronisation. Es gibt alternative Vorgehensweisen zum Starten und Beenden von Threads, welche in dieser Arbeit nicht benötigt werden.

Über das Starten und Stoppen von Threads hinaus kommen in unserem Programmcode nur sehr wenige API Funktionen zur Anwendung. Wenn weitere API Aufrufe verwendet werden, werden wir diese einführen. Grundsätzlich gilt, dass alle API Aufrufe in unserem Framework gekapselt sind. Wir wollen dadurch einen höheren Komfort erzielen. Wegen ihrer C Kompatibilität verwendet die pthread API kein Klassenkonzept und stattdessen Funktionspointer. Eines unserer Desingziele ist eine benutzerfreundliche Handhabung von Threads.

Die pthread API unterstützt den gegenseitigen Ausschluss, dazu mehr im nächsten Abschnitt. Eine vollständige Dokumentation der API ist im Internet beispielsweise unter [16] zu finden.

2.3. Gegenseitiger Ausschluss

Die Threads eines Prozesses teilen sich einen gemeinsamen Speicherbereich. Im Speicherbereich enthalten sind sowohl lokale Variablen der Threads, als auch globale Variablen, die mehreren oder allen Threads bekannt sind. Die Kommunikation zwischen den Threads erfolgt durch Lese- und Schreiboperationen auf gemeinsame Variablen. Wollen mehrere Threads gemeinsame Variablen zeitgleich verändern, befinden sich diese in einer Wettlaufsituation (engl. Race Condition). Standardbeispiel für eine Wettlaufsituation ist das parallele Erhöhen einer gemeinsamen Variable. Die Inkrementierungsfunktion zerfällt in drei Teile: Auslesen der Variable aus dem Speicher, Erhöhen der Variable und Rückschreiben in den Speicher. Interferieren die einzelnen Teile bei einer parallelen Ausführung wird das Ergebnis verfälscht. Die Inkrementierung muss synchronisiert werden. Die übliche Synchronisationsmethode ist der gegenseitige Ausschluss.

Der gegenseitige Ausschluss garantiert, dass zu einem Zeitpunkt nur ein Thread die geschützte Variable lesen oder schreiben darf. Implementiert wird dieses von Mutex Variablen (Mutex von engl. Mutual Exclusion). Mutex Variablen werden von Threads gesperrt (auch besetzt, von engl. to lock) und entsperrt (auch freigegeben, von engl. to unlock). Ist ein Mutex gesperrt, ist er im Besitz des Threads. Ein Mutex kann zu jedem Zeitpunkt nur von einem Mutex gesperrt werden. Jeder weiterer Thread, der versucht einen gesperrten Mutex zu besetzen, wird je nach verwendetem Funktionsaufruf entweder blockiert bis der Mutex frei ist oder erhält eine negative Rückmeldung. Blockierte Threads werden in einen Schlafzustand versetzt bis der Mutex freigegeben ist. Schlafende Threads benötigen nur sehr wenig Rechenzeit. Zudem kann der freie Prozessorkern andere Aufgaben übernehmen. Ein blockierter Thread ist vom Besitzer des Mutex abhängig. Falls mehrere Threads gleichzeitig um einen Mutex konkurrieren, d.h. im gleichen Moment sperren möchten oder bereits blockiert sind, entscheidet der Zufall über den neuen Besitzer. Im weiteren werden wir die Konkurrenz um die gleiche Ressource auch als Kollision bezeichnen.

Um die Korrektheit eines parallelisierten Algorithmus leichter zu beherrschen, sollte der Mutex nur von dem Thread entsperrt werden, der ihn besetzt hält, was technisch leider nicht zwingend ist. Weiter darf jeder Thread mehrere Mutex Variablen gleichzeitig sperren. Aus beiden Bedingungen folgt, dass zyklische Abhängigkeiten zwischen den Threads entstehen können: Nehmen wir beispielsweise an, dass Thread A Mutex 1 besitzt und Mutex 2 sperren möchte. Thread B besitzt Mutex 2 und wartet auf Mutex 1. Damit befinden sich Thread A und B in einer zyklischen Abhängigkeit und blockieren sich gegenseitig. Diese Situation nennt man Deadlock, siehe [24].

Die Vermeidung (engl. Avoidance) von Deadlocks ist ein zentraler Punkt in der Multithread Programmierung, der nicht mit der Deadlock Erkennung (engl. Detection) zu verwechseln ist. Ziel der Vermeidung ist es Deadlocks unter gegebenen Bedingungen vorherzusehen und in der Entstehung zu unterbinden. Deadlock Erkennung zielt auf die Erkennung und Auflösung von bestehenden Deadlocks. Solange alle Konkurrenten um gemeinsame Ressourcen in ihrem Verhalten bekannt sind, (da sie selbst implementiert werden,) ist es möglich Methoden zur Deadlock Vermeidung anzuwenden. Dies werden wir in dieser Arbeit tun. Die Deadlock Erkennung findet beispielsweise bei der Imple-

mentierung von Datenbank Anwendung, wo sich das Verhalten der Datenbank Clients nicht absehen lässt.

Der gegenseitigen Ausschlusses ist nicht kostenfrei, er verursacht Leistungsverluste. Leistung geht sowohl durch Mutex Operationen, als auch durch blockierte Threads verloren. Auf der einen Seite führt also eine unnötig hohe Zahl an Mutex Variablen durch eine hohe Zahl von Mutex Operationen zu Einbußen. Auf der anderen Seite können zu viele und zu lange Blockaden durch global verwendete Mutex Variablen leicht eine hohe Verschwendung von Rechenzeit verursachen. Im Allgemeinen ist es das Ziel sowohl die Zahl der Mutex Variablen, als auch die Häufigkeit von blockierten Threads zu minimieren. Eine angemessen Mutex Granularität für einen gegeben Algorithmus zu finden ist eine Designherausforderung.

Ein weiteres bekanntes Probleme des gegenseitigen Ausschlusses ist die Prioritätsumkehr (engl. Priority Inversion). Die Prioritätsumkehr tritt besonders unter zeitkritischen Echtzeitsystemen auf, wo unterschiedliche Aufgaben mit unterschiedlicher Wichtigkeit parallel abgearbeitet werden müssen. Angenommen Threads sind Prioritäten zugeordnet, so dass Threads mit geringer Priorität weniger Rechenzeit vom System zur Verfügung gestellt wird als Threads mit hoher Priorität. Ein als gering eingestufter Thread A reserviert sich eine gemeinsame Ressource durch gegenseitigen Ausschluss. Ein höher priorisierter Thread B möchte diese Ressource auch reservieren, wird aber durch A blockiert. Da A wenig Rechenzeit erhält, wird die Ausführungsdauer von B, auf das Niveau von A gedrückt. Ein prominenter Fall einer Systemblockade durch Prioritätsumkehr ist der Mars Rover [25]. Da wir in dieser Arbeit Threads verwenden, um die Hardware mit einer einzelnen Aufgabe voll auszulasten, spielen Prioritäten in diesem Sinn für uns keine Rolle. Alle Threads führen gleichwertige Arbeiten aus und haben dieselbe Priorität.

2.3.1. Implementierung einer Mutex Klasse

Der gegenseitige Ausschluss wird von der pthread API durch die Bereitstellung von Mutex Variablen unterstützt. Die wichtigsten Operationen, die Mutex Variablen betreffen, sind folgende:

`pthread_mutex_init(...)` initialisiert einen gegebenen Mutex.

`pthread_mutex_destroy(...)` deinitialisiert einen gegebenen Mutex.

`pthread_mutex_lock(...)` sperrt einen gegebenen Mutex.

`pthread_mutex_trylock(...)` versucht einen gegebenen Mutex zu sperren.

`pthread_mutex_unlock(...)` entsperrt einen gegebenen Mutex.

Wichtig ist die Unterscheidung der beiden sperrenden Funktionen, verkürzt: `lock` und `trylock`. Die lock Operation ist blockieren, d.h. wenn der Mutex besetzt ist, wird der aufrufende Thread blockiert. Die trylock Operation terminiert unmittelbar mit einer Rückmeldung: entweder konnte der Mutex gesperrt werden oder der Versuch wurde abgebrochen, weil der Mutex schon besetzt ist. Die Funktion `unlock` gibt den Mutex frei.

Wir gehen nicht weiter auf diese Funktionen ein, da wir eine Mutex Klasse entworfen haben, um die API Aufrufe zu kapseln:

Code 2.1 (Mutex Klassendeklaration).

```
class mutex {
  void lock();
  void unlock();
  bool try_lock();
}
```

Die Funktionen `lock()` und `unlock()` sperren bzw. entsperren den Mutex. Die `lock()` Operation ist blockierend, die Funktion `try_lock()` die nicht-blockierende Variante. Ist die try_lock Funktion erfolgreich, terminiert sie mit `true` ansonsten mit `false`. Konstruktor und Destruktor führen jeweils Funktionen zur Initialisierung und Deinitialisierung aus der pthread Bibliothek aus.

2.3.2. Deadlock Vermeidung

In dieser Arbeit behandeln wir die parallele Konstruktion komplexer Datenstrukturen. Um die Anzahl von Kollisionen der Threads zu vermeiden, verwenden wir oft eine hohe Zahl von Mutex Variablen. Eine systematische Behandlung der Deadlock Problematik ist unumgänglich.

Wir werden zur Deadlock Vermeidung verschiedene Strategien diskutieren. Zu jeder der Herangehensweisen werden wir Regeln aufstellen, deren Einhaltung durch den Algorithmus einen deadlockfreien Ablauf garantiert. Die Regeln grenzen insbesondere `lock` Operationen ab, die potenziell einen Deadlock verursachen können. Weiter behandeln wir die Reaktion auf einen erkannten potenziellen Deadlock in Form eines begrenzten Abbruchs der Berechnung.

Ein solcher Abbruch folgt immer dem gleichen Schema. Der Thread bringt erst die Datenstruktur in einen konsistenten Zustand zurück hinsichtlich der ggf. durchgeführten Änderungen. Dann entsperrt er alle Mutex Variablen in seinem Besitz und steigt zuletzt wieder an einer früheren Stelle in den Algorithmus ein.

Aus der Abbruchstrategie folgen verschieden Designeranforderungen. Im Algorithmus müssen an entsprechenden Stellen Wiedereinstiegspunkte für die Threads nach einem Abbruch vorgesehen werden. Außerdem müssen die Threads mindestens so viele Mutex Variablen in Besitz halten wie zum konsistenten Rückbau der vorgenommenen Änderungen an der Datenstruktur nötig sind.

Eine wichtiges Mittel zur Deadlock Vermeidung ist die `trylock()` Methode, da diese nie blockiert. Denn in der praktischen Umsetzung geht es bei der Deadlock Vermeidung auch um die Frage, wann ein Thread den blockierenden Funktionsaufruf `lock` verwenden und ggf. warten darf, und wann der Thread nicht blockieren soll. Wenn ein Thread erst einmal blockiert, bleiben uns keine programmatischen Eingriffsmöglichkeiten mehr, um Bedingungen zur Deadlock Vermeidung zu überprüfen, bzw. auf deren Verletzung zu reagieren.

Die `trylock()` Methode bringt uns schnell zu einem naiven Ansatz. Wir wissen, Deadlocks können entstehen sobald mindestens zwei Threads mehr als einen Mutex gleichzeitig sperren.

Satz 2.1 (Naive Deadlock Vermeidung).
Es entsteht kein Deadlock, solange der Algorithmus folgende Bedingungen erfüllt:

1. *Ein Thread darf die* `lock` *Methode nur benutzen solange er keine Mutex Variablen besitzt.*
2. *Falls ein Thread mindestens einen Mutex besitzt, muss er die* `trylock()` *Methode anwenden. Schlägt diese fehl, folgt ein Abbruch nach oben genanntem Schema.*

Ein Thread kann keinen Deadlock verursachen, solange er noch keinen Mutex besitzt. Der erste Teil ist trivial korrekt. Der zweite Teil des Satzes verhindert jegliche Abhängigkeit zwischen den Threads, da kein blockierender Aufruf verwendet wird. Es können keine Deadlocks entstehen. □

Der Nachteil dieser Methode ist zum einen, dass sie zu pessimistisch ist. Wenn jede nicht zyklische Abhängigkeit zwischen den Threads als Deadlock interpretiert wird, ist die Abbruchrate unnötig hoch. Zum andern können sich zwei Threads während der gleichen Berechnung beliebig oft gegenseitig ineinander verhaken, abbrechen und neustarten. Der Ansatz ist nicht deterministisch.

2.3.3. Mutex Hierarchie

Angenommen über alle vorhandenen Mutex Variablen ist eine hierarchische Ordnung definiert. Die Mutex Variablen lassen sich also auf eine Baumstruktur abbilden. Den Vater eines Mutex im Baum nennen wir Vorgänger, die Kinder Nachfolger.

Satz 2.2 (Deadlock Vermeidung mit Mutex Hierarchie).
Es entsteht kein Deadlock, solange der Algorithmus folgende Bedingungen erfüllt:

1. *Ein Thread darf jeden Mutex sperren, solange er keinen anderen Mutex besitzt.*
2. *Ein Thread, der Mutex Variablen besitzt:*
   a) *darf den Nachfolger eines Mutex, den er besitzt, sperren.*
   b) *darf den Vorgänger eines Mutex, den er besitzt, nur nicht-blockierend sperren (*`trylock()`*). Schlägt diese fehl, folgt ein Abbruch.*

Teil 1 ist korrekt. Teil 2 besagt, dass alle Mutex Variablen, die ein Thread besitzt, im Baum zusammenhängen müssen. Aus Teil 2a folgt, dass zu jedem Mutex m höchstens ein Thread A existiert, der m blockierend sperren darf. Thread A ist der Besitzer des Vorgängers von m. Angenommen Thread A wird von Thread B blockiert, dann darf B nach Bedingung 2 keinen Vorgänger eines Mutex von A besitzen. Durch 2b kann B nicht von A blockiert werden. □

Die Bedingung 2a stellt des weiteren sicher, dass der Fortgang des Algorithmus für mindestens einen Thread gewährleistet ist. Der Ansatz ist deterministisch.

Im Kontext von geometrischen Berechnungen sichern Mutex Variablen meist Datenstrukturen, beziehungsweise deren Teilelemente. Solange es sich dabei um Baumstrukturen handelt, steht auch implizit eine Mutex Hierarchie zur Verfügung.

Die Nachteile dieser Strategie sind offensichtlich. Das Problem der zyklischen Abhängigkeit zwischen den Threads wird in die Elemente einer Datenstruktur verlagert. Viele Algorithmen in der Computational Geometry arbeiten mit dichten, zyklischen Graphen. Die Mutex Hierarchie ist dort nicht anwendbar. In jedem Fall muss die Deadlock Vermeidung mit Mutex Hierarchie auf jeden Algorithmus einzeln abgestimmt werden.

2.3.4. Thread Hierarchie

Angenommen über alle Threads ist eine lineare Ordnung definiert. Hierzu wird den Threads eine eindeutige Id (auch Priorität) zugeordnet. Zu den Ids ist eine Ordnungsrelation definiert. Wir verwenden im Weiteren den Begriff der Priorität ausschließlich um Threadkollisionen aufzulösen, nicht zur Zuweisung von Rechenzeit.

Seien A, B Threads, so dass gilt: $Id(A) > Id(B)$. Wir legen fest, dass Thread A mit höherer Id beim Sperren eines Mutex blockieren darf, wenn B den Mutex besitzt. Thread B darf sich von A nie blockieren lassen und muss abbrechen.

Satz 2.3 (Deadlock Vermeidung mit Thread Hierarchie)**.**
Es entsteht kein Deadlock, solange der Algorithmus folgende Bedingungen erfüllt:

1. *Ein Thread darf jeden Mutex sperren, solange er keinen anderen Mutex besitzt.*
2. *Ein Thread T, der einen Mutex besitzt, darf nur nicht blockierende* `lock` *Methoden verwenden. Wenn diese scheitern, überprüft der Thread T die Id des Besitzers B des Mutex. Ist $Id(B) > Id(T)$, folgt ein Abbruch. Ist umgekehrt $Id(B) < Id(T)$, wiederholt der Thread den Versuch des Sperrens.*

Teil 1 ist korrekt. Aus Teil 2 folgt, dass ein Thread A von einem Thread B nur abhängig sein kann, wenn gilt $Id(A) > Id(B)$. Da die Ids eindeutig sind und eine lineare Ordnung existiert, existieren keine zyklischen Abhängigkeiten. □

Ein Thread T, der auf einen Mutex M wartet, blockiert zwar im Sinne, dass der Algorithmus nicht fortschreitet. Er verfällt dabei aber nicht in einen Schlafzustand, sondern betreibt aktives Warten (engl. busy waiting). Dies ist nötig, da der Besitzer des Mutex M wechseln kann ohne das T involviert ist. Auf die technischen Details und Notwendigkeiten werden wir im nächsten Abschnitt weiter eingehen.

Der Vorteil dieser Strategie ist die Unabhängigkeit von konkreten Algorithmen oder Datenstrukturen. Die Thread Hierarchie kann vollständig von einer Mutex Variablen implementiert werden. Lediglich die Abbruchstrategie muss in die parallelisierten Algorithmen eingefügt werden. Der Nachteil ist das aktive Warten. Dieses kann bei naiver Verwendung einen Verbrauch von Rechenzeit verursachen ohne den Algorithmus voranzubringen.

2.3.5. Implementierung einer Thread Hierarchie

Wir implementieren einen Mutex mit den Regeln der Thread Hierarchie. Wir benötigen eine Zuordnung von eindeutigen Ids zu Threads. Die Funktion `pthread_self()` der pthread Bibliothek liefert eine eindeutige Zahl vom Typ `pthread_t`.

Die Mutex Klasse `da_mutex`, die wir entwerfen, enthält neben den üblichen Funktionen zwei Variablen: Die `lock_id` mit der Id des besitzenden Threads, und eine Mutex Variable `mutex`. Die Variable `mutex` sichert auch den schreibenden Zugriff auf `lock_id`. Wenn die Klasse `da_mutex` nicht gesperrt ist, ist `lock_id` gleich null, ansonsten ungleich null.

Code 2.2 (Mutex mit Thread Hierarchie, Klassendeklaration)**.**

```
class da_mutex
{
public:
  da_mutex() : lock_id(0) {};
  ~da_mutex() {};

  int lock();
  int try_lock();
  void unlock();

private:
  leda_mutex mutex;
  pthread_t lock_id;
}
```

Die `lock()` Funktion versucht den Mutex nicht blockierend zu sperren. Falls dies gelingt erfolgt eine positive Rückmeldung (`LOCKED`). Falls dies nicht gelingt, da der Mutex gesperrt ist, wird die Priorität des Besitzers mit der des wartenden Threads verglichen. Hat der Besitzer eine höhere Priorität erfolgt ein Abbruch (`BLOCKED`), ansonsten wird der Versuch wiederholt. Der wartende Thread betreibt aktives Warten. Außerdem wird die Operation überprüfen, ob der wartende Thread den Mutex schon besitzt. Auch in diesem Fall terminiert die Funktion, der Rückgabewert ist `OWNER`.

Code 2.3 (Mutex mit Thread Hierarchie, `lock()`)**.**

```
  int lock()
  {
    pthread_t my_t = pthread_self();
    if (my_t == lock_id) return OWNER;
    while (!mutex.try_lock())
      { if (my_t < lock_id) return BLOCKED; }

    lock_id = my_t;
```

```
  return LOCKED;
}
```

Durch aktives Warten simulieren wir eine blockierende lock Operation. Wir implementieren aktives Warten mit einer Schleife, deren Abbruchbedingung ein erfolgreiches Sperren des Mutex ist. Der Vergleich der Thread Prioritäten wird im Schleifenkörper durchgeführt. Eine Mutex Variable, die aktives Warten betriebt, heißt `Spinlock` Mutex.

Aktives Warten und wiederholte Prüfung des Besitzers sind in dieser Implementierung der Thread Hierarchie notwendig. Wir nehmen den Fall an, dass die Threads A, B, C mit $Id(A) > Id(B) > Id(C)$ existieren, C den Mutex M besitzt und A und B, um M konkurrieren. Da wir grundsätzlich keine Aussage über die Reihenfolge von erfolgreichen lock Operationen der Threads machen können, wissen wir nicht ob A oder B den Mutex M nach C besitzen werden. Wenn B gewinnt, wird A weiter warten. Wenn A gewinnt, muss jedoch B abbrechen. D.h. während ein Thread auf das entsperren eines Mutex wartet, können sich Besitzverhältnisse und Priorität für diesen Thread umkehren. Daher muss der wartende Thread regelmäßig die Priorität des Besitzers überprüfen.

Zu beachten ist, dass in der gezeigten Implementierung die Variable `mutex` auch wiederum Blockaden verursachen kann, auch wenn er jeweils nur sehr kurz gesperrt wird. Darauf kommen wir in Abschnitt 2.4.1.2 nochmal zu sprechen.

Die `try_lock()` Operation der Mutex Klasse kann einfach von der lock Operation abgeleitet werden. Die `while` Schleife ist durch eine `if` Anweisung zu ersetzen. Auch die `unlock()` Operation ist simpel. Der Mutex wird entsperrt, nachdem lock_id null gesetzt wurde.

Code 2.4 (Mutex mit Thread Hierachie, `unlock()`).

```
void unlock()
{
  lock_id = 0;
  mutex.unlock();
}
```

Der Nachteil von aktivem Warten ist, dass aktiv wartenden Threads Leistung verbrauchen. Der Mutex kann nur effizient eingesetzt werden, wenn die Wartezeiten der Threads gering sind und es selten zu Konflikten kommt. Ist die Wartezeit des Threads zu lange, ist es hingegen besser herkömmliche Mutex Variablen zu verwenden. Das System hat dann die Möglichkeit den freien Prozessorkernen andere Aufgaben zuzuteilen. Hier gilt zu bedenken, dass durch den Wechsel zwischen Threads auch wiederum Leistung verloren geht.

Wir möchten kurz eine alternative Implementierung eines Mutex mit Thread Hierarchie ohne aktives Warten skizzieren. Um auf aktives Warten verzichten zu können, müssen wir eine Reihenfolge für die wartenden, konkurrierenden Threads garantieren können. Dadurch können wir unmittelbar entscheiden ob der Sperrversuch eines Threads erlaubt ist oder dieser abbrechen muss, weil ein höher priorisierter Thread zuvor kommt.

Um eine Sperr-Reihenfolge in einem Mutex zu implementieren, benutzen wir eine Queue, die für jeden wartenden Thread einen Eintrag enthält. Die Threads in der Queue sind in einem Schlafzustand. Sobald der Mutex entsperrt wird, wird der erste Thread in der Queue reaktiviert und kommt in den Besitz des Mutex. Die Priorität des höchstpriorisiertesten Threads in der Queue ist dem Mutex bekannt. Mit dieser Information kann ein hinzukommender Thread sofort entscheiden, ob er warten darf oder abbrechen muss.

Das Versetzen eines Thread in einen Schlafzustand und das gezielte Reaktivieren übernehmen Condition Variablen der `pthread` Bibliothek. Wir asoziieren mit jedem Thread eine Condition Variable vom Typ `pthread_cond_t`. Der Aufruf der pthread Funktion `pthread_cond_wait ( pthread_cond_t c, pthread_mutex_t m)` mit der Condition Variable `c` versetzt den Aufrufer in einen Schlafzustand. Wird der Thread geweckt, versucht er Mutex `m` zu sperren. Ein Aufruf von `pthread_cond_signal (pthread_cond_t c)` weckt einen Thread, der auf die Condition Variable `c` wartet.

2.4. Lockfreie Synchronisation

Eine Alternative zur blockierenden Synchronisation durch gegenseitigen Ausschluss ist die Benutzung von lockfreien Methoden. Ein paralleles Programm nennt man lockfrei, wenn es nicht blockiert und garantiert ist, dass jederzeit ein Thread existiert, der Fortschritte erzielt. Eine verschärfte Form sind wartefreie Algorithmen, siehe [37]. Diese garantieren die Terminierung einer Funktion in einer bestimmten Anzahl von Schritten. Wir werden hier nur auf lockfreie Synchronisation eingehen. Es liegt in der Natur der lockfreien Synchronisation, dass keine Deadlocks entstehen können.

Die lockfreie Synchronisation basiert auf atomaren Operationen. Eine atomare Operation entspricht einer einzigen Instruktion auf Prozessorenebene und kann als solche nicht von anderen Threads unterbrochen werden. Zur Synchronisation sind atomare Operationen interessant, die eine Lese- und eine Schreiboperation, meist noch mit einer bedingten Zuweisung, logisch zusammenfassen. Ein inkonsistenter Zustand der Daten durch eine Unterbrechung zwischen der Lese- und der Schreiboperation ist nicht möglich.

Für sich alleine genommen sind die atomaren Operationen, die wir gleich einführen werden, mächtig. Die theoretischen Möglichkeiten der Synchronisation sind groß. Jedoch ist die Verwendung komplex, denn die lockfreie Synchronisation kann fehlschlagen. So kann es in lockfreien Programmen nötig sein, Änderungen aufgrund fehlgeschlagener Synchronisation rückgängig machen zu müssen. Auch die Rückabwicklung wird mit atomaren Operationen synchronisiert. So kann auch die Rückabwicklung selbst gestört werden und es besteht die Gefahr, dass Threads verhungern.

Ein vollständig lockfreies, paralleles Programm benötigt sehr viele atomare Operationen im Zusammenhang. Die Komplexität dieser Programme ist sehr hoch, Lesbarkeit und Wartbarkeit sind gering. Neben der hohen Komplexität und der Gefahr des Verhungerns ist das ABA Problem eine weiterer Nachteil. Wir werden letzteres in Abschnitt 2.4.2 besprechen. Die Vorteile der lockfreien Synchronisation sind die Abwesenheit blockierter Threads und Deadlocks. Im weiteren Verlauf der Arbeit werden wir lockfreie Techniken und den gegenseitigen Ausschluss kombinieren.

Zunächst führen wir aber die wichtigsten atomaren Instruktionen ein und diskutieren zu jeder einen Anwendungsfall. Später in Abschnitt 2.5.2.3 besprechen wir eine lockfreie Deque.

2.4.1. Atomare Instruktionen

Die angesprochenen atomaren Operationen sind in ihrer Funktionsdefinition von System und Compiler abhängig. Wir beziehen uns auf den g++ Compiler der Version 4.4.3. Die entsprechenden Anweisungen sind im g++ Benutzerhandbuch [1] im Kapitel 5.47 "Built-in functions for atomic memory access" zu finden.

Die wichtigsten Operationen sind:

Fetch and Add (FAA)

Code 2.5 (FAA).

```
type __sync_fetch_and_add(type* x, type v)
{
  type h = *x;
  *x += v;
  return h;
}
```

Inkrementiert die Variable x um den Wert v und gibt den neuen Wert zurück.

Test and Set (TAS)

Code 2.6 (TAS).

```
type __sync_lock_test_and_set(int* x,int v)
{
  type h = *x;
  *x = v;
  return h;
}
```

Der Wert von x wird auf v gesetzt. Der Rückgabewert ist der vorherige Wert von x.

Compare ans Swap (CAS)

Code 2.7 (CAS).

```
bool __sync_bool_compare_and_swap(*type p, type old_v, type new_v)
{
  if (*p == old_v)
    {
      *p = new_v;
      return true;
    }
  else
```

```
        return false
    }
```

Die gegebene (Pointer-)Variable p wird auf Adresse new_v gesetzt, wenn p dem Wert old_v gleicht.

Anstelle des Platzhalters `type` erlaubt der g++ Compiler die Basistypen int, long, long long und unsigned Typen, sowie Pointer Typen. In den folgenden Code Beispielen werden wir eine vereinfachte Pseudocode Schreibweise für die Namen der Operationen benutzen.

2.4.1.1. FAA: Ein lockfreier Counter

Mit der Fetch and Add Operation (FAA) lässt sich einfach ein lockfreier Counter erstellen. Häufiger Einsatzort eines threadsicheren globalen Counters ist die Verwaltung der Größe einer Datenstruktur.

Code 2.8 (Lockfreier Conter).

```
class counter {
  int value;

public:
  counter() : value(0) {}

  int operator++() { return Fetch_And_Add(&value, 1); }
  int ++operator() { return Add_And_Fetch(&value, 1); }
};
```

Die Fetch and Add Operation führt schnell zu einer Implementierung wie der Klasse `counter`. Fetch and Add gibt es in verschiedenen Varianten, wie z.B. der vorgestellten und nachgestellten Addition.

2.4.1.2. TAS: Ein Spinlock ohne Systemmutex

Wir designen einen Spinlock Mutex unter Benutzung der Test and Set Operation (TAS). Wir setzen das Sperren eines Mutex mit der Wertänderung einer Variable gleich. Schafft ein Thread erfolgreich eine Wertänderung einer gegebenen Variable mit Test and Set von 0 auf 1, ist er der neue Besitzer. Das aktive Warten implementieren wir mit einer Schleife, die ausgeführt wird, bis die Sperrung erfolgreich ist.

Code 2.9 (Spinlock mit Test ans Set).

```
class spinlock {
  int mtx;

public:
  spinlock() : mtx(0) {}
```

```
  void lock()     { while (Test_And_Set(&mtx,1) == 1); }
  bool try_lock() { return Test_And_Set(&mtx,1) == 0; }
  void unlock()   { mtx = 0; }
};
```

Der Mutex ist gesperrt, wenn die Variable `mtx` den Wert 1 enthält, sonst 0. Beim Setzen des Werts von mtx mit Test and Set wird immer der vorige Wert von mtx zurückgegeben. Das Sperren ist also erfolgreich, wenn Test and Set 0 zurückliefert. Das Entsperren ist trivial.

In der Mutex Implementierung der Thread Hierarchie aus Abschnitt 2.3.5 benutzen wir statt eines `pthread` Mutex den vorgestellten Spinlock Mutex. Da die Mutex Klasse `da_mutex` ohnehin ein Spinlock Mutex ist, sparen wir uns mit dem Einsatz der Test and Set Operation die Deklaration eines echten Systemmutex aus der Bibliothek und dessen teurere Operationen.

2.4.1.3. CAS: Stack mit lockfreiem Push

Die Compare and Swap Operation (CAS) ist die mächtigste der vorgestellten atomaren Operationen. Mit dieser Operation lassen sich einzelne Zeiger und damit zeigerbasierte Datenstrukturen lockfrei synchronisieren.

Die übliche Synchronisation mit CAS läuft in mehreren Schritten ab. Zuerst wird die Speicheradresse des zu synchronisierenden Pointers ausgelesen. Daraufhin wird eine neue Adresse berechnet und mit CAS ausgetauscht. Die CAS Operation wird für gewöhnlich im Kopf einer Schleife ausgeführt und das Auslesen und Auswerten der alten Pointeradresse so oft im Schleifenkörper wiederholt bis CAS erfolgreich ist.

In unserem Beispiel synchronisieren wir die `push` Operation eines Stacks.

Code 2.10 (Lockfreies Push auf einen Stack).

```
template <class T>
class stack_with_cas {
  stack_item* top;
public:
  stack_with_cas() {top = NULL;}
  void push(T val)     {
    stack_item* new_top = new stack_item(val);
    do { stack_item* old_top = top;
         new_top->next = old_top;
       }
    while (!Compare_And_Swap(&top, old_top, new_top)); }
};
```

Zunächst erzeugen wir das neue Stack Element `new_top`, das wir auf den Stack legen wollen. Innerhalb des Schleifenkörpers setzen wir den Nachfolger von `new_top` im Stack auf das oberste Element des Stacks top. CAS prüft ob der Nachfolger von `new_top` noch gleich top ist. wenn ja, wird top ausgetauscht. Wenn nein, muss der Nachfolger von `new_top` neu berechnet werden.

Für den Fall, dass eine CAS Operation fehlschlägt benötigt jede Implementierung eine Backoff Strategie. In der vorgestellten Funktion ist nur ein Zeiger zu synchronisieren und so lässt sich die Backoff Strategie leicht in den Schleifenkörper verschachteln. Bei stark verzeigerten Datenstrukturen müssen oft mehrere Zeiger parallel synchronisiert werden. Die lockfreie Synchronistation jedes einzelnen Zeigers kann fehlschlagen, die Backoff Strategie muss eine immer mehr Zeiger einbeziehen und eine komplexere Situation handhaben. Die Backoff Strategie wird selbst wiedrum nur lockfrei synchronisiert. Ein Beispiel hierfür ist die lockfreie Deque von H. Sundell und P. Tsigas [26], die wir in Abschnitt 2.5.2.3 kurz ansprechen. Die stark steigende Komplexität lockfreier Algorithmen mit zunehmender Anzahl der gleichzeitig zu synchronisierenden Zeiger ist ein großer Nachteil der lockfreien Synchronisierung mit CAS. Aus diesem Grund werden wir die lockfreie Synchronisation nur sehr dosiert einsetzen.

Ein weitere Schwierigkeit begegnet uns bei der Synchronisation der `pop` Operation des Stack. Dort tritt das so genannte ABA Problem auf, siehe nächster Teilabschnitt.

2.4.2. CAS und das ABA Problem

Wir demonstrieren das ABA Problem an der naiven Implementierung der `pop` Funktion eines Stacks. Die folgende `pop` Funktion ist fehlerhaft:

Code 2.11 (ABA Problem: eine naive `pop` Operation auf einen Stack).

```
T pop(T val)     {
  stack_item* pop_item, new_top;

  do { pop_item = top;
    if (pop_item == null) return null;
       stack_item* new_top = pop_item->next;
     }
  while (!Compare_And_Swap(&top, pop_item, new_top));
  T return_val = pop_item->val;
  delete pop_item;
  return return_val; }
```

Der Thread liest das oberste Element des Stacks und dessen Nachfolger aus. Wenn die Adresse des ersten Stack Elements unverändert bleibt, verkleinert der Thread den Stapel und setzt den bekannten Nachfolger als oberstes Element ein.

Das Problem liegt unter der Annahme versteckt, dass zwei Elemente (`stack_items`) identisch sind, wenn sie die gleiche Adresse haben. Wir betrachten folgendes Szenario:

- Thread A überprüft mit Compare and Swap, ob der Zeiger `top` noch auf das ausgelesene `stack_item` I_1 zeigt.
- Das `stack_item` I_1, das Thread A kennt, wurde in der Zwischenzeit von Thread B vom Stapel geholt.
- Weiter hat Thread B ein `stack_item` I_2 auf den Stapel gelegt.

- Thread B legt danach auch `stack_item` I_1 wieder auf den Stapel, ggf mit anderem Inhalt, sicher aber mit einem anderen Nachfolger (I_2).
- Daraufhin akzeptiert Thread A `stack_item` I_1 als gültig. Jedoch hat sich dessen Inhalt geändert. Der Nachfolger von I_1 ist jetzt I_2 und nicht das Element `new_top`, das A ursprünglich errechnet hat. Diese Veränderung wird von der CAS Operation nicht erkannt.

Das grundlegende Problem in diesem Beispiel ist, dass ohne vorhandenes Speichermanagement zwei Pointer gleichzeitig synchronisiert werden müssen (pop_item, new_top). Das kann CAS nicht leisten.

Zum ABA Problem gibt es verschiedene mehr oder weniger praktikable Lösungen. Herlihy [37] hat gezeigt, dass es einen allgemeinen, komplexen Ansatz gibt, mit dem man durch CAS jede Datenstruktur synchronisieren kann. Weiter gibt es theoretische Überlegungen, die sich eine DCAS (Double Compare and Swap) oder CAS2 Operation zu nutze machen. DCAS synchronisiert zwei Zeiger gleichzeitig, CAS2 zwei zusammenhängende Speicheradressen. Beide Operation sind aber in gewöhnlicher Hardware nicht implementiert. Der Nutzen der DCAS Operation wird in [41] analysiert.

Praktikable ist die Idee die zu synchronisierenden Zeiger mit einem Counter zu versehen. Jeder Thread, der den Zeiger in Vorbereitung auf eine schreibende Operation ließt, inkrementiert den Counter. Da die meisten 64-bit Architekturen doppelte Wörter adressieren, lässt sich der Counter in den letzten 8 Bits eines Zeigers verstecken. Zeiger und Counter können gleichzeitig mit einer CAS Operation synchronisiert werden. Gleichzeitig verhindert der inkrementierte Counter einer Verwechslung von Dateninstanzen an der gleichen Speicheradresse.

Ein andere Lösung, die auf einem einfachen Speichermanagement basiert wurde von M. Michael entwickelt, siehe [42] und [43]. Im folgenden Abschnitt präsentieren wir diese Lösung des ABA Problems für einen den Stack. In Abschnitt 2.5.2.3 stellen wir eine lockfreie Deque vor, die korrekt synchronisiert.

2.4.3. Ein lockfreier Stack

Eine einfache Implmentierung eines lockfreien Stacks mit einer Lösung des ABA Problems liefert M. M. Michael in [42] und [43]. Die Lösung basiert auf einer sicheren Speicherverwaltung durch Hazard Pointers.

Hazard Pointers sind Pointer im gemeinsamen Speicherbereich der Threads, auf die jeweils nur ein Thread schreibend und alle anderen lesend zugreifen können. Jedem Thread wird eine kleine, begrenzte Zahl an Hazard Pointers für Schreibzugriffe zugewiesen. Ein Thread benutzt einen Hazard Pointer, um einen Adresse zu markieren, auf die er potentiell schreibend zugreift.

Der lockfreie Stack, den wir hier vorstellen verwendet die `push` Funktion aus Abschnitt 2.4.1.3. Im folgenden Code verwenden wir den Zeiger auf einen Hazard Pointer `*hp`. Praktisch werden die Hazard Pointers der Threads gemeinsam in einem Feld verwaltet.

Code 2.12 (ABA-freie `pop` Operation auf einen Stack)**.**

```
T pop(T val)     {
  stack_item* pop_item, new_top;

  do { pop_item = top;
    if (pop_item == null) return null;
    *hp = pop_item;
    if (pop_item != top) continue;
       stack_item* new_top = pop_item->next;
     }
  while (!Compare_And_Swap(&top, pop_item, new_top));
  *hp = null;
  T return_val = pop_item->val;
  DeleteNode(pop_item);
  return return_val; }
```

Die einzige Funktion der Hazard Pointer ist Speicheradressen zu markieren, die noch von Threads verwendet werden. Erst wenn die Adresse nicht mehr verwendet wird, darf das Element recycelt werden.

Die Funktion `DeleteNode` legt den Knoten in einem Feld des Threads mit zu löschenden Elementen ab. Der Thread versucht regelmäßig das Feld sicher zu bereinigen in dem er für jeden Knoten überprüft, ob noch ein Hazard Pointer eines anderen Threads gesetzt ist. Da eine Speicheradresse nicht wiederverwendet werden kann, solange ein Thread sie noch markiert hat, kann eine Verwechslung die zum ABA Problem führt nicht auftreten. Details zur Funktion `DeleteNode` sind [43] zu entnehmen.

Weitere lockfreie Stacks werden von Dechev et al. in [45] und von Hendler et al. in [44]. Beide Lösungen fallen bei der Vermeidung des ABA Problems wiederum auf die Arbeit von M. Michael ([43]) zurück.

2.5. Threadsichere Datenstrukturen

Die Threadsicherheit der verwendeten Datenstrukturen ist entscheidend für der Korrektheit eines parallelen Algorithmus. Eine Datenstruktur ist threadsicher, wenn deren Funktionen synchronisiert sind, so dass sie von beliebig vielen Threads gleichzeitig aufgerufen werden können ohne inkonsistente Zustände zu verursachen. Diese Datenstrukturen sind allerdings schwierig zu implementieren und können mit hohen Zusatzkosten verbunden sein.

Wenn wir Datenstrukturen, wie beispielsweise Listen oder Stacks, threadsicher designen und die Synchronisation vollständig in den Funktionen `push` und `pop` kapseln, erhalten wir schnell einsetzbare und einfach zu handhabende Strukturen. Die Kapselung hat aber zur Folge das zunächst nur einzelne Operationen auf die Daten synchronisiert werden können. Eine zusammenhängende Reihe von `push` oder `pop` Operationen als ganzes einfach zu synchronisieren ist so nicht möglich.

In der Computational Geometry haben wir es oft mit komplexen, verzeigerten Strukturen zu tun. Ein einzelne logische Änderung an diesen Strukturen betrifft meist mehrere

Elemente und wird durch eine ganze Reihe von Operationen implementiert. Mit vorgefassten, gekapselten Operationen sind konsistente Änderungen an diesen Strukturen kaum zu synchronisieren. Um den spezifischen Anforderungen der Algorithmen gerecht zu werden, benötigen wir häufig effiziente und angepasste Strukturen, siehe beispielsweise Abschnitt 4.4.1. Auch die Vermeidung von Deadlocks ist in spezialisierten Strukturen einfacher mit den Methoden zu realisieren, die wir in Abschnitt 2.3.2 diskutiert haben.

Für einfache Anwendungsfälle können threadsichere Datenstrukturen, die die Synchronisation vollständig kapseln, allerdings nützlich sein. Bevor wir einige dieser Strukturen vorstellen, werden wir zuerst das grundsätzliche Design unserer geometrischer Datenstrukturen diskutieren. In den späteren Kapiteln werden wir spezifische Datenstrukturen für verschiedene Algorithmen implementieren, welche effizient die Threadsicherheit garantieren.

2.5.1. Design von verlinkten, geometrischen Strukturen

Komplexere Datenstrukturen bestehen üblicherweise aus mehreren Komponenten: einem Container (eine Liste oder ein Graph), der die Struktur verwaltet, und einem oder mehrerer eingebetteter, verzeigerter Objekte (Knoten, Kanten). Im Kontext dieser Arbeit handelt es sich bei Letzteren üblicherweise um Repräsentanten geometrischer Objekte.

An Container und eingebettete Objekte werden unterschiedliche Anforderungen gestellt. Das Design der eingebetteten Objekte ist zudem sehr stark vom betrachteten Problem und dem verwendeten Algorithmus abhängig. So wissen wir von der Mechanik der Divide und Conquer Algorithmen, dass Teilprobleme unabhängig voneinander behandelt werden. Teilprobleme werden dabei durch die Objekte repräsentiert. Bei den randomisiert inkrementellen Algorithmen wird eine Datenstruktur an einer zufälligen Stelle von konkurrierenden Threads erweitert oder verändert, was mitunter sogar Löschoperationen auf gemeinsame Objekte nötig macht. Ins Detail werden wir hierzu erst in späteren Kapiteln gehen. Wir werden dort sehen, dass die eingebetteten Objekte sehr individuell anzupassen sind und threadsichere Datenstrukturen für Divide und Conquer Algorithmen tendenziell viel einfacher zu implementieren sind als für inkrementelle Algorithmen.

Wo die Anforderungen an die Objekte sehr verschieden sind, gibt es beim Design der Container große Übereinstimmungen. Wir haben zur Implementierung der Container folgende Regeln aufgestellt:

1. Die Anzahl der Variablen der Containerklasse ist zu minimieren. Variablen des Containers sind aus der Sicht der eingebetteten Objekte global und im Fall einer Schreiboperation zu sichern. Eine unübersichtliche Zahl beschreibbarer, globaler Variablen provoziert Deadlocks.

2. Variablen der Containerklasse sollen, wann immer möglich, als Konstanten deklariert werden. Da Konstanten threadsicher sind, sorgt die Maßnahme für Sicherheit und Übersicht.

   Ein Containerklasse enthält immer mindestens einen Zeiger auf ein eingebettetes Objekt. Diese dienen den Threads häufig als Einstiegspunkt in die Datenstruktur

und stellen einen Flaschenhals dar, der nicht durch eine Synchronisation weiter verengt werden sollte. Im dieser Arbeit werden wir häufig Datenstrukturen mit einer geometrischen Grundform initialisieren. In vielen Fällen bleibt diese Grundform während der gesamten Berechnung erhalten.

3. Sind konkurrierende Schreiboperation auf Containervariablen unvermeidlich, soll die lockfreie Synchronisation, wo immer möglich, verwendet werden. Auch diese Maßnahme dient der Deadlockvermeidung und in einfachen Fällen der Einsparung von Rechenzeit.

   Eine typische Anforderungen an eine Containerklasse ist ein Counter, der ihre Größe angibt. Einen lockfreien Counter haben wir in Abschnitt 2.4.1.1 beschrieben.

Nachdem wir einige einfache threadsichere Datenstrukturen diskutiert haben, werden wir ein threadsicheres Design des Containers des Leda Graphen vorstellen, siehe Abschnitt 2.5.3.

2.5.2. Implementierung von Deques und Listen

Wir möchten eine threadsichere Datenstruktur zum Verwalten von Mengen implementieren. Zur einfachen Benutzung der Struktur wollen wir die Synchronisation vollständig in der Datenstruktur kapseln. Wir diskutieren dazu 3 Implementierungen, die auf die verschiedene Synchronisationstechniken zurückgreifen:

1. Eine Deque, deren Enden von Mutex Variablen naiv geschützt werden.

2. Eine Liste, die auf der Verwaltung von Teillisten basiert und jedem Benutzer (Thread) genau eine Teilliste zuordnet.

3. Eine lockfreie Deque basierend auf CAS.

Die ersten beiden sind einfachere Implementierung mit Mitteln, die die pthread Bibliothek zur Verfügung stellt.

Die lockfreie Datenstruktur basiert auf Arbeiten von H. Sundell und P. Tsigas. Die verwendete Deque ist beschrieben in [26], eine Erweiterung zur lockfreien Liste ist in [27] von H. Sundell zu finden. Die Autoren haben eine Reihe weitere lockfreier Datenstrukturen entworfen: eine lockfreie Fifo Queue [30], eine lockfreie Priority Queue [31], ein lockfreies Wörterbuch [32] und wartefreies Reference Counting [33].

Ein Anwendungsfall für Mengen ist die Speicherverwaltung komplexer Datenstrukturen, beispielsweise Graphen. Diese verwalten häufig eine Mengen aller ihrer Objekte. Die Mengen machen alle Objekte einer stark verzeigerten Struktur leichter verfügbar und dienen zudem der Speicherverwaltung. Bei der parallelen Konstruktion großer Datenstrukturen ist mit einer hohen Anzahl parallel lesender und schreibender Zugriffe zu rechnen.

Wir geben in den folgenden Abschnitten einen Überblick über die Implementierung und stellen die Leistungsfähigkeit der Strukturen im Kapitel Experimente 5.3 gegenüber.

2.5.2.1. Deqeue mit Mutex Variablen

Der naivste Ansatz greift auf eine vorhanden Deque Implementierung und einen globalen Mutex zurück. Vor jeder Operation mit der Deque muss der globale Mutex gesperrt werden. Dieser Weg führt zu einer maximalen Zahl von Kollisionen der zugreifenden Threads.

Eine einfache Erweiterung ist die Einführung eines zweiten Mutex mit dem das vordere und hintere Ende getrennt gesperrt werden. Ist die Deque hinreichend befüllt können beide Enden unabhängig voneinander bedient werden. Bei einer Länge der Deque kleiner oder gleich 3 Elementen muss ein Thread beide Enden sperren. Wir definieren die Minimalgröße von 3 Elementen, damit in jedem Fall ein Puffer zwischen Kopf und Ende bei gleichzeitigen Operationen an beiden Seiten liegt. Zur threadsicheren Bestimmung der Länge führen wir einen lockfreien Counter mit. Außerdem benutzen wir eine Mutex Hierarchie um sicherzustellen, dass keine Deadlocks entstehen. Wir definieren das der Mutex des Kopfs den Vorrang vor dem Mutex am Ende hat.

Die Lösung ist einfach, skaliert aber nach wie vor nicht gut mit vielen Threads. Für den Fall das viele Threads gleichzeitig die Datenstruktur befüllen, kommt es sehr häufig zu konkurrierenden Zugriffen und gegenseitigen Blockaden.

2.5.2.2. Deque mit threadbasiertem Zugriff

In diesem Ansatz entwerfen wir eine Deque D, die eine Menge von Sub-Deques verwaltet. D ist eine Deque über Deques. Jede Sub-Deque ist dabei genau einem Thread zugeordnet und wird durch einen einzelnen Mutex gesichert. Die Deque D ist allen Threads bekannt und der Einstiegspunkt zu den Sub-Deques mit denen die Threads hauptsächlich operieren. Mit der dezidierten Zuordnung von Deques zu Threads und dem sparsamen Umgang mit Mutex Variablen wollen wir Kollisionen und Aufwand minimieren.

Die Zuordnung von Sub-Deques zu Threads setzen wir mit dem Key Mechanismus der pthread API um. Die API gibt uns die Möglichkeit einen Key zu definieren, der allen Threads bekannt ist. Ein einzelner Key assoziiert mit jedem Thread eine andere Speicheradresse. Jeder Thread kann aus einer Key Variable nur den eigenen Wert auslesen. Die wichtigsten Typen und Funktionen sind:

pthread_key_t ist der Typ eines Keys.

void *pthread_getspecific(pthread_key_t key) liefert den Wert des Keys zurück.

int pthread_setspecific(pthread_key_t key, const void *value) setzt den Wert des Keys.

Die Deque D implementieren wir in der Klasse `thread_list`, die Sub-Deques in der Klasse `sub_list`.

Code 2.13 (Deklaration der Klasse `thread_list`, Ausschnitt).

```
template <typename E>
class thread_list {
```

```
    int size;
    pthread_key_t sub_list_key;
    spinlock mutex;

    class sub_list;
    sub_list* first;
    sub_list* last;

public:
    void push_back(const E& element);
    E* pop_back();
    int size();
    ...
};
```

Die Klasse initialisiert den Key `sub_list_key` zur threadbasierten Ablage der Sub-Deques. Sie verwaltet eine doppelt verkettete Liste aller Sub-Deques, dazu sind Kopf (`first`) und Ende (`last`) der Sub-Deque Liste bekannt. Die Variablen der Klasse werden vom Mutex `mutex` geschützt.

Die Klasse `sub_list` instanziiert die Listen der einzelnen Threads. Sie implementiert eine gewöhnliche doppelt verkettete Liste. Zusätzlich ist eine `sub_list` selbst Teil einer Liste und enthält Zeiger auf Vorgänger `pred` und Nachfolger `succ`. Weiter werden alle ihre Funktionen durch einen Mutex geschützt.

Code 2.14 (Definition der Klasse `sub_list`, Ausschnitt).

```
template <typename E>
class sub_list {
    spinlock mutex;
    sub_list* succ;
    sub_list* pred;
    ...

public:
    void push_back(const E& element);
    E* pop_back();
    ...
};
```

Die Funktionen der Klasse `sub_list` sowie die Klasse selbst sind dem zugreifenden Thread unbekannt und werden durch die Deque D maskiert.

Die `push` Funktion der Deque D setzt sich aus zwei Teilen zusammen. Zuerst überprüft die Funktion durch Aufrufen des Key Interfaces, ob zum ausführenden Thread eine `sub_list` existiert. Falls nicht wird diese angelegt, im Key gespeichert und der Liste aus `sub_lists` in D hinzugefügt. Ist eine `sub_list` zum Thread vorhanden, wird dessen lokale `push` Funktion ausgeführt.

Code 2.15 (Push Funktion der Klasse `thread_list`).

```
void push_back(const E& element)
{
  // Check if a Sublist for the Key exists
  sub_list* l = (sub_list*) pthread_getspecific(sub_list_key);

  if (l == 0)
    {
      l = new sub_list();
      mutex.lock();
      // Add the new Sub-Deque to the list of Sub-Deques
      ...
      mutex.unlock();
      pthread_setspecific(sub_list_key, (void*) l);
    }

  // Push Element on the Sublist
  l->mutex.lock();
  l->push_back(element);
  l->mutex.unlock();
  __sync_add_and_fetch( & size, 1 );
}
```

Die `pop` Funktion von `thread_list` wendet die `pop` Funktion der Deque des Threads an, falls diese existiert und nicht leer ist. Ansonsten iteriert die Funktion durch die Liste der `sub_lists` und führt `pop` auf der ersten nicht leeren Liste aus.

Code 2.16 (Pop Funktion der Klasse `thread_list`).

```
E* pop_back()
{
  sub_list* l = (sub_list*) pthread_getspecific(sub_list_key);

 // If l exists and is non empty, execute the local pop operation
 // and return.
  if (l != 0)
    { // call local pop, return on success }

  mutex.lock();
  l = last;
  mutex.unlock();

  while (l != 0) {
      // call local pop, return on success

      mutex.lock();
```

```
      l = l->pred;
      mutex.unlock();
    }
  return 0;
}
```

Der Einsatz von Mutex Variablen erfolgt in dieser Struktur primär auf Ebene der (Teil-)listen. Die Listen werden jeweils vollständig gesperrt. Die Threads sperren jeweils nur einen Mutex der Listen gleichzeitig. Es können keine Deadlocks entstehen. Da wir pro Liste nur einen Mutex benutzen, ist die Mutex Granularität grob.

Wir nehmen an, dass während des Befüllens der Struktur keine Kollisionen auftreten, da die Threads in Ihren eigenen Listen arbeiten. Beim Entleeren müssen wir zwei Fälle unterscheiden: Zum Einen ist denkbar, dass die Threads beliebige Elemente entfernen und so zuerst die eigene Liste leer räumen können. Dann sind Kollisionen selten. Zum Anderen ist es möglich das konkrete Elemente entfernt werden sollen. Diese sind voraussichtlich in anderen Listen als der des löschenden Threads und es kommt wahrscheinlich zur Kollision.

Ein Anwendungsfall dieser speziellen Mengenimplementierung ist das Job Stealing. In der Multithread Programmierung bedeutet Job Stealing, dass Threads ihnen zugeteilte Arbeit (Jobs) zunächst in einer eigenen Queue verwalten. Um die Arbeitsauslastung aller Threads zu verbessern, können Threads ohne Arbeit Jobs anderer Threads stehlen (Job Stealing). Die hierzu noch fehlende Funktionalität der präsentierten Deque lässt sich leicht ergänzen.

Wenn die threadbasierte Deque von vielen Threads über einen längeren Zeitraum bedient wird, besteht die Möglichkeit, dass die Deque in viele kleine Sub-Deques zerfällt. Darunter können Sub-Deques sein, deren assoziierte Threads schon terminiert sind. Es kann daher aus Effizienzgründen nötig werden eine Reinigungsfunktion in der Deque zu implementieren, welche die Deque defragmentiert. Dies ist weiter zu untersuchen.

2.5.2.3. Eine lockfreie Deque

Wir geben einen kurzen Überblick über die lockfreie Deque von H. Sundell und P. Tsigas [26]. Diese ist - nach Aussage der Autoren - die erste lockfreie Implementierung die mit CAS synchronisiert wird, mit der beide Seiten der Deque unabhängig voneinander bedient werden können und für die keine Beschränkung der Länge gilt.

Bisher bekannte lockfreie Implementierungen einer Deque hatten verschiedene spezifische Nachteile:

- Sie unterstützen nur einen Teil der Deque Operationen, haben also Einschränkungen bzgl. der konkurrierenden Nutzung ([38]).
- Sie basieren auf der Nutzung von DCAS (lockfreie Synchronisierung von 2 Zeigern gleichzeitig, auch CAS2 genannt, siehe [39, 40]).
- Sie arbeiten mit einer erweiterten CAS Funktion, die zwei zusammenhängende Wörter gleichzeitig synchronisiert ([35, 36]).

- Sie benötigen einen Hilfselemente zwischen den benachbarten Deque Elementen ([34]).

Die Deque wird wie eine doppelt verkettete Liste konstruiert, also aus Knoten die einen Wert, einen Zeiger auf den Vorgänger und einen Zeiger auf den Nachfolger enthalten. Um einen Knoten hinzuzufügen oder zu löschen müssen beide Zeiger des Knoten und der Nachbarn angepasst werden. Da CAS nur einen Zeiger pro Synchronisierung ändern kann, wird die Liste als einfach verkettet interpretiert. Der Zeiger auf den Nachfolger wird immer als erstes aktualisiert mit dem Ziel jederzeit einen konsistenten Zustand der einfachen Verkettung herzustellen. Der Zeiger auf den Vorgänger hat hingegen den Charakter eines Hinweises. Dahinter steht die Beobachtung, dass ein veralteter Vorgänger Zeiger entweder direkt oder einige Elemente weiter vor den aktuellen Vorgänger zeigt. Der Algorithmus muss zur Korrektur ggf. einige Knoten weitergehen, um den korrekten Vorgänger zu finden. Die unterschiedliche Behandlung von Vorgänger und Nachfolger Zeigern führt zu asymmetrischen `push` und `pop` Funktionen am Anfang und Ende der Queue.

Ein schweres Problem bei der lockfreien Synchronisierung von einfach verketteten lockfreien Listen ist sicherzustellen, dass der Vorgänger A eines gerade eingefügten Knoten B nicht im gleichen Moment gelöscht wird. Mit der Löschung von A, wäre auch B nicht mehr in der Liste verkettet. Eine theoretische Lösung ist hier DCAS. Andere Ansätze benutzen Hilfsknoten zwischen allen benachbarten Elementen. In der Lösung von H. Sundell kommt ein Markierungsbit zur Anwendung, um den Zustand (gültig/gelöscht) des Knoten festzuhalten. Als Speicherort des Markierungsbits missbraucht Sundell die letzte Stelle des Vorgänger/Nachfolger Zeigers. Dieses Bit ist bei wortadressierten System ohnehin immer null. Funktionen zum Auslesen der korrekten Adresse/des Bits müssen entsprechend implementiert werden. Vorteil der Methode ist, dass Zeiger und Bit zusammen in einer CAS Operation synchronisiert werden.

Wichtig zur effizienten Implementierung der lockfreien Deque ist ein passendes Speichermanagement. Das Speichermanagement übernimmt die Lösung des ABA Problems und die Wiederverwendung der Knoten, nachdem diese aus der Deque entfernt wurden. Beide Probleme hängen zusammen. Zum einen ist es nicht unmittelbar möglich ein Element, das aus der Deque entfernt wird, zu dealloziieren. Mit der lockfreien Synchronisation ist es schwer zu garantieren, dass kein andere Thread gerade den Zeiger auf das betroffenes Element dereferenziert. Zum anderen muss ein Mechanismus überprüfen, dass ein entfernter Knoten nicht wiederverwendet wird, solange andere Threads auf ihn referenzieren. Wie von H. Sundell [26] empfohlen, benutzen wir das Speichermanagment von Valois [29] mit den Korrekturen von Michael and Scott [28]. Die Kernidee ist die Verwendung eines Referenzcounters.

Der kompletten Pseudocode ist bei H. Sundell [27] zu finden. Dort sind auch detailliertere Erklärungen, die Erweiterung der Listenfunktionen und ein Beweis zur Korrektheit zu finden. Details zum Speichermanagment sind bei Michael and Scott [28] zu entnehmen.

Bei der Umsetzung dieser Deque von Pseudocode in C++ Code hat sich gezeigt, dass die lockfreie Synchronisation mit CAS sehr komplex, unübersichtlich und fehleranfällig ist.

2.5.3. Der Leda Graph

Die Leda Graph Struktur bietet einer sehr umfassendes Paket von Graphfunktionen. Sie ist zentraler Bestandteil vieler Algorithmen der Leda Bibliothek, insbesondere vieler Algorithmen aus der Computational Geometry. Die Leda Graph Struktur unterstützt unter anderem Einbettung, Bidirektionalität, Maps und Ordnungserhaltung, siehe [7]. Bevor wir auf die Threadsicherheit eingehen werden wir noch kurz die genannten Eigenschaften diskutieren.

2.5.3.1. Eigenschaften des Leda Graphen

Für die Implementierung geometrischer Algorithmen sind die Eigenschaften einer Map und die Ordnungserhaltung wichtig. Ein Map basiert auf Bidirektionalität.

Definition 2.1 (Bidirektionalität)**.** *Ein gerichteter Graph $G = (V, E)$ heißt bidirektional, wenn für jede Kanten $e = (v, w)$ mit $e \in E$ und $v, w \in V$ eine Gegenkante e' mit $e' = (w, v)$ existiert. Die Abbildung von Kante zu Gegenkante ist bijektiv.*

Definition 2.2 (Map)**.** *Ist ein gerichteter Graph bidirektional und sind alle Zuordnungen zwischen Kante und Gegenkante im Graphen definiert, dann handelt es sich um eine Map.*

Mit Maps fällt es leichter durch die vorhandenen Kanten zu navigieren. Da es sich um gerichtet Kanten handelt, sind Anfangs- und Endpunkt klar definiert. Bei einer Iteration über die Adjazenzliste eines Knoten muss das gerade gegenüberliegende Ende der Kante nicht durch einen Test ermittelt werden. Durch die Definition der Gegenkante ist ein Sprung von einem Knoten zum nächsten und eine wiederholte Iteration über die ausgehenden Kanten einfach zu codieren.

Definition 2.3 (Planare Einbettung)**.** *Ein Graph heißt planar eingebettet, wenn sich die Knoten so in zwei Dimensionen abbilden lassen, dass sich die Kanten nicht überschneiden.*

Definition 2.4 (Ordungserhaltende Einbettung)**.** *Wenn die aus- und eingehenden Kanten in den Adjazenzlisten der Knoten so geordnet sind, dass sie der Reihenfolge der adjazenten Knoten gegen den Uhrzeigersinn in der Einbettung entsprechen, ist die Graphstruktur ordnungserhaltend.*

Die Ordnungserhaltung garantiert eine Sortierung der Kanten in den Adjazenzlisten. Zusammen mit der Map Eigenschaft ist einfach gezielt durch den Graphen zu navigieren. Mit einer gegebenen Kante ist es ein Leichtes in einer ordungserhaltenden Map eine Facette zum Umlaufen.

Beispiel 2.1 (Umlaufen einer Facette)**.**
Wir möchten im Graph G die Facette f umlaufen, die links von einer gegebenen, gerichteten Kante e liegt. Wir benutzen die Leda Funktionen:

- `edge G.pred_edge(edge e)`, welche zu einer Kante e den Vorgänger in der Adjazenzliste liefert.

- `edge G.reversal(edge e)`, welche zu einer Kante die Gegenkante zurückgibt.

```
edge e2 = e;
do {
e2 = G.reversal(e2);
e2 = G.succ_edge(e2);
} while (e2 != e)
```

Durch die Wahl der Gegenkante $reversal(e2)$ vertauschen wir Start- und Endknoten in $e2$. Die nachfolgende Kante $succ\_edge(e2)$ in der Adjazenzliste führt zum nächsten adjazenten Knoten im Uhrzeigersinn. Wir umrunden f gegen den Uhrzeigersinn.

Viele der Graphalgorithmen der Leda Bibliothek basieren auf ordnungserhaltenden oder eingebetteten Leda Graphen. Eine Weiterentwicklung des Leda Graphen ist der parametrisierte Graph, der in geometrischen Algorithmen zur Anwendung kommt. Der parametrisierte Graph ist eine Ableitung des Leda Graphen. Mit ihm lassen sich verschärfte Konventionen implementieren:

- Die Knoten des parametrisierten Graphen sind mit Punkten assoziiert.
- Die Kanten stellen Segmente dar und sind damit auf die direkte Linie zwischen den Knoten festgelegt.
- Die Graphen sind ordnungserhaltende Maps, d.h. die Kanten sind in den Adjazenzlisten hinsichtlich der geometrischen Lage ihrer Punkte mit oder gegen den Uhrzeigersinn sortiert.

Diese Technik ermöglicht eine übersichtliche Darstellung und leichte Behandlung komplexer, geometrischer Situationen. In dieser Arbeit werden wir uns dies in späteren Kapiteln zu nutze machen, unter anderem bei der Parallelisierung von Triangulierungen, siehe Abschnitt 3.6.

2.5.3.2. Der Leda Graph mit threadsicherer Kontainerklasse

Eine mächtige, threadsichere Graphstruktur ist für die Entwicklung paralleler, geometrischer Algorithmen sehr nützlich. Wir wollen die Techniken zur Threadsicherheit aus dem letzten Abschnitt auf den Leda Graphen anwenden.

Verschiedene Algorithmen stellen sehr unterschiedliche Anforderungen an die Threadsicherheit des Graphen. Da eine umfassende Sicherheit einen hohen Aufwand bedeutet, ist es unser Ziel zunächst nur diejenigen Funktionen des Leda Graphen zu sichern, die private Objekte der Containerklasse manipulieren. Wir wollen die Speicherverwaltung des Leda Graphen und die zugehörigen Funktionen, welche das Erzeugen, das Entfernen und das Verwalten von Knoten und Kanten betreffen, absichern.

Die Struktur des Leda Graphen ist (vereinfacht) in Abbildung 2.1 dargestellt. Die Basisklasse aller Graphobjekte, also Knoten, Kanten und im weiteren auch Facetten, ist

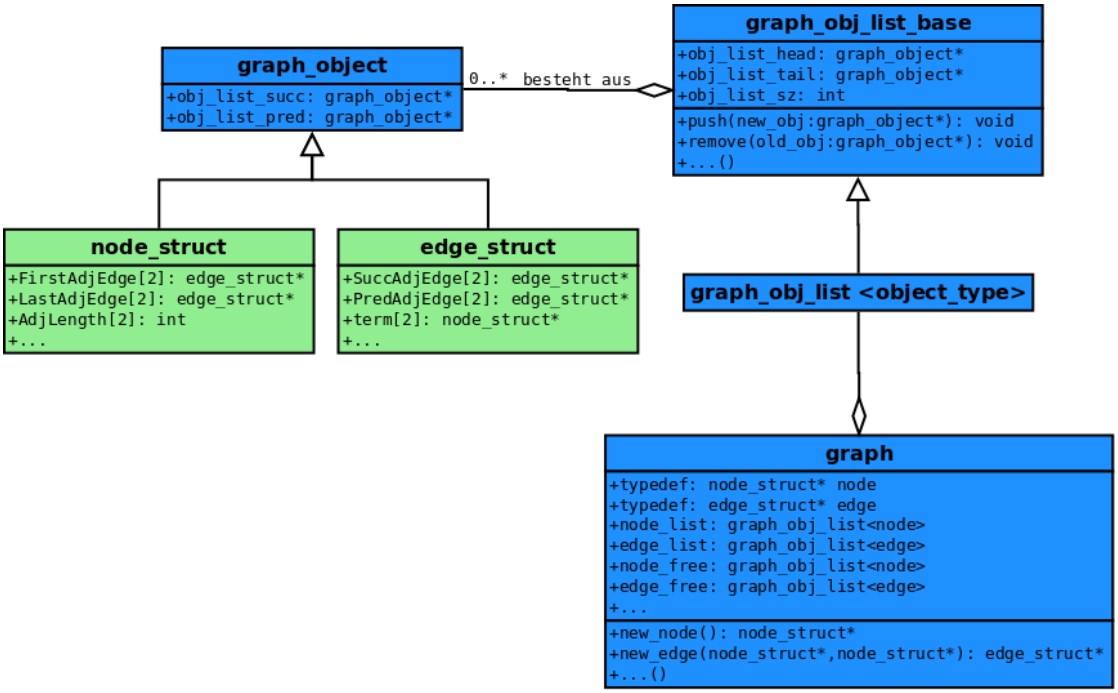


Abbildung 2.1.: UML Objekt Model der wichtigsten Leda Graph Klassen

die Klasse `graph_object`. Die Instanzen von `graph_object` sind doppelt verkettete Listenelemente, die im Listencontainer `graph_obj_list_base` ihre Anwendung finden. Entsprechend enthält die Klasse `graph_object` Zeiger und die Klasse `graph_obj_list_base` Listenfunktionen. Die Klassen `node_struct` und `edge_struct` repräsentieren die Knoten und Kanten des Graphen und sind von `graph_object` abgeleitet. Die eingehenden und ausgehenden Kanten sind wiederum zyklisch doppelt verkettet und werden jeweils als Adjazenzlisten in den Knoten verwaltet.

Analog zur Ableitung der Knoten und Kanten vom Graphobjekt wird die templatisierte Klasse `graph_object_list<>` von der `graph_obj_list_base` Liste abgeleitete. Die Instanzen von `graph_object_list<>` repräsentieren die Mengen der typisierten Knoten- und Kantenobjekte eines Graphen.

Der Graph enthält jeweils ein Listenpaar für Knoten (`node_list/node_free`) und Kanten (`edge_list/edge_free`). In der einen Liste werden jeweils die verwendeten Objekte aufbewahrt in der anderen, die derzeit freien. Der Grund je zwei Listen mitzuführen liegt im Leda Speichermanagement. Das Management alloziiert aus Effizienzgründen Objekte blockweise im Speicher und stellt diese einzeln zur Verfügung. Eine `delete` Funktion für Leda Objekte existiert nicht. Die vorübergehend nicht gebrauchten Elemente werden daher vom Leda Graphen in Listen zwischengelagert. Das Leda Speichermanagement selbst ist threadsicher.

Die grün markierten Klassen der Knoten und Kanten enthalten die Funktionalität, die für Design und Implementierung von Algorithmen wichtig ist. Die Verwaltung der

Graphobjekte findet jedoch stets im Hintergrund der Graphstruktur statt, betrifft also die blau markierten Klassen des UML Diagramms. Diese Klassen machen wir bezüglich der Funktionen Hinzufügen und Entfernen von Graphobjekten threadsicher. Dazu benutzen wir die threadbasierte Liste, die wir in Abschnitt 2.5.2.2 eingeführt haben.

Wir nehmen mehrere Anpassungen am Objektdiagramm vor. Anstelle einer Instanz der Liste `graph_obj_list_base` pro Graph und Nutzungszweck, benutzen wir eine Instanz pro Thread und pro Nutzungszweck. Das heißt, dass jedem Thread, der ein Graphobjekte löscht oder hinzufügt, mehrere Listen vom Typ `graph_obj_list_base` exklusiv zugeordnet werden. Mehrere Listen deshalb, da jeder Thread jeweils zwei Listen für benutzte und freie Objekte jeweils für Knoten und Kanten erhält. Die Liste vom Typ `graph_obj_list_base` entspricht der Listenklasse `sub_list` aus Abschnitt 2.5.2.2.

Die einzelnen Listen des Typs `graph_obj_list_base` bilden mit der Hilfe der Zeiger `obj_list_pred/obj_list_succ` selbst wieder eine doppelt verkettete Liste, die im Graph verankert ist. Die Verkettung der Listen entspricht der Verkettung in der Klasse `thread_list` aus Abschnitt 2.5.2.2. Weiter führen wir Variablen ein, um die Gesamtzahl der Knoten und Kanten mitzuführen (`node_num, edge_num`) und einen Mutex (`mutex`) zur Absicherung der Listenfunktionen.

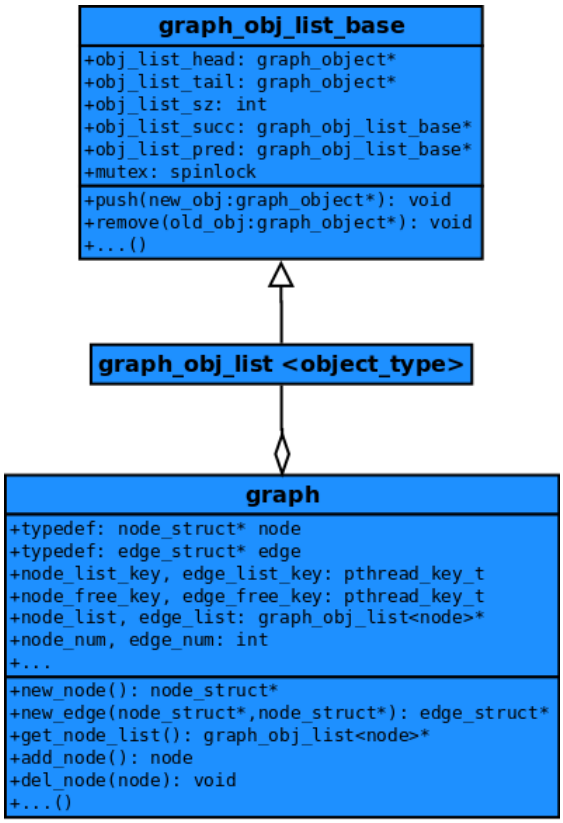


Abbildung 2.2.: Anpassungen zur Threadsicherheit der Leda Graphklassen

Der Graph verwendet statt der Klasse `graph_obj_list_base` die abgeleitet Klasse

graph_object_list<>. Die Instanzen der Klasse graph_object_list<> bilden eine doppelt verkettete Liste. Anfang und Ende dieser Listen werden im Graph abgelegt. Beispielsweise ist der Listenkopf der Listen der verwendeten Knoten im Zeiger node_list gespeichert.

Da die einzelnen Instanzen der Klasse graph_object_list<> Threads zugeordnet sind, führen wir mehrere pthread_keys ein. Jede der Listen über Listen hat einen eigenen Schlüssel. Der Schlüssel zur Liste der aktiven Knoten heißt node_list_key, der Schlüssel zur Liste der freien Knoten node_free_key. Die Schlüssel zu den Listen der Kanten werden analog benannt.

Die Anlage neuer Listen vom Typ graph_object_list<> funktioniert dabei analog der Funktion push_back der threadbasierten Listen und wird von den Hilfsfunktionen get_*listname*() implementiert. Die Klasse graph_obj_list_base enthält einen Spinlock Mutex mit dem alle Listenoperationen dieser Klasse abgesichert werden, siehe Abbildung 2.2.

Zuletzt besprechen wir kurz wir die Funktionen zum Anlegen und Löschen von Graphobjekten, hier beispielhaft eines Knoten:

Code 2.17 (Hinzufügen und Entfernen von graph_objects).

```
node add_node() {
  node v;
  graph_obj_list<node>* node_free = get_node_free();
  graph_obj_list<node>* node_list = get_node_list();

  if ( node_free->empty() )
    { v = // get node from the thread safe Leda Memory Management }
  else
    { v = node_free->pop(); }
  node_list->push(v);
  __sync_add_and_fetch(&node_num,1);
  return v; }

void del_node(node v) {
   // delete adjacent edges
   ...
   graph_node_list* node_free = get_node_free();
   graph_node_list* node_list = get_node_list();
   __sync_sub_and_fetch(&node_num,1);
   node_list->remove(v);
   node_free->push(v); }
```

Sowohl die add_node(), als auch die delete_node() Funktionen beginnen mit der Lokalisierung der Threadlisten mit Hilfsfunktionen. Im Falle der add_node() Funktion wird das Vorhandensein von unbenutzten Knoten getestet und nur bei Bedarf der Leda Speichermanager aufgerufen. Was bleibt sind einfache Listenoperationen und das Anpassen des Knotenzählers mit einer threadsicheren Funktion.

Wir haben die im Graphen gekapselte Mengenverwaltung der Graphobjekte hinsichtlich der Funktionen Hinzufügen und Entfernen threadsicher implementiert. Die Klassen der Knoten und Kanten selbst sind nicht gesichert. Derzeit muss eine Absicherung der Knoten und Kantenverkettung und der Adjazenzlisten vom konkreten Algorithmus übernommen werden. Davon versprechen wir uns einen Leistungsvorteil, da die Synchronisation gezielt angewendet werden kann, gerade wenn die Synchronisation beispielsweise mehrere Adjazenzlisten gleichzeitig betrifft. Ein starr vorgegebener Mechanismus ist bei Synchronisierung komplexer Operationen schnell ineffizient oder erzeugt einen Flaschenhals. Außerdem ist es schwierig innerhalb der Datenstruktur die spezifischen Probleme der Synchronisationsmethoden generell aufzulösen.

Die vorgestellte Variante des Leda Graphen reicht aus, um Divide und Conquer Algorithmen zu implementieren, die sich einer einzigen Graph-Datenstruktur bedienen. Dies werden wir uns noch in Kapitel 3 zu nutze machen.

In einem weiteren Variante haben die Objektverwaltung des Leda Graphen lockfrei implementiert. Hierzu haben wir die Liste `graph_obj_list_base` lockfrei synchronisiert nach der Arbeit von H. Sundell [27], die wir in Abschnitt 2.5.2.3 bereits angesprochen haben. Der Speichermanager, der dort vorgestellten lockfreien Liste, verwaltet integral die unbenutzten Listenelemente. Der Leda Graph besitzt demnach nur noch je eine Liste für Knoten und Kanten, die alle Threads bedient. Wir gehen auf diese lockfreie Implementierung nicht weiter ein.

2.6. Häufige Operationen als parallele Objekte

Im Laufe unserer Arbeit sind wir auf immer wiederkehrende Operationen gestoßen. Diese sind die Partitionierung in zwei oder drei Teilmengen und die Suche nach Extremwerten. Aufgrund ihrer Häufigkeit bieten wir diese Operationen als parallelisierte Klassen an.

Wir nehmen an, dass das Problem jeweils durch ein Random Access Iterator Paar gegeben ist. Weiter benötigen die Module eine Relation zur Bestimmung der Partitionierung, bzw. Extremwertbestimmung. Diese Relation wird wiederum abhängig von Typ und Algorithmus in Form einer Klasse übergeben. Die parallelisierten Klassen sind so flexibel einsetzbar. Gleichzeitig garantieren wir eine Skalierung über eine beliebige Zahl von Threads.

2.6.1. Extremwert Suche

Die Suche nach Extremwerten tritt an vielen Stellen auf. In unserem Kontext geht es meist um die lexikographische Suche eines minimalen oder maximalen Punktes entlang der x-Achse. Im folgenden sehen wir einen Ausschnitt der Klassendefinition:

Code 2.18 (Definiton der Klasse `extrema`, Ausschnitt).

```
template <class iterator, class relation>
class extrema
{
  extrema(int thread_num, relation r);
```

```
  void operator()(iterator a, iterator b);

  iterator min_pos();
  iterator max_pos();
}
```

Die Klasse wird mit einer Ordnungsrelation und der Zahl der Threads initialisiert. Die parallele Operation wir mit dem ()-Operator gestartet, der als Argumente die Iteratoren der Eingabe enthält. Die Iteratoren a, b sind Random Access Iteratoren, die das Intervall $[a, b[$ beschreiben. Das Modul berechnet sowohl das Minimum, als auch das Maximum ohne Vertauschungen vorzunehmen. Das Ergebnis wird in Form von Iteratoren zurückgegeben. Das Maximum bzw. Minimum ist an den Positionen `max_pos()` bzw. `min_pos()` zu finden.

Die Klasse startet die gegeben Zahl Threads und verteilt gleichmäßig die Eingabe. Die Threads berechnen die Extrema ihrer Teilintervalle. Die Klasse fügt dann die Teilergebnisse zum Endergebnis zusammen. Die Berechnungsfunktion der Threads ist trivial:

Code 2.19 (Berechnung der Extrema in den Threads).

```
  iterator a; //Intervalanfang
  iterator b; //Intervalende
  iterator min = a;
  iterator max = a;
  relation rel; // gegebene Relation
  while (a != b)  {
    if(rel(*a,*max)) max = a;
    if(!rel(*a,*min)) min = a;
    a++;}
```

Die Ordnungsrelation wird mit dem zweielementigen ()-Operator bedient. Die Relation liefert $true$, wenn das erste Element im gegebenen Sinn größer ist als das zweite, ansonsten $false$. Das Modul Extremasuche liegt auch in einer einfacheren Variante vor, die nur das Maximum berechnet.

2.6.2. Partitionierung zweier Mengen

Die Partitionierung, also die Aufteilung einer Menge in mehrere disjunkte Teilmengen, ist essenziell für Divide und Conquer Algorithmen, die wir in Kapitel 3 diskutieren. Jede Divide Operation enthält eine mehr oder weniger komplexe Partitionierung. Eine einfache Partitionierung begegnet uns bei Mergesort. Dort wird die gegebenen Menge einfach quantitativ in zwei gleich große Hälften geteilt. Bei Quicksort hingegen findet die Partitionierung aufgrund einer Ordnungsrelation statt und ist deutlich teurer. Für diese Fälle haben wir diese Modul zur parallelen Partitionierung in zwei Mengen entworfen. Unsere Implementierung folgt der Arbeit von Tsigas und Zhang [62].

Die Eingabe der Partitionierung ist wie üblich als Feld oder in Form von Random Access Iteratoren gegeben. Anhand einer gegeben Ordnungsrelation soll die Eingabemenge

im Feld in einen linken und einen rechten Teil aufgespalten werden. Die beiden Teilmengen sind nicht notwendigerweise gleich groß. Die Partitionierung erfolgt blockweise. Die arbeitenden Threads allozieren einen freien Block fester Größe am rechten und am linken Rand des Feldes. Die Threads partitionieren die Elemente in ihren Blocks entsprechend der Ordnungsrelation in den linken bzw. rechten Block.

Die Threads alloziieren neue Blöcke nach Bedarf, sobald sie einen Block abgeschlossen haben. Da die Threads weitgehend unabhängig arbeiten, ist die Reihenfolge in der Sie Blöcke alloziieren nicht deterministisch. Können keine neuen Blöcke mehr alloziiert werden, bricht ein Thread ab. Der Algorithmus ist illustriert in Abbildung 2.3. Am Ende bleibt in der Mitte ein Rest stehen und pro Thread ist ein Block nicht vollständig aufgeräumt. Diese Reste werden sequentiell abgetragen.

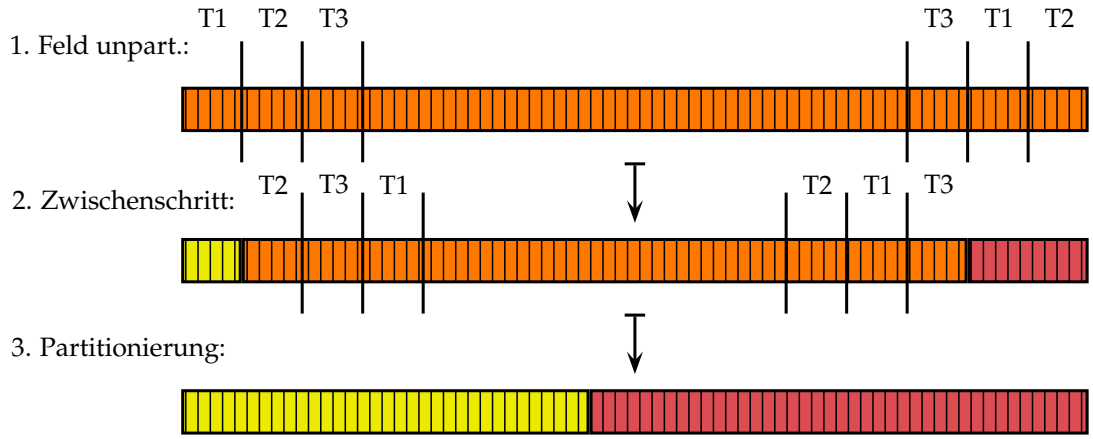


Abbildung 2.3.: Paralleles Partitioinieren in 2 Mengen ohne Hilfsfeld: Die gegeben Threads (T1,T2,T3) starten in zufälliger Reihenfolge mit der blockweisen Partitionierung (1) an den Enden beginnend. Bei Bedarf alloziieren die Threads neue Blöcke (2), der unpartitionierte Teil in der Mitte wird sukzessive verkleinert. Am Ende steht ein partitioniertes Feld(3).

Ein Vorteil dieser Methode ist, dass die Partitionierung ohne Hilfsfeld stattfindet. Zu synchronisieren ist hier die Alloziierung der Blöcke. Die Synchronisierung erfolgt lockfrei. Wir gehen wie folgt vor:

- Die Anzahl der verfügbaren Blöcke verwalten wir in einem threadsicheren Counter `blocks`, siehe 2.4.1.1.
- Die Threads dekrementieren den Counter und überprüfen, ob noch Blöcke verfügbar sind.
- Der linke und der rechte Rand des nicht alloziierten Bereich des Felds markieren zwei Iteratoren `Left_Iterator, Right_Iterator`.
- Ein Thread, der sich einen Block gesichert hat, benutzt `Fetch_And_Add`, um den linken oder rechten Iterator threadsicher um die Blockgrösse `blocksize` zu verschieben.

Der folgende Pseudocode fasst die Synchronisation schematisch zusammen.

Code 2.20 (Lockfreie Synchronisation der Partitionierung).

```
int block_available = Sub_And_Fetch(blocks,1)
if (block_available < 0)
{
  Add_And_Fetch(blocks,1)
  \\ ... Clean up
  return;
}
else
{
  if (left_block_needed)
    new_left_block = Add_And_Fetch(Left_Iterator,blocksize)
  if (right_block_needed)
    new_right_block = Sub_And_Fetch(Right_Iterator,blocksize)
}
\\ Continue Partition ...
```

Das Partitionierungsmodul ist als templatisierte Klasse implementiert. Die Klasse wird initialisiert mit der Zahl der zu verwendenden Threads, der Ordnungsrelation und der Blockgrösse. Die Relation ist problemabhängig als Klasse gegeben. Damit ist die Partitionierung sehr flexibel einsetzbar. Die Partitionierung wird mit dem zweielementigen ()-Operator gestartet. Der Operator erhält als Argument das Iteratorenpaar, das das Eingabefeld definiert. Der Rückgabewert des Operators ist der Trenner der Teilfelder.

Code 2.21 (Definiton der Klasse `partition`, Ausschnitt).

```
template <class iterator, class relation>
class partition
{
public:
  partition(relation, int thread_num, int blocksize);
  iterator operator()(iterator a, iterator b);
}
```

Die Klasse der Relation wiederum enthält einen einelementigen ()-Operator. Der Operator erhält ein Element der Eingabe und liefert $true$ oder $false$ zurück, entsprechend der Zugehörigkeit zur "kleineren" oder "größeren" Teilmenge.

2.6.3. Partitionierung dreier Mengen

Die Partitionierung in drei Teilmengen ist ungleich schwieriger als die Partitionierung in zwei Teilmengen aus dem vorherigen Abschnitt. Sie wird uns bei Triangulierungsalgorithmen und dem Quickhull Algorithmus, siehe Abschnitt 3.5.2, begegnen. Da wir die Größen der Teilmengen nicht kennen, haben wir uns im Abschnitt oben von den beiden Rändern zur Mitte gearbeitet. Aufgrund der dritten Menge entfällt hier diese Option.

In unserem ersten Ansatz teilen wir das Eingabefeld in gleichgroße Teilfelder entsprechend der Anzahl der Threads auf, siehe Abbildung 2.4 Schritt 1. Die Threads partitionieren ihre Teilfelder unabhängig voneinander (Schritt 2). Die Teilergebnisse werden wiederum parallel in zwei Schritten zusammengefügt, erst die rechte Teilmenge (Schritt 3), dann die mittlere Teilmenge (Schritt 4).

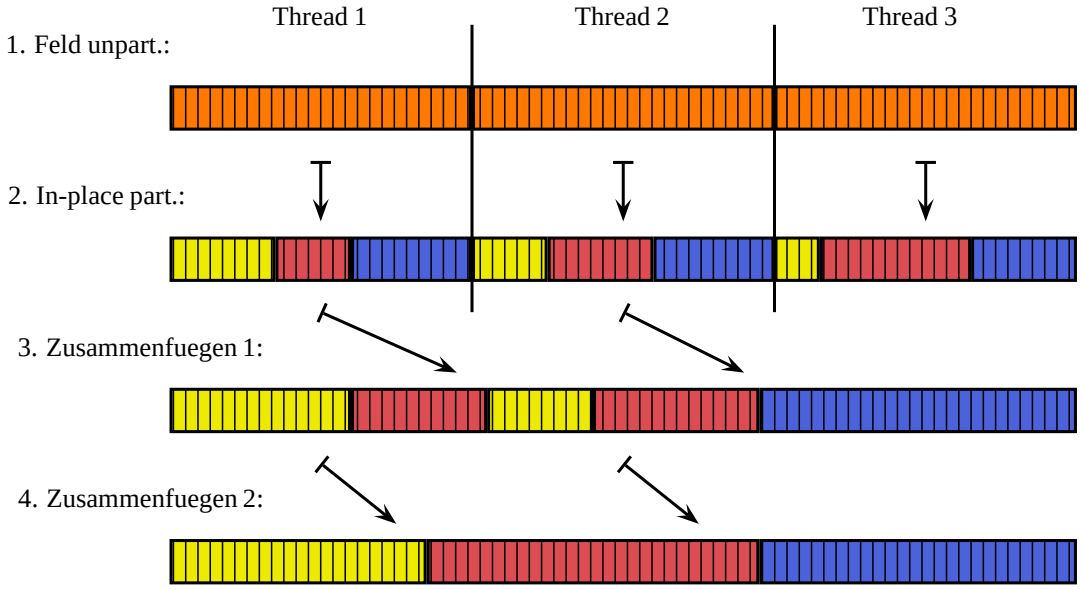


Abbildung 2.4.: Paralleles Partitioinieren in 3 Mengen ohne Hilfsfeld. Die gegeben Threads (1) partitionieren die Eingabe getrennt ohne Hilfsfeld (2). Daraus werden die Offsets berechnet, um die Partitionierung parallel in zwei Schritten zusammenzufügen (3+4).

Die Synchronisation erfolgt hier über das Starten und Aufsammeln beendeter Threads zu jedem Berechnungsschritt (2,3,4). Vor den Schritten (3) und (4) wird sequentiell die Vertauschung der partitionierten Teilmengen berechnet.

Die Partitionierung ist als templatisierte Klasse implementiert. Die Klasse wird initialisiert mit der Zahl der zu verwendenden Threads und der Ordnungsrelation. Die Partitionierung wird mit dem vierelementigen ()-Operator gestartet. Der Operator erhält als Argument erstens das Iteratorenpaar, das das Eingabefeld definiert, zweitens als Referenz die beiden Markierung `b,c`, die die Teilmengen im Ergebnis trennen.

Code 2.22 (Definiton der Klasse `partition_3`, Ausschnitt).

```
template <class iterator, class relation>
class partition_3
{
  partition_3(relation r, int thread_num);
  void operator()(iterator l, iterator r , iterator& b, iterator& c);
}
```

Die Relation wird wieder problemabhängig als Klasse gegeben. Die Klasse enthält den einelementig ()-Operator, Argument ist ein Element der Eingabe. Der Rückgabewert ist in der Menge $1, 0, -1$, je nach Mengenzugehörigkeit.

Der Vorteil dieser Methode ist, dass die Partitionierung ohne Hilfsfeld stattfindet. Dafür nehmen wir einen größeren Aufwand beim Zusammenfügen der partitionierten Abschnitte in Kauf. In unserem zweiten Ansatz haben wir stattdessen ein Hilfsfeld verwendet. So vereinfacht sich der Algorithmus deutlich. Wenn wir die Partitionierung im Hilfsfeld durchführen, können wir die Indizes im Originalfeld direkt berechnen und in einem Schritt das Ergebnis zusammenfügen, siehe Abbildung 2.5.

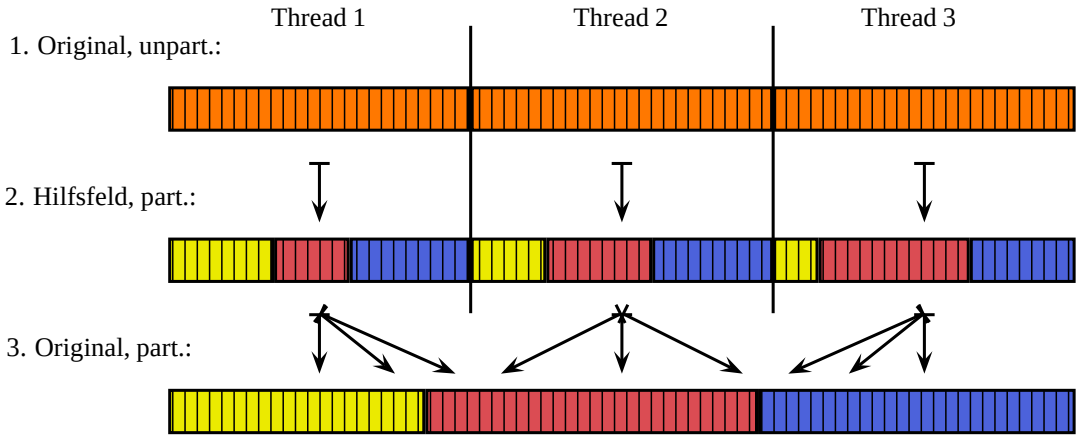


Abbildung 2.5.: Paralleles Partitioinieren in 3 Mengen mit Hilfsfeld. Die gegeben Threads (1) partitionieren die Eingabe getrennt in ein Hilfsfeld (2). Daraus werden die Offsets berechnet, um die Partitionierung parallel im Originalfeld zusammenzufügen (3).

Die Definition der Partitionierungsklasse ist ähnlich. Die Berechnung wird mit einem 5-elementigen ()-Operator gestartet. Hierbei ist `l` der Iterator auf den linken Rand des Eingabefeldes, `lh` der Iterator des Hilfsfeldes, `size` die Grösse der Felder. Die Iteratoren `b, c` sind bekannt. Die Spezifikation der Relation ist gleich der obigen.

Code 2.23 (Definiton der Klasse `partition_3h`, Ausschnitt).

```
template <class iterator, class relation>
class partition_3h
{
  partition_3h(relation r, int thread_num);
  void operator()(iterator l, iterator lh, int size,
        iterator& b, iterator& c);
}
```

Abschließend möchten wir noch hinzufügen, dass wir auch über eine Variante verfügen, die auf Listen statt Feldern basiert. In dieser Variante spielen Mengengrößen und Indizes

keine Rolle, was zu einfacherem Code führt. Aus Gründen der Effizienz bevorzugen wir jedoch immer die feldbasierten Varianten.

2.7. Offene Probleme

Zur Implementierung von threadsicheren Datenstrukturen sind viele Möglichkeiten zur Vertiefung zu finden. Dazu gehören weitere Designüberlegungen zu threadsicheren, komplexen geometrischen Datenstrukturen, wie Graphen und Bäumen oder die Weiterentwicklung lockfreier Datenstrukturen.

Auch die parallele Implementierung von Basisfunktionen ist weiterzuverfolgen. Da wir in dieser Arbeit insbesondere Divide und Conquer Algorithmen behandeln, bietet sich die Implementierung einer parallelen Merge Operation an, wie sie auch in der MCSTL Bibliothek [9] verwendet wird.

2.8. Zusammenfassung

In diesem Kapitel haben wir die verfügbaren Synchronisationsmethoden des Multithreadings vorgestellt: gegenseitiger Ausschluss und lockfreie Synchronisation. Weiter haben wir die jeweiligen Vor- und Nachteile diskutiert.

Auf Seiten des gegenseitigen Ausschlusses haben wir uns des Deadlock Problems angenommen. Wir haben verschiedenen Implementierungen von Mutex Variablen vorgestellt, die das Problem mit Hilfe der Thread Hierarchie lösen. Die Lösungen sind benutzerfreundlich und skalieren gut über die Anzahl der Threads.

Zur lockfreien Synchronisation haben wir verschiedene kleine Anwendungsbeispiele gezeigt. Die Beispiele demonstrieren die einfache Handhabung bei der Synchronisierung einfacher Variablen. Bei der systematischen Anwendung der lockfreien Synchronisation in Datenstrukturen sind wir an ihrer Komplexität gescheitert. Die Leistungsfähigkeit der vorgestellten lockfreien Deque werden wir in Abschnitt 5.3 noch genauer untersuchen.

Weiter haben wir eine Datenstruktur zur Implementierung von Mengen vorgestellt, deren Synchronisierung auf Mutex Variablen basiert. In Abschnitt 5.3 werden wir auch die Effizienz dieser Datenstruktur untersuchen. Wir haben die Anwendbarkeit dieser Datenstruktur bei der Verbesserung der Threadsicherheit des Leda Graphen demonstriert.

Die vorgestellten parallelen Objekte implementieren häufig verwendet Funktionen zur Partitionierung und Extremwertsuche. Die Objekte skalieren alle über die Anzahl der Threads sind flexibel einsetzbar. Sie kommen in beinahe allen parallelisierten Algorithmen zur Anwendung.

3. Divide and Conquer

3.1. Einleitung

In diesem Kapitel präsentieren wir ein Framework zu Parallelisierung von Divide und Conquer Algorithmen. Das Framework besteht aus templatisierten C++ Klassen und verwendet die `pthread` API [16] und verschiedene `LEDA` Klassen [6]. Das Framework wurde bereits in den beiden Veröffentlichungen "A Framework for Multi-Core Implementations of Divide and Conquer Algorithms and its Application to the Convex Hull Problem" [4] und "Multi-core Implementations of Geometric Algorithms" [5] vorgestellt.

In diesem Kapitel beschreiben wir das Design des Frameworks und zerlegen es in seine Komponenten, Abschnitt 3.3. Wir werden sowohl die Implementierung der Klassen als auch die Programierschnittstelle detailliert diskutieren. Weiter werden wir die Benutzerfreundlichkeit des Frameworks an Hand verschiedener (geometrischer) Algorithmen und seiner Anwendung auf bestehende Implementierungen zeigen. Bei der Diskussion verschiedener Codebeispiele gehen wir auf spezifische Datenstrukturen und die Threadsicherheit ein.

Im weiteren werden wir folgende Algorithmen parallelisieren:

Sortieren: Quick- und Mergsort, Abschnitt 3.4.

Konvexe Hülle: Quickhull und Gift Wrapping, Abschnitt 3.5.

Triangulierung: Einfache Triangulierung, konvexe Triangulierung und Delaunay Triangulierung, Abschnitt 3.6.

Zu den vorgestellten Implementierungen führen wir in Kapitel 5.4 ausführliche Experimente durch. Im Anhang A ist ein Auszug aus der Dokumentation des zu dieser Arbeit gehörenden Softwarepakets zu finden.

3.2. Parallelisierung von Divide und Conquer Algorithmen

Divide und Conquer Algorithmen bearbeiten Probleme durch rekursives Aufteilen in Teilprobleme, Lösen der Teilprobleme und Zusammenfügen der Teillösungen in eine Gesamtlösung. Die Teilprobleme haben jeweils die gleiche Struktur wie das Hauptproblem und können unabhängig voneinander bearbeitet werden. Durch die Unabhängigkeit der Teilprobleme ist es ein Leichtes die Lösung des Rekursionsbaums über parallel arbeitende Threads zu verteilen. Diese Parallelisierungsstrategie ist denkbar einfach und direkt. Die Schwierigkeit einer effizienten parallelen Implementierung liegt in der Ausbalancierung der Arbeit der Threads.

Unsere Framework implementiert ein Managementsystem für (Teil-) Probleme und Threads mit dem Ziel alle Kerne der CPU voll auszulasten und die höchstmögliche Beschleunigung zu erzielen.

Um die Effizienz weiter zu erhöhen gehen wir einen Schritt über den intuitiven Ansatz hinaus. Im Allgemeinen kann nicht garantiert werden, dass immer genügend Teilprobleme existieren, um alle Threads auszulasten und so optimale Geschwindigkeitsvorteile zu erzielen. Daher ist es von Vorteil, wenn auch die Teilprobleme in sich parallel gelöst werden können, wenn Threads verfügbar sind. Unser Framework wird dies unterstützen.

Eine bekannte Technik ist zudem das Lösen von Teilproblemen "naiver" Größe mit einem dritten Algorithmus. Experimentell kann sich herausstellen, dass andere Algorithmen im sequentiellen Vergleich zum Divide und Conquer Algorithmus mit einer kürzeren Laufzeit terminieren. Sind diese Algorithmen aber schwierig oder gar nicht zu parallelisieren, kann ein Divide und Conquer Algorithmus als Vehikel zur Parallelisierung und Threadbalancierung dienen. Wir werden experimentell ein Maß für die naive Problemgröße feststellen und bei einigen Algorithmen verschiedene sequentielle Ergänzungen vergleichen.

3.3. Framework Design

Das Framework setzt sich aus zwei Teilen zusammen: einer `Solver` Klasse und einer `Job` Klasse. Der Solver verwaltet den Rekursionsbaum und die Threads. Ein Job modelliert ein Divide und Conquer Problem.

Die `Jobs` repräsentieren das Problem anhand der Eingabedaten. Sie implementieren Funktionen zur eigenen Aufteilung in Teiljobs (Kinder), zur Sofortlösung trivialer Jobs und zum Zusammenfügen der Teillösungen ihrer Kinder. Ein Job stellt also situationsabhängig das Problem dar, einen Arbeitsschritt oder die Lösung. Jobs fassen den algorithmenspezifischen Teil des Frameworks zusammen. Die Definition der Jobs besprechen wir im nächsten Abschnitt.

Das `Solver` Objekt übernimmt verschiedene Verwaltungaufgaben und liegt in unterschiedlichen sequentiellen und parallelen Implementierung vor. Die sequentiellen Varianten sind sehr einfach, da sie lediglich den Rekursionsbaum ausführen, siehe Codebeispiel 3.3.2. Ein paralleler `Solver` ist anspruchsvoller. Er verantwortet unter anderem folgende Dienste:

- Er verwaltet threadsicher den Stapel unbearbeiteter Jobs.
- Er speichert den Rekursionsbaum, der zum Zusammenfügen der Teillösungen benötigt wird.
- Er startet und beendet die gegebenen Höchstanzahl Threads und teilt ihnen Arbeit zu.

Neben der Unterteilung in parallele und sequentiell Solver ist das wichtigste Unterscheidungskriterium zwischen den implementierten Solvern der unterstützte Grad der

Rekursion. Wir stellen jeweils eine Implementierung für Rekursionen vom Grad n und Optimierungen für Rekursionen von den typischen Graden 2 und 3 zur Verfügung.

Die zentrale Anforderung an die parallelen Solver ist die vollständige Auslastung der CPU, also die Erzeugung und Beschäftigung hinreichend vieler Threads. Angenommen es existieren genügend Jobs, um die Threads zu bedienen. Dann muss das Framework die Arbeitslast der Threads ausbalancieren, um Leerlauf von Threads zu vermeiden. Die Solver verwalten die offenen Jobs in einem zentralen, threadsicheren Stack an dem sich die Threads nach Bedarf bedienen.

In unserem Framework werden die Threads von einem einzigen Stack bedient. Eine ausgefeiltere Balancierungstechnik für Threads ist das Work Stealing wie es auch in der MCSTL Bibliothek [9] verwendet wird. Dort hat jeder Thread eine eigene Queue und kann bei Bedarf und nach bestimmten Regeln Arbeit eines Anderen stehlen, siehe auch [68]. Bei der Zuweisung von Jobs zu Threads, bzw. der Reihenfolge der Zuweisung und Abarbeitung spielt die Cacheeffizienz eine Rolle, siehe [69] und [70].

Durch die Verwendung eines einzelnen Stacks implementieren wir einerseits einen einfachen und robusten Balancierungsmechanismus, haben aber andererseits einen Flaschenhals geschaffen. Alle Threads, die Jobs anfragen oder abgeben, blockieren kurz den Stack. Mit der Rekursionstiefe eines Algorithmus steigt die Anzahl der Jobs quadratisch (solange der Rekursionsbaum annähernd ausbalanciert ist). Gleichzeitig werden die dargestellten Jobs immer kleiner. Im Laufe der Programmausführung würden die Threads zu viel Zeit beim verschieben trivialer Jobs verlieren und sich gegenseitig blockieren. Einen Mehraufwand durch die Anlage und Verwaltung zu vieler, zu kleiner Jobs vermeiden wir durch die Definition einer Mindestgröße für Jobs. Fällt ein Job unter das Limit, wird er nicht mehr weiter aufgespalten, sondern durch einen sequentiellen Solvers gelöst. Zu diesem Zweck haben wir zu jedem parallelen Solver einen analogen sequentiellen Solver definiert.

Für eine effiziente Auslastung des Systems durch unser Frameworks bewegen wir uns in dem Widerspruch, dass ein Job einen bestimmten - möglicherweise wachsenden - Verwaltungsaufwand unabhängig von seiner Größe verursacht, wir aber hinreichend viele Jobs brauchen, um die Threads ausbalancieren und die CPU auslasten zu können.

Die erste Frage nach der benötigten Höchstzahl an Jobs haben wir mit der Festlegung des Limit der Jobgröße experimentell beantwortet, siehe 5.4.1. Gleichzeitig haben wir gezeigt, dass unsere Balancierungsmechanismus mit einem Stack effizient arbeitet.

Die zweite Frage, nach der Mindestmenge an Jobs und Threads, ist zunächst leicht zu beantworten. Es ist unmittelbar klar, dass wir mindestens so viele Threads benötigen wie CPU Kerne auf dem System vorhanden sind und mindestens so viele Jobs wie laufende Threads. Wie sich die Threadzahl zur Anzahl der Kerne bezüglich der Effizienz verhält, untersuchen wir in vielen Experimenten.

Eine Mindestmenge an Jobs für alle Threads zu generieren ist praktisch nicht immer möglich. Die Threads, die hierzu von einem parallelen Solver gestartet und bedient werden, nennen wir primär. Wir haben zwei Fälle identifiziert in denen wir die primären Threads nicht durch die Verteilung von Jobs auslasten können:

1. Bei Teilung, bzw. Zusammenfügen des Ursprungproblems oder allgemein geringen Rekursionstiefen.

2. Bei der Lösung von degenerierten Rekursionsbäumen.

Zur besseren Auslastung der CPU durch den parallelen Solver führen wir sekundäre Threads ein. Der Solver kann den primären Threads zur Lösung eines konkreten Jobs sekundäre Threads zuordnen. Die primären Threads können die sekundären Threads während des Teilens oder Zusammenfügens eine Jobs starten, verwenden und wieder beenden. Die Anwendung sekundärer Threads, also die Parallelisierung auf Funktionsebene der Jobs, ist schwierig und erhöht die Komplexität der Implementierung deutlich. Wir werden die Effizienz sekundärer Threads in Experimenten untersuchen. Die Gesamtzahl der primären und sekundären Threads wird das vorgegebene Threadlimit nicht überschreiten.

Wir haben oben schon erwähnt, dass wir sequentielle Solver heranziehen, um Jobs geringer Größe zu lösen. Unserm Designansatz folgend, existiert für jeden parallelen Solver ein analoger Solver, der einen Job unabhängig vom Algorithmus löst. Damit stellen wir die universelle Einsetzbarkeit unseres Frameworks sicher. Gleichzeitig verfolgen wir das Ziel maximaler Leistung. Dafür kann eine strikte Modularisierung hinderlich sein. Alle Solver unseres Frameworks implementieren eine Schnittstellenklasse. Es ist ausdrücklich möglich einen spezialisierten Solver zu einem gegebenen Job zu implementieren und statt des sequentiellen Standardsolvers anzuwenden. Wir erwarten so eine weitere Steigerung der Leistung. Wie hoch diese Steigerung ausfällt muss allerdings jeweils untersucht werden.

Ein unbestreitbarer Vorteil der Divide und Conquer Algorithmen ist, dass sie durch die Unabhängigkeit ihrer Teilprobleme über eine natürliche Threadsicherheit verfügen. Die Jobs sind logisch unabhängig und verursachen keine Kollisionen zwischen Threads, solange die Jobs ihre Daten vollständig kapseln. Verweisen die Jobs jedoch auf eine gemeinsame Struktur muss diese entsprechend threadsicher implementiert werden. Beachtet man diese Einschränkung frühzeitig in der Job Definition, zunächst sequentielle implementierte Jobs unmittelbar parallel gelöst werden.

3.3.1. Divide und Conquer Jobs

Divide und Conquer Jobs formulieren ein (Teil-)Problem, das mit einem Divide und Conquer Algorithmus lösbar ist. Der Job enthält hierzu die Eingabe des Problems, meist in Form eines STL Iteratorpaares.

Unsere Betrachtung eines Jobs geht über das Vorhandensein von Eingabedaten hinaus. Ein Job implementiert alle notwendigen Funktionen, um sich verarbeiten zu lassen. Dazu gehören insbesondere die Funktionen `divide` und `merge`, aber auch Funktionen zur Erkennung und Behandlung trivialer Jobs. Die Funktionsdefinitionen von `divide` und `merge` sind dem jeweiligen `solver` bezüglich des unterstützten Rekursionsgrades angepasst. Die anderen Funktionen werden in der Definition der Job Basisklasse zusammengefasst.

Die Basisklasse `JOB` definiert folgende abstrakte Funktionen:

`int size()` Liefert die Grösse des Jobs zurück, meist die Differenz eines Iteratorpaares.

`bool is_leaf()` Gibt zurück, ob es sich um einen Job trivialer Größe handelt.

`void handle_leaf()` Löst einen trivialen Job.

Mit Hilfe der Funktion `size()` ist es dem `solver` grundsätzlich möglich flexibel auf verschiedene Jobgrößen zu reagieren, wohingegen die `leaf` Funktionen die trivialen Größen behandeln.

Im folgenden Codebeispiel sehen wir einen Ausschnitt der Mergesort Job Definition. Das unsortierte Feld ist gegeben als das Random Access Iteratorpaar `[left .. right[`.

Code 3.1 (Ausschnitt aus der Definition des Mergesort Jobs).

```
template <class iterator>
class msjob : public JOB {
  typedef msjob<iterator> job;
  iterator left;
  iterator right;
  ...
public:
  int size() { return right - left; }
  bool is_leaf() { return size() <= 3; }

  void divide(job& j1, job& j2) {
    iterator m = left + (right - left) / 2;
    j1.left = left;
    j1.right = m;
    j2.left = m;
    j2.right = right;
    ...
    return;
  }
  void merge(job& j1 , job& j2); };
```

Die Definition von `size()` ist einfach. Jobs in diesem Beispiel gelten als trivial, wenn sie aus höchsten drei unsortierten Elementen bestehen. Die Definition von `handle_leaf()` ersparen wir uns hier.

Da Mergesort Probleme typischerweise in zwei Teilprobleme aufgespalten werden, ist die Definition der `divide()` Funktion zur Benutzung eines Solvers mit Rekursionsgrad 2 ausgelegt. Der Solver erzeugt für jeden `divide` Aufruf genau zwei neue Jobs (gegeben durch $j1, j2$), gibt diese als Argumente mit und erwartet deren korrekte Initialisierung. Die Merge Funktion hat die gleiche Signatur wie Divide und implementiert das Mischen der Teillösungen. Sowohl Merge als auch Divide sind bei Vorhandensein einer sequentiellen Implementierung schnell definiert.

Durch die Bereitstellung der verschiedenen Divide und Conquer Funktionen repräsentiert ein Job neben der Eingabe auch die verschiedenen Arbeitsschritte. Und nach der Abarbeitung sogar seine Lösung.

Im oben aufgeführten Mergesort Beispiel wird die sortierte Menge, wie üblich, im Eingabefeld abgelegt. Die Form der Lösung ist hier vorgegeben.

Wir definieren keine seperate Klasse zur Repräsentation der (Teil-)lösungen. Damit wollen wir den Code übersichtlich halten und Ressourcen beim Instanziieren zusätzlicher Objekte sparen. Die Datenstruktur der Lösung kann sich von Algorithmus zu Algorithmus stark unterschieden. Wir empfehlen jedoch die Strukturen, die die (Teil-)Lösung der Jobs abbilden, physisch zu trennen. Damit ist die Threadsicherheit der Lösung sichergestellt. Greifen hingegen mehrere Jobs beispielsweise auf die gleiche Graphenstruktur zu, welche Teilgraphen nur logisch und nicht physisch trennt, benötigt der Graph ein threadsicheres Design.

Das Framework unterstützt neben der parallelen Abarbeitung der Jobs auch die Parallelisierung einzelner Jobs. Jobs werden von primären Threads behandelt, welche je nach Verfügbarkeit sekundäre Threads zur Parallelisierung der Funktionen `divide` und `merge` heranziehen können. Hierzu sind in der `JOB` Basisklasse die Funktionen

- `void set_threads(int i)`
- `int get_threads()`

implementiert. Die maximal verfügbaren Threads zu einem Job werden vom Framework gesetzt (mindestens 1). Die primären Threads können die Verfügbarkeit auslesen und die zusätzliche Threads zur Leistungssteigerung ausnutzen.

Verzichtet man auf die Verwendung von sekundären Threads und kapselt die Daten des Jobs vollständig, kann man ohne tieferes Wissen schnell einen Divide und Conquer Job definieren. Bei dieser einfachen Herangehensweise treten an keiner Stelle Mutex Variablen oder Deadlocks auf.

3.3.2. Ein Divide und Conquer `Solver`

Wir implementieren die `Solver` Klassen des Frameworks als C++ Templateklassen. Die Basisklasse jedes Solvers ist `SOLVER<Job>`. Der Solver löst Jobs entweder parallel oder sequentiell. Zu diesem Zweck spannt er den Rekursionsbaum des Divide und Conquer Algorithmus auf und verwaltet ihn. Balancierung und Threadsicherheit spielen noch keine Rolle. Im folgenden sehen wir die Implementierung eines sequentiellen Solvers:

Code 3.2 (Sequentieller Solver D&C, Klassendefinition).

```
template <class Job>
class dc_serial_solver : public SOLVER<Job> {

public:
void solve(Job& j)
{ recurse(j); }

private:
void recurse(Job& j) {
  if (j.is_leaf())
  { j.handle_leaf();
    return;
```

```
 }

list<Job> L = j.divide();
list_item it;
forall_items(it,L) recurse(L[it]);
j.merge(L); } };
```

Wir stellen verschiedene Solver zur Verfügung, die sich im Grad der unterstützten Rekursion unterscheiden. Jeder der Solver akzeptiert einen Job, der die beschriebene Job Schnittstelle und den entsprechenden Rekursionsgrad implementiert. Im Code oben sehen wir einen Solver der Jobs mit beliebigem Rekursionsgrad behandelt. Die Teiljobs werden in Listen übergeben. Im Normalfall existieren pro Job nur zwei oder drei Teiljobs. Mit an den Rekursiongrad angepassten Solvern, können wir eine exakte Anzahl von Argumenten übergeben und die Liste aus dem Beispiel wegoptimieren.

Im Beispiel oben wird die Verwaltung des Rekursionsbaumes vom Funktionsstack der CPU implizit mit übernommen und hat noch keine große Bedeutung. Iterative und parallele Implementierungen der Solver müssen hingegen eine (threadsichere) Baumstruktur verwalten, um die gelösten Teilprobleme zusammenzuführen. Die Knoten der Baumstruktur verknüpfen die Kind-Jobs mit dem Vater. Gleichzeitig wird die Anzahl der ungelösten Kinder mitgeführt und der Zustand des Knoten überwacht.

Innerhalb der Baumstruktur können sich die Jobs in drei Zuständen befinden:

unbearbeitet: Das Problem wurde noch nicht behandelt. Alle unbearbeiteten Jobs liegen auf dem Job Stack.

geteilt: Der Job wurde geteilt und hat ungelöste Kinder. Sind alle Kinder gelöst, werden die Teillösungen unmittelbar von dem Thread zusammengefügt, der das letzte Kind abgearbeitet hat. Geteilte Jobs liegen nicht auf dem Stack, sondern werden über die Baumstruktur wieder aufgegriffen.

gelöst: Gelöste Jobs repräsentieren die Lösung Ihres Problems. Sie existieren solange Ihr Vater ungelöst ist.

Auf dem Stack liegen also ausschließlich unbearbeitete Jobs.

Unbearbeitete Jobs werden von parallelen Solvern je nach Größe auf 3 verschiedenen Wegen gelöst:

- Jobs mit trivialer Größe nach Funktion `is_leaf()` werden mit `handle_leaf()` gelöst.
- Jobs deren Größe unter ein gegebenes Limit fällt, werden sequentiell durch den Standardsolver oder einen dritten Solver gelöst.
- Die restlichen Jobs werden mit Hilfe von `divide` geteilt und weiterverarbeitet.

Die Verwaltung der Jobs macht einen wesentlichen Anteil des Frameworks aus.

Die Klassendefinition eines parallelen Solvers umfasst unter anderem folgende Definitionen:

`struct jobNode` Die Struktur repräsentiert einen Knoten des Rekursionsbaums. Sie enthält einen Verweis auf einen Job und seine Kinder (wenn vorhanden) und ist mit dem Vorgängerknoten verknüpft.

`stack<jobNode*> unsolved_jobs` Der Stapel enthält die unbearbeiteten Jobs, gegeben durch die Baumknoten.

`int thread_num` Die maximale Zahl gleichzeitig eingesetzter Threads, gesetzt durch den Konstruktor.

`int limit` Das Limit für die sequentielle Lösung, gesetzt durch den Konstruktor.

`SOLVER<Job>* serial` Der Verweis auf den sequentiellen Solver, gesetzt durch den Konstruktor.

`int active_threads` Die Zahl der gerade aktiven Threads wird verwaltet.

`int work_in_pro` Die Gesamtgröße der gerade bearbeiteten Jobs wird verwaltet.

`int unfinished_leaves` Die Zahl der ungelösten Baumknoten wird verwaltet.

`void solve(Job& j)` Die `solve` Funktion startet den Solver. Der gegeben Job j ist der Wurzeljob. Die `solve` Funktion existiert auch in einer Variante, die eine Liste von Jobs akzeptiert. Der Solver kann mit mehreren Problemen gleichzeitig gestartet werden.

Das Framework startet primäre Threads automatisch. Es werden so viele primäre Threads gestartet, wie unbearbeitete Jobs vorliegen oder das Threadmaximum erreicht ist. Wenn primäre Threads arbeitslos sind, werden sie gestoppt. Falls Threadkapazitäten frei sind, werden den Jobs, die von primären Threads bearbeitet werden, sekundäre Threads zugeordnet. Die Implementierung der Job Funktionen kann die zugeteilten Threads berücksichtigen. Die Anzahl der sekundären Threads t, die einem Job j zugeteilt werden, wird anteilig berechnet. Wir setzen dazu die Gesamtgröße aller gerade in Bearbeitung befindlicher Jobs mit der Größe des Jobs j ins Verhältnis zur maximalen Anzahl der Threads und t.

Für viele Divide und Conquer Algorithmen gilt, dass einer der beiden Schritte `merge` oder `divide` trivial ist. Im Fall einer trivialen `merge` Funktion können wir den Solver modifizieren und auf die Verwaltung des Rekursionsbaumes verzichten. Ein Solver, der nur `divide` Funktionen ausführt, spart Ressourcen.

3.4. Sortieralgorithmen

Zu den einfachsten Divide und Conquer Algorithmen gehören die Sortieralgorithmen. Da geometrische Divide und Conquer Algorithmen häufig sortierte Eingaben voraussetzen, diskutieren wir zunächst die Implementierung von Jobs der bekannten Algorithmen Quicksort und Mergesort. Beide Algorithmen sind in [8] näher beschrieben.

3.4.1. Mergesort

Die Definition des Mergesort Jobs haben wir bereits in Abschnitt 3.3.1 detailliert besprochen.

3.4.2. Quicksort

Unsere Quicksort IMplementierung ist an die Leda Implementierung angelehnt. Der Quicksort Job akzeptiert als Eingabe das Iteratorpaar `[left, right[`. Die meisten Funktionen sind trivial. Der Quicksort Algorithmus sortiert die Eingabe durch Partitionierung in der Divide Funktion. Die Merge Funktion hat einen leeren Rumpf. Weiter behandeln wir einen Job als Blatt, wenn er die Grösse 1 hat und damit schon sortiert ist. Das folgende Codebeispiel ist für den Solver ohne Rekursionsgradbeschränkung ausgelegt:

Code 3.3 (Klassendeklaration des Quicksort Jobs).

```
template<class iterator> class qs_job : public JOB {
 iterator left, right;
public:
  qs_job(iterator l, iterator r): left(l),right(r){}
  int  size()    { return right - left; }
  bool is_leaf() { return size() <= 1; }
  void  handle_leaf() {}
  void  merge(list<qh_job>& children){}

  list<qh_job> divide()
    { iterator m = partition(left,right);
      list<qh_job> L;
      L.push_back(qs_job(left,m));
      L.push_back(qs_job(m + 1,right));
      return L;
    } };
```

Die Funktion divide ruft die Funktion `partition(l,r)` auf, welche ein Pivot Element wählt und die Eingabe partitioniert. Die Funktion gibt die Position m des Pivot Elements zurück, mit der wir zwei neue Kinder anlegen können. Die Implementierung der Funktion `partition(l,r)` basiert auf dem Objekt, das wir in Abschnitt 2.6.2 vorgestellt haben. Mit einer gegebenen Relation ist die Funktion leicht zu formulieren:

Code 3.4 (Parallele Partitionierung im Quicksort Jobs).

```
class relation {
  E pivot;
public:

  relation(E p) : pivot(p) {}
  bool operator()(E v)
```

```
    {return v < pivot;}
};

iterator partition(left,right) {
  // Select randomly an s from [left, right[
  relation rel(s);
  partition<iterator,relation> part(rel,get_threads(),1000);
  return part(left, right); }
```

Die Funktion `get_threads()`, die wir hier benutzen ist in der Jobs Basisklasse enthalten und gibt die Anzahl der zur Verfügung stehenden Threads an (mindestens 1). Die Relation wird mit einem zufällig gewählten Pivot Element initialisiert. Der einelementige ()-Operator liefert das Ergebnis des $<$ Vergleichs zwischen einem Element und dem Pivot Element. Der Eingabetyp muss den $<$ Operator implementieren. Für selbstdefinierte Typen existiert eine Version des Quicksort Jobs, welche die Klasse Relation als Template Argument erhält. In unseren Anwendungsfällen sortieren wir hauptsächlich Punkte und haben dafür eine angepasste Relationsklasse.

Wir haben einen Quicksort Job vorgestellt, der bei Bedarf auch parallele Partitionierungen durchführen kann. Der Quicksort Job ist durch die Anpassungsfähigkeit der Ordnungsrelation sehr flexibel.

3.5. Konvexe Hülle

Die Berechnung der konvexen Hülle ist ein Grundproblem der Computational Geometry. Durch die Vielzahl der bekannten Algorithmen mit einer tatsächlichen oder erwarteten asymptotisch Laufzeit von $O(n \log n)$ ergeben sich schon beim Betrachten dieses einen Problems viele interessante Erkenntnisse und Experimente.

Wir diskutieren zunächst den bekannten Gift Wrapping Algorithmus in einer Divide und Conquer Variante in Abschnitt 3.5.1. Danach stellen wir unseren parallelen Quickhull Code vor in Abschnitt 3.5.2. In unseren Beispielen verwenden wir immer 2-dimensionale Geometrie, der Gift Wrapping Code liegt auch in einer 3d Variante vor.

3.5.1. Gift Wrapping

Der Gift Wrapping Algorithmus konstruiert die konvexe Hülle durch das Falten einer Halbebene um die Eingabemenge der Punkte, so dass alle Punkte immer auf der gleichen Seite der Halbebene liegen. In der einfachen Variante wird die Halbebene um die komplette Punktmenge gefaltet. Der Algorithmus ist in zwei Dimensionen auch unter dem Namen Jarvis March bekannt, siehe Kapitel [8]. Die asymptotische Laufzeit liegt bei $O(nh)$, wobei h für die Grösse der Hülle steht.

In der Divide und Conquer Variante des Algorithmus von Preparata und Hong, siehe [67], vereinen wir zwei disjunkte konvexe Hüllen durch Berechnung ihrer Tangenten. Der Algorithmus läuft wie folgt ab:

Das Problem ist gegeben als lexikographisch sortierte Punktmenge. Wie beim Mergesort Algorithmus teilen wir die sortierte Eingabe des (Teil-)Problems P gleichmäßig in eine linke Hälfte L und eine rechte Hälfte R (wir orientieren uns dabei intuitiv an der x-Achse, so dass L die "kleineren" Punkte enthält). Erreichen wir einfache Problemgrößen berechnen wir die konvexe Hülle $CH(L)$, bzw. $CH(R)$ von L und R naiv. Aus der Sortierung der Punkte folgt, dass die konvexen Hüllen $CH(L)$ und $CH(R)$ disjunkt sind. Wir berechnen bzw. speichern zu jeder Hülle CH deren Extrempunkte $min(CH)$ und $max(CH)$ der Sortierung folgend. Die Hüllen $CH(L)$ und $CH(R)$ vereinigen wir durch die Berechnung der oberen und der unteren Tangente. Wir starten die Berechnung der Tangenten im Segment S mit $(max(CH(L)), min(CH(R)))$. Da die Hüllen disjunkt sind, schneidet S keine der Hüllen. Durch Iteration über die Hülle und Anwenden von Orientation Tests lassen sich die Tangenten leicht finden. Punkte, die nach der Vereinigung der Hüllen im Inneren liegen, werden entfernt.

Zu beachten ist, dass die Divide Operation des Algorithmus einfach ist. Den Hauptteil der Arbeit erledigt die Merge Operation. Die `merge` Operation wird in Abbildung 3.1 nochmal veranschaulicht.

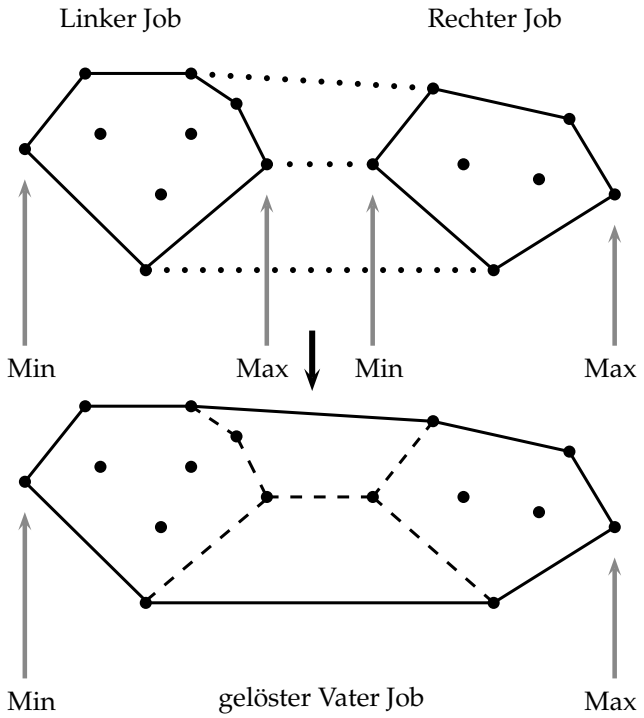


Abbildung 3.1.: Die `merge` Operation des Gift Wrapping Algorithmus zur konvexen Hülle. Oben: Das Startsegment S und die Tangenten sind gepunktet dargestellt. Unten: Die vereinigte Hülle. Einige Punkte und Kanten der Teilhüllen liegen im Inneren und entfallen.

Wir gehen einen Schritt weiter. Wir nehmen an, dass die Eingabe unsortiert ist. Wir teilen die Punktmenge daher nicht mit der oben beschriebenen, einfachen Partitionierung,

sondern benutzen die Divide Funktion des Quicksort Jobs, siehe 3.4.2. Dies bedeutet, dass wir die Eingabe durch die Zerlegung in Teilprobleme gleichzeitig sortieren und während des Zusammensetzens die konvexe Hülle berechnen. Eine ungleichmäßige Zerlegung der Jobs bedeutet allerdings auch eine Verschlechterung der asymptotischen Laufzeit von $O(n \log n)$ im *schlechtesten* Fall auf eine *erwartete* Laufzeit von $O(n \log n)$, wie das bei Quicksort der Fall ist. Diese theoretische Verschlechterung hat nur bedingte Auswirkungen in der Praxis, wie der experimentelle Vergleich zwischen Mergesort und Quicksort zeigen wird. Technisch lösen wir die Situation durch die Ableitung des Quicksort Jobs unter der Ausnutzung der Tatsache, dass die Merge Funktion von Quicksort trivial ist.

3.5.1.1. Implementierung

Die Job Definition des Gift Wrapping Jobs lautet wie folgt:

Code 3.5 (Klassendeklaration des Gift Wrapping Jobs).

```
template<class iterator> class gw_job : qs_job {
 list<point> result;
 list_iterator min, max;

public:

qs_job(iterator l, iterator r): qs_job(l,r){}
void  handle_leaf() {
  if (size() == 1) {
  result.push_back(*left);
  max = min = result.begin();}
}
void  merge(list<qh_job>& children) {
  qh_job jleft = children.front();
  qh_job jright = children.back();
  result = compute_tangents(jleft.result,
      jleft.max,jright.min,jright.result);
  min = jleft.min;
  max = jright.max; } };
```

Die Jobklasse `gw_job` erbt von Quicksort die Eingabeiteratoren `left, right` und die Funktionen `divide`, `is_leaf` und `size`. Die doppelt verkettete, zyklische Liste über Punkte `result` repräsentiert die konvexe Hülle. Benachbarte Punkte der Liste definieren ein Segment der Hülle. Die Listeniteratoren `min` und `max` markieren die Extrempunkte der Hülle. Entsprechend passen wir `handle_leaf` an, um die Liste und ihre beiden Iteratoren in den trivialen Jobs zu initialisieren.

Der komplexe Teil des Algorithmus findet in der Funktion `compute_tangents` statt. Dort werden die Listeniteratoren berechnet, die die obere und untere Tangente der Hüllen repräsentieren. Auf die Listeniteratoren werden Listenoperationen angewandt, um die

Ergebnishülle zu konstruieren. Gegebenenfalls werden an dieser Stelle die Teillisten auch verkleinert. Als Folge der Berechnung wird die Konvexe Hülle des Vaters und dessen Listeniteratoren initialisiert. Wir haben bisher keinen Ansatz gefunden diesen Merge Schritt zu parallelisieren.

3.5.1.2. Datenstrukturen und Threadsicherheit

Ein Job enthält die Iteratoren `left, right` der Eingabe, die Liste `result` mit dem Ergebnis und die Iteratoren `min` und `max`, die sich auf die Ergebnisliste beziehen.

Durch die Divide und Conquer Strategie teilt sich die Eingabemenge disjunkt auf die Jobs einer Rekursionsstufe auf. Die Iteratoren `left, right`, die eine Teilmenge repräsentieren, sind daher ebenso threadsicher, wie das Feld, das die Teilmenge enthält.

Die Liste `result` wird von jedem Job selbst angelegt und exklusiv verwaltet. Gleiches gilt für die Iteratoren `min` und `max`. Ein konkurrierender Zugriff findet hier nicht statt.

Alle Komponenten des Jobs sind threadsicher.

3.5.1.3. Weitere Eigenschaften

Indem wir die Sortierung der Punktmenge in den Job verlageren, vereinfachen wir auch die Annahmen des Algorithmus bzw. die Initialisierung des Wurzel Jobs. Eine Vorsortierung der Eingabe ist jetzt nicht mehr nötig.

Für diesen konvexe Hülle Algorithmus haben wir einige spezielle sequentielle Solver geschrieben. Wir können die Gift Wrapping Jobs, die unter das Limit fallen, sequentiell mit der Sweep Line [66], dem randomisiert inkrementellen Algorithmus [49], Quickhull [63] und [64] oder Graham Scan [65] lösen. Unser Experimente haben gezeigt, dass der Divide und Conquer Algorithmus sehr nützlich ist, um das Problem aufzusplitten und die Last zu verteilen, andere Algorithmen zur Lösung aber effizienter sind. Unsere Untersuchungsergebnisse sind in Kapitel 5.4.3 dargestellt.

Wie Eingangs erwähnt haben wir auch eine 3 dimensionale Variante des Gift Wrapping implementiert. Die 3. Dimension verkompliziert die `compute_tangents` Funktion enorm. Neue Ideen zur Parallelisierung sind darin nicht enthalten, daher sparen wir uns die Diskussion.

3.5.2. Quickhull

Auch der Quickhull Algoritmus ist aus der Literatur gut bekannt, siehe [63] und [64]. Die Idee des Quickhull Algorithmus basiert auf dem Finden von Extrempunkten der gegeben Punktmenge. Extrempunkt heißt hier extrem in einer beliebigen Richtung des Koordinatensystems. Diese Methode baut auf die Tatsache auf, dass jeder Punkt einer konvexen Hülle aus mindestens einer Perspektive extrem ist.

Der Einfachheit halber betrachten wir eine Quickhull Version, die die obere und untere konvexe Hülle getrennt berechnet. Ein (Teil-)Problem der oberen Hülle ist gegeben durch eine Segment $S = (a, b)$ dessen Endpunkte a, b Teil der konvexen Hülle sind und der Punktemenge P die oberhalb von S liegt (S ist ein Hilfsmittel, keine Begrenzung der konvexen Hülle!). Wir suchen den Punkt e in P, der senkrecht auf dem Segment S

stehend, die größte Entfernung zu S hat. Da e in der gegebenen Richtung ein Extrempunkt ist, ist er Teil der Hülle. Weiter können wir mit Hilfe von e die Punkte P in 3 Mengen partitionieren: Punkte im Dreieck a, b, e liegen im Inneren und werden aussortiert, Punkte die über den Segmenten (a, e), bzw. (e, b) liegen definieren die rekursiven Teilprobleme. Die Aufteilung der Punktmenge wird in Abbildung 3.2 dargestellt.

Der Alogrithmus konstruiert die konvexe Hülle aus Extrempunkten. Diese Extrempunkte dienen weiter als Pivot Element zur Partitionierung der Eingabemenge. Durch die zufällige Wahl des [illegible] en ungleich mächtig. Daraus folgt eine erwa [illegible] rt schon gesehen.

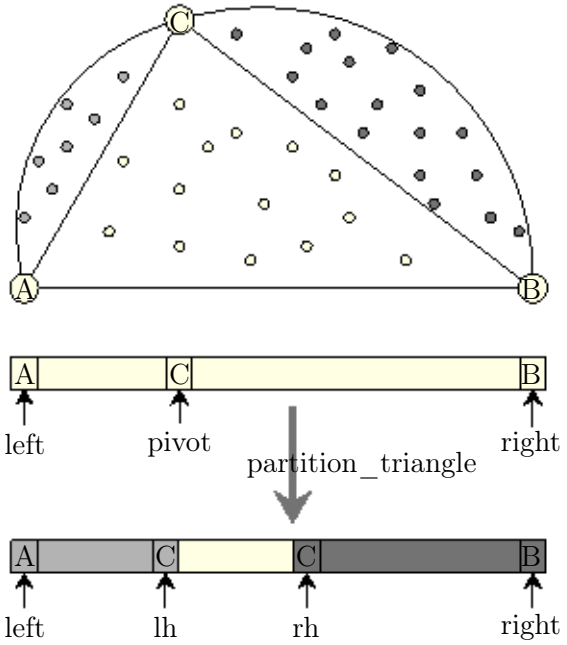


Abbildung 3.2.: Die Partitionierung durch Quickhull.

3.5.2.1. Implementierung

Wir formulieren das obere (untere) Hülle Problem als `qh_job` Klasse. Die Eingabemenge ist wieder durch ein Iteratorpaar gegeben. Die konvexe Hülle wird durch die doppelt verkettete, zyklische Liste `ch` repräsentiert. Ein Job setzt sich zusammen aus einer Punktmenge und einem Segment S, das die Basis der untern bzw. oberen Hülle markiert. Die beiden Punkte, die S definieren sind Teil der konvexen Hülle und bereits in `ch` vorhanden. S wird durch 2 benachbarte Punkte in `ch` dargestellt und durch den Listeniterator `list_it` lokalisiert.

Code 3.6 (Klassendeklaration des Quickhull Jobs).

```
class qh_job : public JOB {
public:
  iterator left, right;
```

```
  tlist<POINT>* ch;
  tlist_iterator list_it;

  int      size() { return right - left; }
  bool      is_leaf() { return size() <= 1; }
  void      handle_leaf()
    { if (size() == 1)  { ch->insert(*left,list_it);} }

  void divide(qh_job& j1, qh_job& j2) {
    pivot = extrem_point(left, right, lit)

    iterator lh, rh;
    partition_3(left,right,pivot,lh,rh);
    ch->insert(pivot,list_it);

    j1 = job(ch,left, lh,ch->succ(list_it));
    j2 = job(ch,rh, right,list_it); } };
```

Die `divide` Funktion nutzt zwei Hilfsfunktionen erstens für die Extrempunktsuche `extrem_point` und zweitens für die Partitionierung `partition_3`. Die Hilfsfunktionen sind parallelisiert. Jeder Job ermittelt durch die Extrempunktsuche einen Punkt der konvexen Hülle. Dieser wird der Hülle `ch` hinzugefügt. Nach der erfolgten Partitionierung werden zwei Kind Jobs erzeugt. Für den Quickhull Algorithmus gilt wie für Quicksort, dass die Arbeit beim Teilen der Eingabe erledigt wird. Die `merge` Funktion ist leer.

Die beiden Hilfsfunktionen benutzen die bekannten Objekte aus Abschnitt 2.6. Die Funktion `extrem_point(...)` bestimmt parallel den Punkt aus der Eingabemenge der am weitesten vom Segment S mit den Endpunkten a, b entfernt ist. Die Funktion greift auf das Objekt zur Extremasuche zurück, das wir in Abschnitt 2.6.1 vorgestellt haben. Als wesentliches Argument erhält das Objekt die Relation mit der das Maximum bestimmt wird:

Code 3.7 (Extrempunktsuche des Quickhull Alg.).

```
class max_dist {
  rat_vector vec_ab;
public:
  max_dist(POINT& a, POINT& b) : vec_ab(b - a) {};

  bool operator()(POINT& test,POINT& max)
    {
      POINT help = max + vec_ab;
      int orient = orientation(max,help,test);
      return (orient == 1 || orient == 0 && compare(test,max) == 1);
    } };
```

Die Relation testet gegen den Vektor $vec_a b = (b - a)$. Mit diesem Vektor und einem angenommenen Maximum max lässt sich ein Segment $max, max + vec_a b$ bestimmen, das

parallel zu S liegt. Mit einem Orientation Test können wir feststellen ob der zu testende Punkt zwischen den beiden Segmenten oder weiter entfernt von S liegt und somit das neue Maximum darstellt.

Die zweite Hilfsfunktion `partition_3(...)` partitioniert die Eingabemenge der Punkte anhand des Dreieck $a, pivot, b$. Es gibt drei Ergebnismengen: Punkte innerhalb des Dreiecks, die für die konvexe Hülle Berechnung irrelevant sind, Punkte jenseits des Segments $a, pivot$ und solche jenseits von $pivot, b$. Aus Letzteren konstruieren wir die rekursiven Teilprobleme. Auch hier nutzen wir ein paralleles Objekt, welches wir in 2.6.3 eingeführt haben. Die Ordnungsrelation für das Objekt ist gegeben durch die Klasse:

Code 3.8 (Partitionierung des Quickhull Alg.).

```
class part_triangle {
  POINT a,b,pivot;
public:
  part_triangle(POINT& a_, POINT& pivot_, POINT& b_)
      : a(a_), pivot(pivot_), b(b_){};

  int operator()(const POINT test) {
       if (orientation(a,pivot,test) == 1)
         return 1;
       if (orientation(pivot,b,test) == 1)
         return -1;
       return 0;
    } };
```

Die Relation enthält die drei Punkte des Dreiecks. Durch zwei Orientation Tests wird die Lage der Eingabepunkte zum Dreieck ermittelt.

Die Wirkung des parallelen Teilens werden wir in Abschnitt 5.4.3.4 experimentell untersuchen.

3.5.2.2. Datenstrukturen und Threadsicherheit

Wie schon gesehen, beschrieben die Iteratoren `left, right` die Eingabe des Jobs, die Liste `ch` repräsentiert die konvexe Hülle und der Listiterator `list_it` markiert ein Element der Hülle.

Die Iteratoren `left, right` ist threadsicher, siehe 3.5.1.2.

Die Liste `ch` wird von allen Jobs gemeinsam, d.h. von mehreren Threads konkurrierend, konstruiert. Die Liste `ch` muss threadsicher sein. Im konkreten Fall sind die Anforderungen allerdings nicht hoch. Die Liste muss eine threadsichere `insert` Operation an einer gegebenen Position (`list_it`) unterstützen. Typische Operationen wie `push, pop` und auch `delete` benötigen wir nicht.

Wir designen den Listenkontainer entsprechend den Überlegungen aus Abschnitt 2.5.1. D.h. wir reduzieren die Listenvariablen auf ein Zeigerpaar für Anfang und Ende der Liste und ggf. einen threadsicheren Counter für die Listenlänge, siehe 2.4.1.1. Listenanfang und

-ende werden sequentiell initialisiert und im Bedarfsfall von der threadsicheren `insert` Operation angepasst.

Ein Listenelement besteht aus den beiden Zeigern der doppelten Verkettung und einem Wert. Der Wert, der hier einen Ecke der konvexen Hülle repräsentiert ist unveränderlich und daher threadsicher. Die Verkettung der Liste hingegen wird durch die Insert Operation ständig angepasst. Die Threadsicherheit wird auch hier durch die Divide und Conquer Strategie garantiert. Zwar wird jedes Listenelement (jede Ecke) in zwei Jobs referenziert, doch sind einem Job durch den `list_it` Iterator implizit zwei benachbarte Ecken A, B eindeutig zugeordnet. Die `Insert` Operation fügt ein Element genau zwischen A und B ein und ändert nur die Verzeigerung zwischen A und B. Diese Zeiger sind dem Thread eindeutig zugeordnet. Die betroffenen Zeiger werden also nur von einem Thread gelesen und geschrieben. Zur Sicherung des Containers sind keine Mutex Variablen nötig.

Alle verwendeten Datenstrukturen des Jobs sind threadsicher.

3.5.2.3. Weitere Eigenschaften

Die Initialisierung der ersten beiden Jobs (obere/untere Hülle) erzeugt die Liste `ch` aus zwei Extrempunkten, die das erste Basissegment bilden. Die Eingabe wird in eine darüber und eine darunter liegende Punktmenge partitioniert und zwei Jobs angelegt. Da unser Framework eine Liste von Jobs abarbeitet ist ein gleichzeitiges Lösen der oberen und unteren Hülle kein Problem. Außerdem haben wir die Initialisierung selbst wieder parallelisiert. Zur Berechnung der Extrempunkte für die Liste `ch`, sowie die erste Partitionierung in zwei Mengen, greifen wir wieder auf die bekannten, parallelen Objekten zurück, siehe Abschnitt 2.6.

Versuchsweise steht auch eine Quickhull Implementierung zur Verfügung, die die Eingabe als Liste akzeptiert. Das Arbeiten mit Listen vereinfacht den Code der Partitionierung, siehe Kapitel Experimente 5.4.3.3.

3.6. Triangulierungen

Wir werden drei Triangulierungsalgorithmen besprechen.

Zunächst beschäftigen wir uns mit einem naiven Algorithmus, der Punkte in ein umgebendes Dreieck einfügt. Die neu entstandenen Dreiecke zerteilen die Menge der restlichen Punkte.

Danach werden wir den Gift Wrapping Algorithmus der konvexen Hülle erst zu einem Triangulierungsalgorithmus erweitern und in einer weiteren Stufe zur Delaunay Triangulierung.

Die Triangulierungsalgorithmen haben wir jeweils mit mindestens zwei verschiedenen Datenstrukturen implementiert. Einmal mit einer spezialisierten, dem Algorithmus angepassten Struktur und einmal mit einer unserer Variante des Leda Graphen aus Abschnitt 2.5.3.

3.6.1. Unterteilung eines Dreiecks

Der naive Algorithmus arbeitet von außen nach innen. Startpunkt ist eine gegebene Punktmenge, um die wir ein Dreieck legen. Aus der Punktmenge wählen wir zufällig einen Punkt P. P ist das Pivotelement, mit dem wir das umgebende Dreieck in drei Teildreiecke und die enthaltene Punktmenge in drei Teilmengen unterteilen. Die drei Teilmengen in den kleineren Dreiecken bilden die rekursiven Teilprobleme. Der Algorithmus wird in Abbildung 3.3 illustriert.

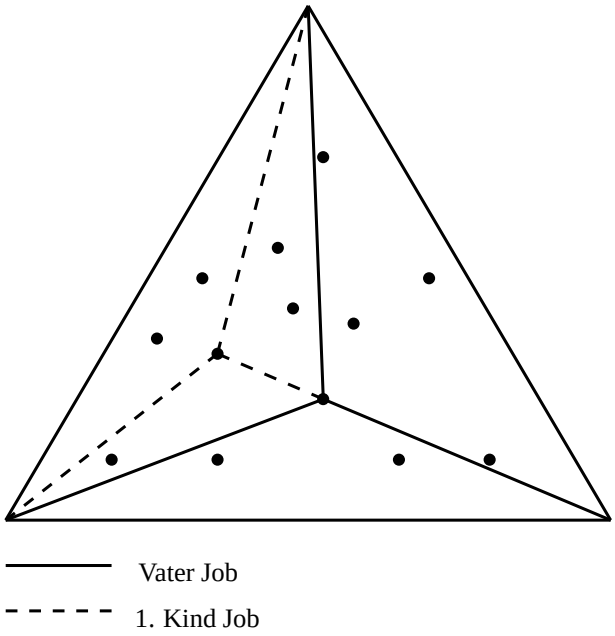


Abbildung 3.3.: Ein einfacher Triamgulierungs Algorithmus. Die Punkte liegen innerhalb eines Dreiecks. Ein zufällig gewählter Punkt definiert drei weitere Dreiecke (durchgehende Linie) eine unterteilte Punktmenge wird rekursiv weiterverarbeitet (gestrichelte Linie).

In jedem Rekursionsschritt verringern wir die Eingabemenge um ein Element. Der Algorithmus wird in jedem Fall terminieren. Durch die zufällige Wahl des Pivot Elements erreichen wir Teilprobleme unterschiedlicher Grösse und einen unbalancierten Rekursionsbaum. Die erwartet Laufzeit des Algorithmus liegt daher bei $O(n \log n)$.

Wir haben diesen Algorithmus mit drei unterschiedlichen Datenstrukturen implementiert. Im folgenden werden wir drei Job Definitionen vorstellen:

- Einen Job basierend auf dem LEDA Graph.
- Einen Job mit einer kantenbasierten Datenstruktur.
- Einen Job mit einer dreiecksbasierten Datenstruktur.

3.6.1.1. Implementierung mit Hilfe des LEDA Graphen

In Abschnitt 2.5.3 haben wir Eigenschaften und Threadsicherheit des LEDA Graphen diskutiert. Ein Zeiger auf eine Instanz des parametrisierten LEDA Graphen ist Teil der Job Definition. Alle Jobs teilen sich die gleiche Graphinstanz und damit einen gemeinsamen Speicherbereich. Damit sind wir in der Lage die Triangulierung zu berechnen ohne teuer Graphstrukturen zu vereinen, müssen dafür aber die Threadsicherheit feststellen. Im Folgenden sehen wir einen Ausschnitt der Job Deklaration:

Code 3.9 (Ausschnitt Job Deklaration, Triangulierung mit LEDA Graph).

```
template <class iterator>
class triang_ledagraph : public MC_JOB {
  iterator left, right;
  GRAPH<POINT,int>* G;
  edge bc;

public:
  int size() {return right - left;}
  bool is_leaf() {return size() <= 1;}
  ... }
```

Ein Job Definition enthält außer des Verweises auf den LEDA Graphen G die Punktemenge als Iteratorenpaar $left, right$ und eine Kante bc, die das Dreieck identifiziert. Der Graph G ist eine ordnungserhaltende Map. Die gerichtete Kante bc bezieht sich auf das links liegende Dreieck.

In das durch bc definierte Dreieck von G wird ein Punkt mit Hilfe der Funktion `insert_node_in_map` eingefügt. Diese benutzen wir unter anderem in der `handle_leaf` Funktion:

Code 3.10 (Einfügen eines Punkts, Triangulierung mit LEDA Graph).

```
void      handle_leaf()
{
  if (size() == 0) return;

  node n = G->new_node(*left);
  edge ab = G->face_cycle_pred(bc);
  edge ca = G->face_cycle_succ(bc);
  insert_node_in_map(n,ab,bc,ca)
}
```

Im oben gezeigten Code fügen wir zuerst einen neuen Knoten n in den Graphen G ein. Geometrisch liegt n in der Facette von G, die durch bc bestimmt wird. Die Facette ist ein Dreieck, das aus den Kanten ab, bc, ca besteht. Die Kanten ab und ca sind Vorgänger bzw. Nachfolger von bc in der Facette und können mit der entsprechenden Funktion des

Graphen ermittelt werden. Zuletzt müssen die Kanten zwischen n und dem umgebenden Dreieck eingefügt werden, so dass die Eigenschaften von G erhalten bleiben. Die Hilfsfunktion hierzu `insert_node_in_map` basiert auf einfachen Graphfunktionen und wird von uns daher nicht weiter diskutiert.

Zusätzlich zur Einfügeoperation in den LEDA Graphen haben wir in der `divide` Funktionen den Partitionierungsschritt. Die Struktur des Funktionsrumpfes und die Partitionierungsmethode, die wir verwenden sind schon aus Abschnitt 3.5.2.1 bekannt:

Code 3.11 (Partitionierung, Triangulierung mit LEDA Graph).

```
void divide(job& Aj, job& Bj, job& Cj) {
    Aj.G = Bj.G = Cj.G = G;

    node n = G->new_node(*left);
    edge ab = G->face_cycle_pred(bc);
    edge ca = G->face_cycle_succ(bc);

    insert_node_in_map(n,ab,bc,ca);
    set_bc_in_children(A,B,C,n);

    relation rel(G->inf(G->source(ab)),G->inf(G->source(bc)),
        G->inf(G->source(ca)),G->inf(G->source(n)));
    partition_3<iterator,relation> part(rel,get_threads());
    iterator b,c;
    left = left + 1;
    part(left,right,b,c);

    set_intervals_in_children(A,B,C,b,c) }
```

Außer der Partitionierung verwenden wir die Hilfsfunktionen `set_bc_in_children` und `set_intervals_in_children`. Erstere initialisiert die Variable bc der Kindjobs, letztere die Variablen $left$ und $right$. Übrig bleibt die Ordnungsrelation der Partitionierung:

Code 3.12 (Ordungsrelation der Triangulierung).

```
class relation {
public:
  relation(POINT a_, POINT b_, POINT c_, POINT m_) :
    a(a_), b(b_), c(c_), m(m_) {}

  int operator()(POINT p) {
    int o1 = orientation(a,m,p);
    if (o1 == -1)
      return (orientation(b,m,p) == 1) ? 1 : 0;
    else
```

```
      return (orientation(c,m,p) == -1) ? -1 : 0;
    }
private:
  POINT a,b,c,m; };
```

Zu partitionieren ist eine Punktmenge, die innerhalb des Dreiecks a, b, c liegt. Das Dreieck wird durch den Punkt n in drei Teile unterteilt. Die Teildreiecke definieren die Teilmengen der Punkte. Die Relation wird initialisiert mit den Punkten a, b, c und m. Der zu testende Punkt p wird mit zwei Orientation Tests in einem Teildreieck lokalisiert. Der Rückgabewert liegt in der Menge $-1, 0, 1$.

Die Threadsicherheit wird durch die Struktur des Divide und Conquer Algorithmus und den threadsicheren LEDA Graphen gewährleistet. Der Algorithmus garantiert, dass ein Dreieck von nur einem Thread bearbeitet wird. Ein Thread unterteilt ein Dreieck durch Hinzufügen von Knoten und Kanten. Die Erzeugung und Verwaltung der Knoten und Kanten ist threadsicher, siehe 2.5.3. Außerdem werden die Adjazenzlisten vorhandener Knoten durch die Funktion `insert_node_in_map` erweitert. Da die Knoten, Ecken mehrerer Dreiecke repräsentieren, die parallel bearbeitet werden können, müssen wir uns diese Funktion im Detail anschauen:

Code 3.13 (Einfügen von Knoten und Kanten in die Map)**.**

```
insert_node_in_map(node n, edge ab, edge bc, edge ca) {
  edge na = G->new_edge(n,ca);
  edge nb = G->new_edge(n,ab);
  edge nc = G->new_edge(n,bc);

  edge cn = G->new_edge(ca,n);
  edge bn = G->new_edge(bc,n);
  edge an = G->new_edge(ab,n);

  G->set_reversal(az,za);
  G->set_reversal(bz,zb);
  G->set_reversal(cz,zc); }
```

Die Graphfunktion `new_edge(edge e, node n)` erzeugt eine Kante e' vom Anfangsknoten v von e zum Knoten n und fügt e' hinter e in die Adjazenzliste von v ein. Die Funktion ändert dabei die paarweisen Listenzeiger zwischen e und dessen Nachfolger.

Im konkreten Fall ist ein Dreieck gegeben durch die Kanten ab, bc, ca bzw. durch die Gegenkanten ac, cb, ba und die Knoten a, b, c. Da der Graph ein ordnungserhaltende Map ist, sind beispielsweise die Kanten ac und ab in der Adjazenzliste von Knoten a benachbart. Der Aufruf `new_edge(ab,n)` erzeugt eine Kante vom Knoten a zum Knoten n. Die Kante an wird zwischen ac und ab in die Adjazenzliste eingefügt. Da uns die Implementierung der LEDA Funktion `new_edge(edge e, node n)` zur Verfügung steht, wissen wir, dass die Funktion nur die Verzeigerung verändert, die die Kanten ab, bc, ca und deren Gegenkanten verknüpft. Benachbarte Dreiecke sind nicht betroffen.

Die Abarbeitung dieses Jobs erfolgt threadsicher.

3.6.1.2. Kantenbasierte Datenstruktur

Wir implementieren den einfachen Triangulierungsalgorithmus mit einer spezialisierten, handlichen Datenstruktur. Mit spezialisierten Strukturen können wir im Allgemeinen die Threadsicherheit leichter sicherstellen.

Die kantenbasierte Datenstruktur stellt einen einfachen, gerichteten Kantentyp zur Verfügung, der ordnungserhaltende Maps unterstützt. Es wird kein Typ für Knoten definiert, diese sind in den Kanten implizit enthalten.

Die Kantendefinition besteht aus dem Start- und Endpunkt der Kante und fünf Zeigern auf weitere Kanten. Ein Zeigerpaar dient der doppelten Verkettung in der Adjazenzliste und ein weiteres Kantenpaar zur doppelten Verkettung in einer globalen Kantenliste. Der letzte Kantenzeiger verweist auf die Gegenkante.

Weiter enthält die Kantendefinition Hilfsfunktionen zur Anlage und Verknüpfung von Kanten. Die Klasse ist hinsichtlich des Punkttyps templatisiert.

Die Definition des Jobs enthält wie oben eine Kante, die das Dreieck identifiziert, und analog eine Liste aller verwendeter Kanten im Job. Der Ablauf des Algorithmus folgt dem im vorigen Abschnitt Beschriebenen. Die wichtige Ordnungsrealtion zur parallelen Partitionierung ist identisch, die divide Funktion folgt dem bekannten Muster. Lediglich die Funktionsaufrufe zur Konstruktion der Kanten und Listen müssen angepasst werden. Wir sparen uns daher Codebeispiele. Wie oben wird die Threadsicherheit dadurch garantiert, dass Threads nur auf jene Zeiger der Kanten zugreifen, die Ihr Dreieck betreffen. Wir schließen einen wechselseitigen Zugriff per Konstruktion aus.

3.6.1.3. Dreiecksbasierte Datenstruktur

Wir haben eine weitere, alternative Datenstruktur zur Implementierung des Triangulierg-salgorithmus entwickelt. Die Struktur basiert auf Dreiecken. Ein Dreieck wird mit seinen drei geometrischen Nachbarn der Triangulierung zu einem dualen Graphen verknüpfen.

Ein Dreiecksobjekt besteht aus drei Punkten, drei Zeigern zu den benachbarten Dreiecken und einem Zeiger, um zur Speicherverwaltung einen Stack aus allen Dreiecken zu bilden. Mit dieser simplen Definition konstruieren wir den dualen Graphen der Delaunay Triangulierung. Die Navigation durch diesen Graphen ist umständlicher, da in den Dreiecken, im Gegensatz zu den greichtenten Kanten keine Orientierung kodiert ist und diese in jeder Situation erst berechnet werden muss.

Ein Job wird definiert durch ein Dreieck und die darin liegenden Punkte. Die Partitionierung der Punktmenge erfolgt wie gehabt. Allein beim Zusammensetzen der einzelnen Dreiecke gibt es eine entscheidende Anpassung. Die Punktmenge wird in der `divide` Funktion geteilt. Die Dreiecke werden erst in der Wurzel der Rekursion angelegt, da zuvor nur Zwischenergebnisse bekannt sind. Erst im `Merge` Schritt erfolgt die Verkettung eines Dreiecks mit den benachbarten Dreiecken. Weiter Erkenntnisse zur Parallelisierung gibt es hier nicht, wir sparen uns Codebeispiele.

3.6.2. Gift Wrapping

Der Gift Wrapping Algorithmus zur Triangulierung basiert auf dem gleichnamigen Algorithmus zur Berechnung der konvexen Hülle, siehe Abschnitt 3.5.1, und ist eine konsequente Weiterentwicklung.

Die gegebene Punktmenge wird, wie in Abschnitt 3.5.1 beschrieben, sortiert und in gleich große Teile unterteilt. Zwei konvexe Triangulierungen werden vereinigt indem die Tangenten der Hüllen berechnet werden. Der Zwischenraum zwischen Tangenten und Hüllen wird trianguliert. Die Triangulierung erfolgt mit den Zwischenergebnissen der Tangentenberechnung, so dass der Zusatzaufwand überschaubar ist. Der Algorithmus ist in Abbildung 3.4 illustriert.

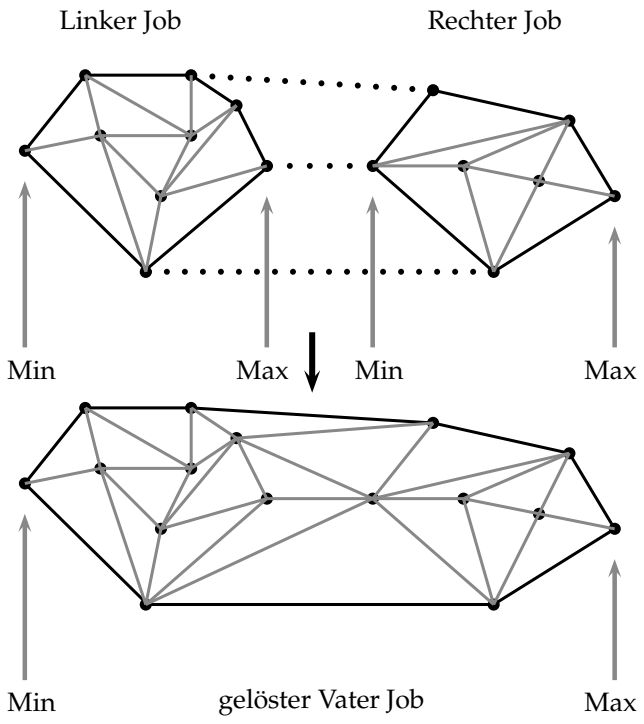


Abbildung 3.4.: Die `merge` Operation des Gift Wrapping Algorithmus zur Triangulierung. Oben: Das Startsegment S und die Tangenten sind gepunktet dargestellt. Unten: Die vereinigte Triangulierung. Der Bereich zwischen den Tangenten wurde trianguliert.

Wir haben den Algorithmus mit zwei verschiedenen Datenstrukturen implemenetiert. Zum einen mit dem threadsichere LEDA Graph und zum anderen mit der kantenbasierten Datenstruktur. Beide Datenstrukturen sind aus dem vorhergehenden Abschnitt bekannt. Durch die Verwendung dieser beiden Datenstrukturen im bekannten Giftwrapping Algorithmus ergeben sich natürlich Änderungen bei der Implementierung. Keine der Änderungen betrifft direkt die Parallelisierung oder die Threadsicherheit.

3.6.3. Delaunay Triangulierung

Auch die Delaunay Triangulierung folgt dem bekannten Schema. Der Algorithmus ist bekannt durch Guibas und Stolfi [72]. Aufstellungen von Delaunay Algorithmen sind bei Steven Fortune [50] und Peter Su [51] zu finden.

Bei diesem Divide und Conquer Algorithmus wird die Eingabemenge sortiert und gleichmäßig in immer kleiner Teilprobleme partitioniert. Ein gelöstes Teilproblem ist eine konvexe Delaunay Triangulierung. Auch die Vereinigung folgt dem bekannten Muster. Ausgehend von zwei bekannten Punkten auf den beiden Hüllen werden die Tangenten zwischen den Hüllen berechnet. Der Zwischenraum wird trianguliert. Neu ist hier, dass die neu entstandenen Dreiecke hinsichtlich der Delaunay Eigenschaft überprüft und ggf. korrigiert werden müssen. Der Algorithmus ist illustriert in Bild 3.5.

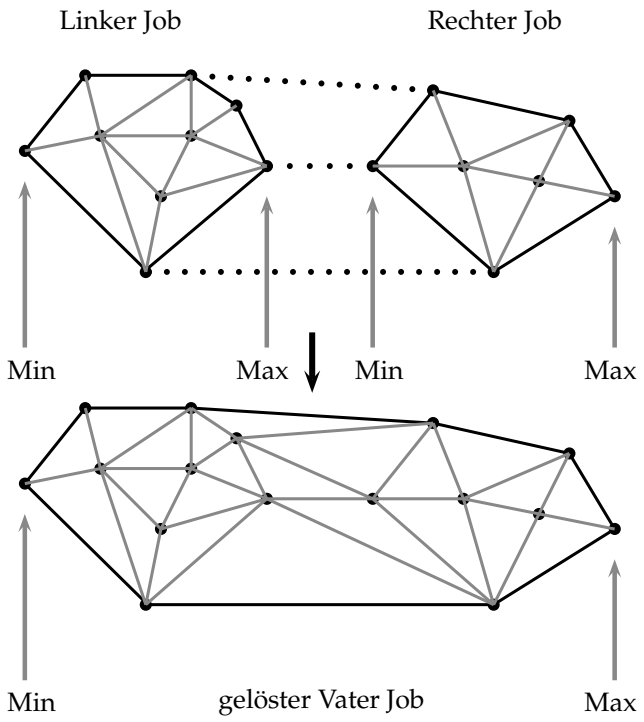


Abbildung 3.5.: Die `merge` Operation des Gift Wrapping Algorithmus zur Delaunay Triangulierung. Oben: Das Startsegemnt S und die Tangenten sind markiert. Unten: Die vereinigte Delaunay Triangulierung. Der Bereich zwischen den Tangenten wurde trianguliert, vorhandene Kanten zur Erfüllung des Delaunay Kriteriums korrigiert.

Wir haben den Algorithmus für den threadsicheren LEDA Graphen und die kantenbasierte Datenstruktur implementiert. Die Erweiterungen zur gewöhnlichen Triangulierung betreffen im wesentlichen den sequentiellen Codeabschnitt innerhalb der Merge Funktion, Parallelisierungsschritte sind nicht direkt betroffen. Wir werden auf weitere Erläuterungen der Implementierung verzichten.

3.7. Offene Punkte

Sowohl im Bereich der Solver Klassen, als auch bei parallelisierten Algorithmen, gibt es viele mögliche Anknüpfungspunkte zur weiteren Vertiefung.

Ein Ansatz zur Weiterentwicklung der Solver Klassen bietet sich beim Problem der Ausbalancierung der Threadbelastung. In den Experimenten hat sich zeigen, dass die Auslastung der Threads mit der jetzigen Implementierung hoch ist. Wenn die Zahl der Prozessorkerne weiter steigt und sich mehr Threads die vorhandene Arbeit aufteilen, kann es sich lohnen die Balancierung erneut zu überprüfen. Beispielsweise ist es denkbar den Work Stealing Algorithmus von Blumofe und Leiserson [68] zu verwenden, der auch schon in der MCSTL Bibliothek [9] zu Einsatz gekommen ist. Das Problem der Ausbalancierung von Threads betrifft nicht nur die primären, sondern auch die sekundären Threads. Denn auch die Zuteilung von sekundären Hilfsthreads funktioniert derzeit nach einem einfachen Schema. Beim Design des Verteilungsmechanismus von Jobs kann auch über Cache Effizienz nachgedacht werden, siehe [69].

Zu einem Grundproblem der Computational Geometry haben wir keine Implementierung vorgenommen: das Finden des engsten Punktpaares in einer Punktmenge. Ein Divide und Conquer Algorithmus ist bei T. Cormen et al. [8], Kapitel 33.4, Seite 957 beschrieben. Ein weiteres bekanntes Problem ist der Schnitt von Segmentmengen. Hierfür hat I. Balaban einen Divide und Conquer Algorithmus entworfen, siehe [71]. Mit der Parallelisierung weiterer Algorithmen geht der Entwurf von threadsicheren Datenstrukturen einher.

3.8. Zusammenfassung

Wir haben ein Framework zur Implementierung und Parallelisierung von Divide und Conquer Algorithmen vorgestellt. Das Framework ist durch den Einsatz von C++ Templates generisch und durch seine Modularisierung in Solver und Job leicht erweiterbar.

Die parallelen Solver maskiert die Verwaltung der Threads und balanciert effizient die Threads aus. Die Skalierbarkeit der Threads ist hoch, da nicht nur der Rekursionsbaum, sondern auch einzelnen Operationen parallelisiert werden können.

Sequentielle Implementierung lassen sich leicht in ein Job Definition einfügen oder als sequentieller Solver zur Effizienzsteigerung verwenden. Die Jobs lassen sich durch das Klassenkonzept wiederverwenden.

Die einfache Anwendung des Frameworks haben wir durch die Implementierung der Sortieralgorithmen Quicksort und Mergesort, durch die konvexe Hülle Algorithmen Gift Wrapping und Quickhull und durch verschiedene Traingulierungsalgorithmen demonstriert.

4. Randomisiert inkrementelle Konstruktion

4.1. Einleitung

In diesem Kapitel präsentieren wir ein Framework zu Parallelisierung von inkrementell randomisierten Algorithmen. Das Framework besteht aus templatisierten C++ Klassen und verwendet die `pthread` API [16] und verschiedene `LEDA` Klassen [6].

Wir beschreiben das Design des Frameworks 4.3 und zerlegen es in seine Komponenten. Diese sind zum einen `Solver` Klassen, deren Implementierung wir diskutieren, und zum anderen die Programmierschnittstelle 4.3.1, die durch ein Interface einer Datenstruktur gegeben ist.

Im weiteren Verlauf werden wir Effizienz und Benutzerfreundlichkeit des Frameworks anhand verschiedener geometrischer Algorithmen zeigen: der konvexen Hülle 4.4 und der Delaunay Triangulierung 4.5. Unsere Implementierungen basieren auf einer hohen Anzahl konkurrierender Zugriffe auf eine gemeinsame Datenstrukturen. Wir gehen auf die Threadsicherheit sowie die Deadlock Vermeidung 4.4.2 ein. Zu den vorgestellten Implementierungen führen wir in Kapitel 5.5 Experimente durch. Im Anhang A ist ein Auszug aus der Dokumentation des zu dieser Arbeit gehörenden Softwarepakets zu finden.

4.2. Randomisiert Inkrementelle Algorithmen

Die randomisiert inkrementelle Konstruktion von Datenstrukturen ist eine Standardtechnik in der Computational Geometry. Das Prinzip beruht auf der Idee, dass Elemente einzeln, unabhängig voneinander und in einer zufälligen Reihenfolge in eine geometrische Datenstruktur eingefügt werden. Die Randomisierung kann dabei die erwartete Anzahl an Berechnungsschritten pro Einfüge Operation reduzieren. Der Gesamtalgorithmus wird mit einer erwarteten Laufzeit angegeben.

Zu vielen geometrischen Problem sind effiziente randomisiert inkrementelle Algorithmen bekannt, z.B. zur konvexe Hülle, der Delaunay Triangulierung oder dem Linienschnitt.

Der erste systematische Ansatz die Randomisierung in der Computational Geometry einzusetzen geht auf K. L. Clarkson ([47], [48], [49]) zurück. Aus dieser Quelle stammt letztlich auch der inkrementelle konvexe Hülle Algorithmus, den wir in dieser Arbeit parallelisieren. R. Seidel hat in [46] die Laufzeit einer ganzen Reihe von randomisiert inkrementellen Algorithmen analysiert und zu diesen erwartete Laufzeiten ermittelt.

Eine andere, weit verbreitete Form inkrementeller Algorithmen ist der Sweep Line Algorithmus. Der Sweep Line Algorithmen arbeiten die Eingabe in der Reihenfolge einer

(geometrischen) Sortierung ab. Dadurch ist es leichter möglich vereinfachende Annahmen zu formulieren und Behelfsdatenstrukturen zu verwalten. Die Sweep Line Technik ist daher weit verbreitet und kann bei beinahe jeder Problemstellung der Computational Geometry angewendet werden. Da die geometrische Sortierung jedoch die Reihenfolge der Abarbeitung jedes einzelnen Eingabeelements deterministisch vorschreibt, ist ein Sweep Line Algorithmus zwar inkrementell, aber auch inhärent sequentiell. In dieser Arbeit beschäftigen wir uns nicht mit Sweep Line Algorithmen.

4.3. Framework Design

Das Framework besteht aus zwei Komponenten: Zum Einen aus einem Container, der eine Struktur enthält, die durch den inkrementellen Algorithmus konstruiert wird. Zum anderen aus einem `Solver`, der die (parallele) Iteration implementiert.

Der Container verwaltet die Struktur und stellt vordeklarierte Funktionen zur Verfügung. Die Funktionen dienen zur inkrementellen Konstruktion der Datenstruktur und sind nach Bedarf threadsicher. Der Container ist der algorithmenspezifische Teil des Frameworks. Wir nennen ihn auch inkrementelle Datenstruktur.

Der `Solver` führt die Iteration über die Eingabeelemente durch, verwaltet also die Eingabe und führt den Algorithmus aus. Er funktioniert unabhängig vom konkreten Algorithmus und liegt in verschiedenen Varianten als Templateklasse vor.

Der parallele Solver übernimmt zudem das Management der Threads und die Verteilung der Eingabe auf die Threads nach verschiedenen Vorgaben. Inkrementellen Algorithmen behandeln die Eingabeelemente einzeln. Daher kann die Arbeit leicht und nach verschiedenen Mustern unter den arbeitenden Threads aufgeteilt werden. Die Threads des `Solver`s iterieren über den ihnen zugeordneten Teil der Eingabe und rufen die threadsicheren Funktionen der inkrementellen Datenstruktur D auf. Jedes Element x der Eingabe wird in zwei Schritten behandelt:

1. Lokalisierung: Suche zukünftige Position p von x in der Datenstruktur D.

2. Update: Erweitere D mit x bei der Position p.

Die Datenstruktur D ist in unserem Kontext eine geometrische Struktur, die sich in der Regel aus einfachen geometrischen Objekten zusammensetzt. Die Position p beschreibt ein Element von D, das die neue Position von x eindeutig identifiziert. Die Update Operation fügt das neue Elemente zu D hinzu und/oder verändert vorhandene Objekte. Nach jeder Update Operation repräsentiert D eine komplette und konsistente Lösung der bis dahin abgearbeiteten Elemente.

Die Lokalisierung wird in einigen Algorithmen von einer Suchstruktur unterstützt, so dass die lesenden Operation der Lokalisierung von den Schreibenden des Updates getrennt werden können. Die Trennung führt bei konkurrierenden Threads zu weniger Konflikten und mehr Leistung.

4.3.1. Eine inkrementelle Datenstruktur

Die inkrementelle Datenstruktur D ist eine templatisierte `C++` Klasse. D ist ein Container für Objekte einer eingebetteten Klasse E. Die Funktionen von D verarbeiten eine Eingabe, die im Iteratortyp IT gekapselt ist.

Die Datenstruktur implementiert die folgende Schnittstelle:

- `typedef elem_pointer E*` definiert einen Zeiger zum gekapselten Typ E. Der Zeigertyp wird durch den `Solver` genutzt.
- `bool locate(IT x,elem_pointer) p` lokalisiert ein Objekt p von D, das die Position von x in D spezifiziert.
- `bool update(IT x,elem_pointer) p` fügt x mit Hilfe von p zu D hinzu.
- `void init(elem_pointer)` einmalige Ausführung pro Thread vor dem ersten Aufruf von `locate`.
- `void clean_up(elem_pointer)` einmalige Ausführung pro Thread nach dem letzten `update` Aufruf.

Die Klasse D implementiert in den Funktionen `locate` und `update` einen inkrementellen Algorithmus und repräsentiert gleichzeitig die Lösung. Die Struktur D muss so initialisiert werden, dass jeder Aufruf von `locate` mit einem beliebigen Element der Eingabe immer ein Ergebnis liefert.

Die aufgezählten Funktionen - insbesondere `locate` und `update` - werden von den Threads genutzt um konkurrierend auf D zuzugreifen. Sie müssen daher threadsicher sein. Synchronisationstechniken haben wir in Kapitel 2 behandelt. Bei der Verwendung des gegenseitigen Ausschlusses müssen wir auch die Verwendung von Strategien zur Deadlock Vermeidung vorsehen. Hierzu unterstützt unser Framework den begrenzten Abbruch von Berechnungen wie in Abschnitt 2.3.2 beschrieben. Wir definieren, dass die Funktionen `locate` und `update` eine erfolgreiche Ausführung mit $true$ und einen erzwungenen Abbruch wegen einer Threadkollision mit $false$ quittieren. Vor einem Abbruch, muss die fehlgeschlagene Funktion einen konsistenten Zustand von D wiederherstellen. Der Thread muss alle Mutex Variablen in Besitz entsperren. Das Framework wiederholt die Berechnung mit dem aktuellen Eingabeelement x zu einem beliebigen späteren Zeitpunkt.

Weitere Designziele des Containers haben wir bereits in Abschnitt 2.5.1 zusammengefasst. Der Container soll eine möglichst kleine Anzahl an Mitgliedsvariablen enthalten. Diese sollen nach Möglichkeit Konstanten sein. Damit minimieren wir den Synchronisationsbedarf für globale Variablen der Threads bereits im Ansatz. In Abschnitt 2.5 haben wir zudem verschiedene threadsichere Methoden zum Thema Memory Management und Garbage Collection vorgestellt. Zur Initialisierung von threadbezogenen Hilfsstrukturen dienen die Funktionen `init` und `clean_up`.

Ein Container, der die geforderten Funktionen threadsicher implementiert, kann automatisch parallel konstruiert werden. Die Definition eines Zeigertyps für das eingebettete Element der Struktur ist eine technische Notwendigkeit für die Schnittstelle mit der `Solver` Klasse. Unser Framework unterstützt nur eine eingebettete Klasse.

Beispiel 4.1 (Sortieren in einem Binärbaum).
Wir entwerfen eine inkrementelle Baumstruktur. Die Elemente des Binärbaums sind knotenorientiert sortiert. Dieses Beispiel verdeutlicht die Funktionsweise des Containers und ist nicht notwendigerweise effizient.

Code 4.1 (Inkrementeller Binärbaum, Klassendeklaration).
```
template<class T> class binary_tree {
  class node {
    mutex mtx;
    T value;
    node *left, *right;
  };

  typedef node* elem_pointer;
  elem_pointer root;

  bool locate(const T& x, elem_pointer& v)
  { // usual search in binary tree}

  bool update(const T& x, elem_pointer v); }
```

Der Container `binary_tree` enthält die Deklaration der Baumknoten. Ein Knoten besteht aus einem Wert `value` und zwei Zeigern auf die Kinder. Zusätzlich ist ein Knoten durch einen Mutex gesichert. Des Weiteren sind im Container ein Zeiger auf die Baumwurzel und die nötige Typdefinition für den `Solver` deklariert. Die Lokalisierungsfunktion durchläuft den Baum auf der Suche nach einem Knoten v, der Vater des neuen Knoten mit Wert x werden soll. Da neue Knoten ausschließlich als Blätter angefügt werden, bleibt die vorhandene Verzeigerung unverändert. Die `locate` Funktion sperrt die Knoten nicht.

Code 4.2 (Inkrementeller Binärbaum, `update` Funktion).
```
bool update(const T& x, elem_pointer v) {
  bool success = true;
  v->mtx.lock();
  if (x < v->value)
    { if (v->left != NULL) success = false;
      else v->left = new node(v);
     }
  else
    { if (v->right != NULL) success = false;
      else v->right = new node(v);
    }
  v->mtx.unlock();
  return success; }
```

Die `update` Funktion erhält als Argumente den einzufügenden Wert x und den Zeiger auf den vermeintlichen Vater v. Da wir v in der `locate` Funktion nicht gesperrt haben,

muss v threadsicher auf seine Gültigkeit getestet werden. Die Funktion sperrt den Mutex und führt den Test durch.

Wenn ein anderer Thread in der Zwischenzeit einen Knoten eingefügt hat, der den Platz von x eingenommen hat, kommt es zu einer Kollision zwischen den Threads. Die Funktion muss mit $false$ abbrechen. Da zu diesem Zeitpunkt noch keine Änderungen an dem Baum durchgeführt wurden, müssen diese auch nicht abgewickelt werden. Nur der Mutex wird entsperrt. Läuft der Test hingegen erfolgreich, kann der neue Knoten an den gesperrten Knoten v angefügt werden.

In diesem Beispiel können keine Deadlocks auftreten, da jeder Thread nur höchstens einen Knoten gleichzeitig sperrt.

4.3.2. Ein sequentieller `Solver`

Ein sequentieller Solver ist denkbar einfach, da Threadverwaltung und Deadlockvermeidung wegfallen. Die randomisierte Eingabe wird in beliebiger Reihenfolge abgearbeitet.

Code 4.3 (Struktur eines randomisiert inkrementellen Algorithmus).

```
void start(iterator start, iterator end) {
  data_struct::elem_pointer p = 0;
  D.init(p);
  iterator it = start;
  while (it!=end)
  {
    D.locate(*it,p);
    D.update(*it,p);
    it++;
  }
  D.clean_up(p); }
```

Der Solver iteriert mit der Variable `it` über die Eingabe und benutzt die Funktionen des Containers zur Konstruktion der Struktur. Die Funktion `locate` weißt an die Variable `p` die ermittelte Position zu, an der `update` einfügt. Die Rückgabewerte der Funktionen spielen keine Rolle. Mit diesem `Solver` können auch linear inkrementelle Algorithmen gelöst werden.

4.3.3. Ein paralleler `Solver`

Die parallele Solver Klasse hat folgende Aufgaben:

- Starten und Beenden der Threads.
- Verteilung der Eingabedaten auf die Threads.
- Unterstützung von Abbruchstrategien bei Synchronisationsproblemen bzw. Kollisionen von Threads.

Der Solver startet eine gegebene Anzahl an Threads und teilt die Eingabe zu. Die Verteilung kann dabei in mehreren Varianten erfolgen. So kann das Framework beispielsweise die Eingabe beim Start der Threads gleichmäßig verteilen oder die Eingabe blockweise bei Bedarf zuweisen. Die Eingabe ist als Iteratorenpaar gegeben.

Die Threads des Frameworks verarbeiten ihre Eingabe mit den Funktionen `locate` und `update` der Struktur D. Die Funktionen sind threadsicher und brechen bei einer Kollision ggf. ab, siehe Abschnitt 2.3.2. Grund für einen Abbruch sind die Erkennung eines potenziellen Deadlocks oder die Nichterfüllung einer notwendigen Bedingung zur Ausführung.

Hierzu signalisiert der Rückgabewert *true* eine erfolgreiche Durchführung einer Lokalisierung oder Aktualisierung der Datenstruktur. Nach erfolgreicher Ausführung einer Operation fährt ein Thread normal fort. Terminiert eine der Funktionen mit $false$ muss der Thread die Berechnung neu starten. Um die Wiederholungsgefahr der gleichen Kollision zu verringern, startet der Thread die nächste Berechnung nicht mit dem gleichen Element. Der Thread vertauscht zufällig das gerade fehlgeschlagene Element mit einem anderen unbearbeiteten Element, siehe folgendes Codebeispiel. Die Bearbeitung des fehlgeschlagenen Elementes wird in jedem Fall nachgeholt.

Code 4.4 (Schleife des parallelen Solvers).

```
void start(iterator start, iterator end) {
  data_struct::elem_pointer p = 0;
  D.init(p);
  iterator it = start;
  while (it!=end)
  { if (D.locate(*it,p) && D.update(*it,p))
      it++;
    else
      { int remain = end-it-1;
        swap(*it,*(it + rand(0,remain));
      }
  }
  D.clean_up(p); }
```

In dieser Solver Variante wird allen Threads eine disjunkte, gleich große Eingabemenge statisch zugewiesen, repräsentiert durch die Iteratoren `start/end`. In der gezeigten Funktion `start` iterieren die Threads über die Eingabe und versuchen Element für Element in die Struktur D einzubauen. Solange `locate` und `update` erfolgreich verlaufen, wird die Eingabe planmäßig abgearbeitet. Schlägt eine der Operationen fehl, findet eine zufällige Vertauschung des aktuellen mit einem unbearbeiteten Element statt. Hierzu benötigen wir Random Access Iteratoren.

Im folgenden Codebeispiel sehen wir einen Ausschnitt der Definition des Solvers. Die Klasse erhält zwei Template Argumente: der Iteratortyp der Eingabe und den Typ der inkrementellen Datenstruktur. Wir schreiben die Benutzung eines Random Access Iterator vor. Das Interface aller Solver beschränkt sich auf zwei Funktionen:

- Der Konstruktor erhält die initialisierte Datenstruktur D und die Anzahl der zu nutzenden Threads.
- Die `solve` Funktion startet die Berechnung. Die Argumente sind ein Iteratorpaar, das Anfang und Ende der Eingabemenge markiert.

Code 4.5 (Standart `solver`, Ausschnitt der Klassendefinition).

```
template <class iterator, class data_struct>
class inc_parallel_solver {

  data_struct& D;
  int thread_num;
  void start(iterator start, iterator end);
  ...
public:
  inc_parallel_solver(data_struct& ds, int num) : D(ds),
     thread:num(num) { };

  void solve(iterator start, iterator end)
  { //initiate "thread_num" threads with common structure res
    //and distribute the input [start,end] equally
    //Each thread executes start(...)
    ...} };
```

Die angeführte Klasse `inc_parallel_solver` ist der von uns verwendete Standard Solver. Zu ihr gehört die oben gegeben Implementierung der Funktion `start`.

Wir stellen verschiedene Solver Varianten zur Verfügung, die sich nur in der Art und Weise der Eingabeverteilung unterscheiden:

`inc_parallel_solver` Die Eingabe wird statisch gleichmäßig verteilt.

`inc_parallel_solver_dar` Die Eingabe wird blockweise und bei Bedarf an die Threads verteilt. Bei großen zeitlichen Unterschieden in der Behandlung eines einzelnen Eingabeelements sichern wir uns praktisch gegen ungleiche Lastverteilung der Threads ab. Theoretisch soll die Gleichverteilung aufgrund der Randomisierung der Eingabe ohnehin gegeben sein.

`inc_parallel_solver_exp` Die Eingabe wird erst sortiert (parallel), danach gleichmäßig auf die Threads aufgeteilt und anschließend von den Threads wieder permutiert. Wir versprechen uns davon bei einigen Algorithmen eine Minimierung der Kollisionen der Threads, da diese geometrisch betrachtet weniger Schnittpunkte haben.

4.3.4. Zusammenfassung

Wir haben die beiden Teile unseres Frameworks vorgestellt. Die Solver Klasse übernimmt die Verwaltungstätigkeiten für Eingabe und Threads und ist unabhängig vom inkrementellen Algorithmus anwendbar. Sie liegt dazu in unterschiedlichen parallelen und einer

sequentiellen Variante vor. Die parallelen Solver unterstützen den Abbruch von Berechnung wegen Synchronisationsproblemen.

Die inkrementellen Datenstruktur ist der komplexere Teil des Frameworks. Passend zur `Solver` Klasse haben wir eine Schnittstelle für die Datenstruktur definiert. Beim Design der Datenstruktur bestehen genügend große Freiräume. Im Wesentlichen hängt die Implementierung der Datenstruktur an den Funktionen `locate` und `update`. Wir machen keine Vorgaben über Größe oder Komplexität der Datenstruktur. Bisher kann der `solver` nur einen eingebetteten Typ der Datenstruktur handhaben. Dieser Umstand war praktisch noch kein Problem, muss ggf. zukünftig nochmal überprüft werden.

Ein wichtiger Aspekt beim Design der konkurrierenden Funktionen ist die Definition der Mutex-Lock Reihenfolge und damit zusammenhängend die Frage der Deadlock Vermeidung. Beides muss bis zum einem gewissen Grad individuell auf den konkreten Algorithmus angepasst werden, basiert aber auf den Techniken, welche wir in Abschnitt 2.3.2 eingeführt haben. Wir werden hierzu in den folgenden Abschnitten einige Beispiele sehen.

4.4. Die konvexe Hülle

Der randomisiert inkrementelle Algorithmus zur Berechnung der konvexen Hülle in 2D wird bei Mehlhorn und Näher [6] diskutiert. Dort ist eine ausführliche Beschreibung inklusive Korrektheitsbeweis zu finden. Wir erklären kurz den Algorithmus und gehen dann auf die Details der Parallelisierung ein, inklusive der Reihenfolge der Mutex Sperrung und der Deadlock Vermeidung. Unserer parallelen Implemntierung dient die Arbeit von Mehlhorn und Näher als Vorlage. Die erwartete asymptotische Laufzeit dieses Algorithmus beträgt $O(n \log n)$ für die Eingabegröße n.

Die inkrementelle Datenstruktur repräsentiert die konvexe Hülle als doppelt verkettete Liste aus Kanten. Punkte die nicht im Inneren der Hülle liegen, werden Außen angebaut. Beim sukzessiven Ausdehnen der Hülle werden paarweise Kanten hinzugefügt, die eine oder mehrere vorhandene Hüllkanten ersetzen. Die ersetzten Kanten werden nicht aus der Struktur entfernt, sondern bleiben als Suchkanten im Inneren der Hülle erhalten. Es existieren also Hüllkanten, die mit ihren Nachbarn verzeigert sind, und Suchkanten, die auf die weiter außen liegenden Kanten verweisen, durch die sie selbst ersetzt wurden. Die inkrementelle Datenstruktur wird mit drei beliebigen, nicht collinearen Punkten initialisiert, also aus einem Dreieck von Hüllkanten.

Sei S die Punktmenge zu der die konvexe Hülle $CH(S)$ berechnet wurde. Die Erweiterung der Menge S um Punkt p erfolgt in zwei Schritten. Wir nehmen hierzu an, das $p \notin CH(S)$ gilt: Zuerst lokalisieren wir mit den Suchkanten eine Hüllkante e, die p "sieht". D.h. eine Verbindung von p nach e ist nicht in $CH(S)$. Im zweiten Schritt berechnen wir ausgehend von e durch Iteration über die Hüllkanten von $CH(S)$ die Tangentialpunkte a, b zwischen $CH(S)$ und p. Wir fügen die Kanten (p, a) und (p, b) ein und erzeugen die Hülle $CH(S \cup p)$. Die alte Hülle $CH(S)$ bestand zwischen a und b aus bis zu $|CH(S)| - 1$ Hüllkanten. Diese verweisen jetzt als Suchkanten auf (p, a) und (p, b).

Die Lokalisierung einer Hüllkante e zu einem Eingabeelement p bedient sich einer Such-

struktur. Die Suchstruktur besteht aus vormaligen Hüllkanten, die nach ihrem Gebrauch umgewidmet wurden. Verliert eine Hüllkante h ihre Funktion, wird sie von zwei neuen Hüllkanten a, b überdeckt. Statt einer doppelten Verkettung mit den Nachbarn, verweist h nun auf die Kanten a und b. Die Suchkanten bilden im Inneren der Hülle eine hierarchische Suchstruktur, welche der Historie früherer Hüllen entspricht. Im Innersten liegt das initiale Dreieck. Auf der Suche nach Kante e für Punkt p starten wir im Dreieck und iterieren durch die Historie der Hülle. Ein Iterationsschritt besteht aus der Suche nach einer Kante, die von p aus in der jeweiligen Ebene der Suchstruktur zu sehen ist. Dazu stehen im ersten Schritt drei Kanten zur Wahl, die auf eine Sichtbarkeit von p zu prüfen sind, in den Folgeschritten jeweils zwei Kanten. Der Lokalisierungsalgorithmus bewegt sich durch die Historie nach außen bis er eine Hüllkante findet oder keine weitere sichtbare Suchkante ermitteln kann. Im letzten Fall liegt p in der Hülle und wird verworfen. Lokalisierung und Ersetzung sind nochmals in Abbildung 4.1 illustriert.

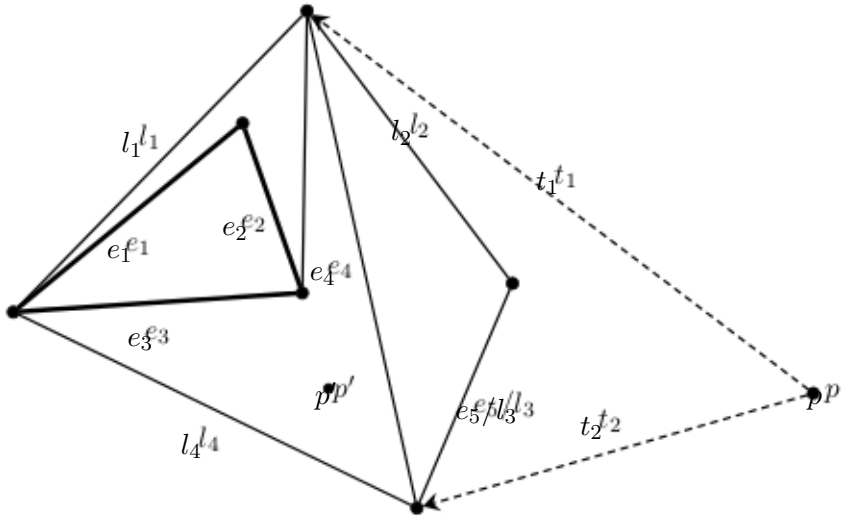


Abbildung 4.1.: Inkrementelle Konstruktion der konvexen Hülle; Initialisierung: Die Hülle wurde mit den Kanten e_1 bis e_3 initialisiert; Lokalisierung: Für Punkt p wird durch die Sequenz der Kanten e_3, e_4, e_5 die Hüllkante e_5 ermittelt. Für p' stoppt die Suche bei e_3 welche keine Hüllkante ist; Update: die Hüllkanten l_2 und l_3 werden ersetzt durch t_1 und t_2. Die Kanten l_1 und l_4 werden mit ihren neuen Nachbarn verlinkt. Die Kanten l_1 bis l_4 müssen gesperrt werden.

4.4.1. Implementierung

Die Klasse `hull_element` repräsentiert sowohl Hüll- als auch Suchkanten der konvexen Hülle. Der Kantentyp wird durch die Variable `outside` bestimmt. Hüllkanten sind mit *true* markiert. Weiter speichert jede Kante ihre Endpunkte in den Variablen `source` und `target` und ihre Nachbarn (Hüllkante) bzw. ihre "Überdachung" (Suchkante) in den Zeigern `pred` und `succ`. Die Hüllkanten bilden eine doppelt verkettete zyklische Liste.

Jede Kante verfügt weiter über einen Mutex `mtx`. Wir definieren, dass die Kanten gegen den Uhrzeigersinn gerichtet sind, d.h. der Zeiger `succ` der Hüllkante weißt gegen den Uhrzeigersinn.

Code 4.6 (Ausschnitt Klassendeklaration einer Kante).

```
template <class point>
struct hull_element {
  ...
  point   source,target;
  bool    outside;
  hull_element *succ, *pred;
  mutex mtx; }
```

In der Definition der inkrementellen Datenstruktur stecken keine Überraschungen. Die von der Schnittstelle geforderten Funktionen wurden implementiert. Der Zeigertyp `elem_pointer` wird als Zeiger auf ein `hull_element` definiert. Die Variable `T` enthält das initialisierte Dreieck. Da kein konkurrierender schreibender Zugriff auf die Klassenvariablen stattfindet, kommt der Container ohne Mutex aus.

Code 4.7 (Ausschnitt Klassendeklaration).

```
template <class point>
class hull_struct {
  struct hull_element;
  hull_element* T[3];
public:
  ...
  typedef hull_element* elem_pointer;

  bool locate(const point& p, elem_pointer& ptr)
  { // walk through the location structure }

  bool update(const point& p, elem_pointer& ptr); }
```

Die threadsichere `locate` Funktion entspricht der Implementierung von Mehlhorn und Näher [6]. Sie liefert eine Hüllkante e, die vom Eingabepunkt gesehen werden kann. Die `locate` Funktion führt ausschließlich lesende Operationen auf Suchkanten und einer Hüllkante durch. Suchkanten werden an keiner Stelle (auch nicht der `update` Funktion) verändert. Zu Kollisionen kann es nicht kommen, eine Synchronisation ist in dieser Funktion daher nicht notwendig.

Zu Beginn der Ausführung der `update` Funktion müssen überprüfen, ob die gefunden Hüllkante e immer noch eine Hüllkante ist. Da die Funktion locate keine Kanten sperrt, ist e erst zu sperren und dann zu überprüfen. Ist die Bedingung nicht erfüllt, muss die update Funktion abbrechen.

Umgekehrt darf die Funktion update Hüllkanten erst durch setzen der `outside` Variable zu Suchkanten umwidmen, nachdem die Verzeigerung angepasst wurde. Da Suchkanten

von locate nie gesperrt werden, ist diese Reihenfolge notwendig, um locate nicht in einen undefinierten Zustand laufen zu lassen.

An diesen einfachen Beispielen ist zu erkennen, dass Annahmen über den Zustand der Datenstruktur genauer zu formulieren oder zu überprüfen sind, als man dies von der sequentiellen Programmierung gewohnt ist.

Code 4.8 (Konvexe Hülle, Berechnung einer Tangente `update`).

```
template <class point>
bool update(const point& p, elem_pointer& ptr)
  { if (!ptr->mtx.trylock()) return false;
    if (!ptr->outside) { ptr->unlock(); return false; }

      hull_element* high = ptr;
      hull_element* low = ptr;

     // compute upper tangent
      bool ok = true;
      do { high = high->succ;
           if (!high->mtx.lock()) { ok = false; break; }
         } while (!left_turn(high->source,high->target,p));

     if (!ok)
     { ptr = low;
       while (ptr != high)
       { hull_element* q = ptr->succ;
         ptr->mtx.unlock();
         ptr = q; }
       high->mtx.unlock();
       return false; }
...
```

Wir benutzen zunächst den naiven Ansatz der Deadlock Vermeidung 2.3.2. D.h. wir reagieren, sowohl auf einen erfolglosen Sperrversuch durch `trylock()` auf eine Kante, als auch auf eine Verletzung einer Ausführungsbedingung, mit einem Abbruch. Ein Abbruch bedeutet das Entsperren aller Mutex Variablen und eine Terminierung mit $false$.

Die Funktion update berechnet die obere Tangente mit Hilfe des `left_turn` Prädikats. Der Thread iteriert über die Kanten bis die Tangente gefunden ist und sperrt diese. Die Berechnung der unteren Tangente erfolgt analog. Konnten die Tangenten erfolgreich berechnet werden, sind gleichzeitig alle betroffenen Hüllkanten gesperrt und die neuen Hüllkanten werden angefügt. Zum Abschluss werden alle Sperren aufgelöst und *true* zurückgegeben.

Code 4.9 (Konvexe Hülle, Erfolgreiche Terminierung `update`).

```
Fortsetzung von oben:
     // mark edges between low and high as "inside",
```

```
    // define refinements and unlock all mutexes
    ptr = low;
    while (ptr != high)
    { hull_element* q = ptr->succ;
      ptr->pred = e_l;
      ptr->succ = e_h;
      ptr->outside = false;
      ptr->mtx.unlock();
      ptr = q; }
    high->mtx.unlock();
  return true; }
```

Wichtig ist es folgende Reihenfolge zu beachten:

- zuerst werden alle benötigten Elemente berechnet *und* gesperrt,
- danach folgen die Änderungen
- und am Ende werden alle Sperren gelöst.

Mit dieser Vorgehensweise kann man einfach auf erzwungene Abbrüche durch Threadkollisionen reagieren.

Wir benutzen in der Datenstruktur weiter ein einfaches Speichermanagment bestehend aus einem Stapel auf dem alle erzeugten Kanten abgelegt werden. Der Stapel ist lockfrei synchronisiert, siehe 2.4.1.3. Im Folgenden werden wir die Deadlock Vermeidung verfeinern.

4.4.2. Deadlock Vermeidung

Wir besprechen zunächst die Deadlock Vermeidung mit der Mutex Hierarchie. Um die Mutex Hierarchie anwenden zu können, müssen wir eine Hierarchie über die Mutex Variablen definieren, siehe 2.3.3.

Der Algorithmus muss die Hüllkanten sperren, die zyklisch verkettet und gegen den Uhrzeigersinn sortiert sind. Durch die Sortierung ist eine zyklische Ordnung bereits gegeben. Wir müssen noch den Zyklus aufbrechen, um eine hierarchische Ordnung herzustellen. Dazu suchen wir einen Extrempunkt der Punktmenge. In welcher Richtung der Punkt extrem ist, ist dabei gleich. Ein Extrempunkt einer beliebigen Richtung ist Teil der konvexen Hülle. Dieser Punkt markiert Anfangs- und Endsegment einer nicht-zyklischen, doppelt verketteten Liste.

Wird ausgehend von einer gegebenen Kante gegen den Uhrzeigersinn gesperrt, muss der Thread die `lock()` Operation verwenden. Mit dem Uhrzeigersinn ist weiter `trylock()` im Einsatz und bei Fehlschlag folgt der Abbruch. Damit ist sichergestellt, dass mindestens ein Thread ein Fortkommen erzielen kann, ohne durch einen Abbruch aufgehalten zu werden.

Zum Einsatz der Thread Hierarchie verwenden wir die Mutex Implementierung, die wir in Abschnitt 2.3.5 vorgestellt haben. Von diesem Mutex verwenden wir hier prinzipiell die `lock()` Operation.

Die Thread Hierarchie lässt sich unmittelbar und denkbar einfach anwenden, worin ihr ausdrücklicher Vorteil liegt. Zur Mutex Hierarchie müssen wir Bedingung für die Datenstruktur aufstellen, was bedeutet die Technik auf jeden konkreten Algorithmus einzeln abzustimmen und Zusatzaufwände in Kauf zu nehmen. Im konkreten Fall müssen wir einen Extrempunkt berechnen.

4.5. Delaunay Triangulierung

Eine Delaunay Triangulierung ist die Triangulierung einer Punktmenge bei der jedes einzelne Dreieck die s.g. Delaunay Bedingung erfüllt. Diese besagt, dass der Kreis, der durch die Ecken eines Dreiecks definiert ist (Umkreis), keine weiteren Punkte der Triangulierung enthält. Die Delaunay Bedingung wird mit dem `incircle` Prädikat getestet.

Zur Parallelisierung der Delaunay Triangulierung wählen wir den Flipping Algorithmus. Der Flipping Algorithmus ist in der Literatur wohl bekannt. Er basiert auf der Arbeit von Guibas et al. [53] und ist im LEDA Buch [6] und viele weiteren Papieren beschrieben [50, 51, 52]. Eine bestehende Implementierung ist in der LEDA Bibliothek zu finden [7]. Eine alternative Implementierung ist in [54] zu finden.

Der Flipping Algorithmus transformiert eine gegebene Triangulierung in eine Delaunay Triangulierung. Eingabe des ursprünglichen Algorithmus ist also nicht eine Punktmenge, die inkrementell verarbeitet wird, sondern eine beliebige Triangulierung. Wir verwenden den Flipping Algorithmus in unserem Schema für randomisiert inkrementellen Algorithmen wie folgt:

- Zuerst lokalisieren wir zu einem zufällig gewählten Punkt p die umgebende Facette, bzw. das umgebende Dreieck D der vorhandenen Delaunay Triangulierung.
- Danach erweitern wir die Triangulierung in D naiv um den Punkt p ohne Rücksicht auf die Delaunay Eigenschaft zu nehmen.
- Zuletzt korrigieren wir die Delaunay Triangulierung mit dem Flipping Algorithmus ausgehend von p.

Das Update folgt dabei im Einzelnen der Vorgehensweise, welche wir schon zur konvexen Hülle empfohlen haben. Die Synchronisation erfolgt durch Mutex Variablen. Der Flipping Algorithmus bestimmt die Dreiecke, die durch Hinzunahme von p die Delaunay Bedingung verletzen und sperrt diese gleichzeitig (und deren Nachbarn). Die markierten Dreieck werden aus der Triangulierung entfernt und Delaunay konforme Dreiecke eingefügt. Die gesperrten Dreiecke werden in einer Liste aufgereiht, um am Ende alle verbliebenen Mutex Variablen einfach zu lösen, siehe Abbildung 4.2.

Zur Deadlock Vermeidung verwenden wir die bekannte Thread Hierarchie. Der Thread, der kollidiert und abbricht, hat noch keine Änderungen an der Struktur vorgenommen. Er löst die Mutex Variablen und startet neu.

Zur Lokalisierung des Dreiecks D sind verschiedene Techniken verfügbar, die auf Suchstrukturen basieren. Wir verzichten auf solche und implementieren einen naiven Ansatz.

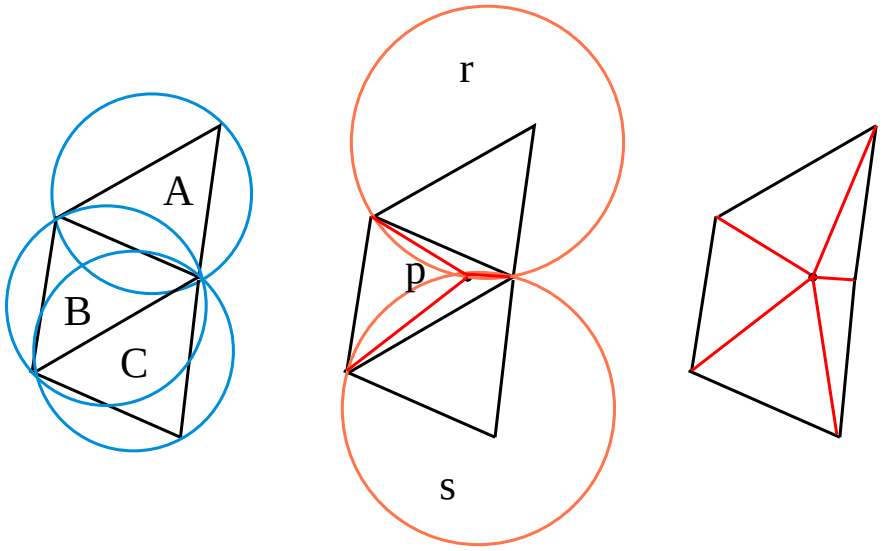


Abbildung 4.2.: Delaunay Triangulierung, Update: Links: Delaunay Triangulierung aus den Dreiecken A,B,C, die Umkreise sind blau; Mitte: Naive Triangulierung mit p, die Umkreise r und s (rot) verletzen die Delaunay Bedingung; Rechts: Korrektur durch den Flipping Algorithmus.

D.h. wir starten bei einem beliebigen Dreieck A und benutzen einen einfachen Wegfindungsalgorithmus, um zum Ziel zu gelangen. Ausgehend von einem Dreieck testen wir mit Hilfe des `orientation` Prädikats, welche Kante das Ziel p sieht, also eine gedachten Linie zwischen Ausgangsort und Ziel schneidet. Den Test wiederholen wir, bis wir am Zieldreieck angelangt sind, siehe Abbildung 4.3.

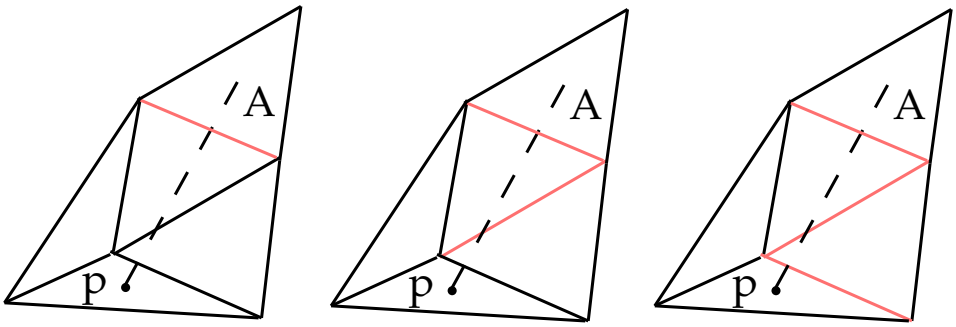


Abbildung 4.3.: Delaunay Triangulierung, Lokalisierung; Die Lokalisierung startet in Dreieck A. Ein Orientation Test wird auf alle Kanten angewendet. Die rote Kante sieht p. Der Vorgang wiederholt sich bis p gefunden ist.

Die Lokalisierungsfunktion arbeitet ohne eigene Suchstruktur direkt auf der Delaunay

Triangulierung. Die Delaunay Triangulierung änder sich ständig. Daher sperrt die Lokalisierungsfunktion das Dreieck, das sie aktuell besetzt, ab. Das nachfolgende Dreieck wird gesperrt bevor das vorangegangenen abgegeben wird. Auch die Lokalisierung kann hier Kollisionen verursachen.

Die Triangulierung wird initialisiert durch ein heinreichend großese Dreieck, das alle Punkt enthält. Die Seiten des Dreiecks dienen auch als Einstiegspunkt für die Lokalisierung eines Punktes.

4.5.1. Implementierung

Unsere Implementierung basiert im Wesentlichen auf der Implementierung der LEDA Bibliothek. Diese Implementierung haben wir auf unsere Datenstruktur angepasst, die wir weiter unten einführen. Darüber hinaus möchten wir keine Codebeispiele zeigen, da die Änderungen am LEDA Code einfach umsetzbar sind. Die Modifikationen am Code zur Threadsicherheit sind oben beschrieben und folgen dem Schema, das wir zur konvexen Hülle diskutiert haben.

Wir führen kurz die eingebettete Klasse `delau_element` des Containers `delau_struct` ein:

Code 4.10 (Element der Delaunay Struktur).

```
struct delau_element {
  point   P[3]; // A = P[0], B = P[1], C = P[2], counterclockwise
  delau_element *N[3]; // AB = 0, BC = 1, CA = 2,
  int mark[3];\label{MTP:MUT:thread}
  mutex mtx;

  bool incircle(const point& p)
    {return leda::incircle(P[0],P[1],P[2],p);}
  ... };
```

Die Klasse `delau_element` repräsentiert ein Dreieck gegeben durch die 3 Punkte `P[]`, doppelt verkettet mit den drei Nachbarn der Triangulierung `N[]` und gesichert durch einen Mutex `mtx`. Die Klasse enthält außerdem Markierungen für die Kanten `mark[]` und implementiert das `incircle` Prädikat.

Der Container `delau_struct` besitzt nur eine Klassenvariable, das initiale Dreieck. Das Memory Management dieses Containers muss auch das löschen von Elementen unterstützen, welches von der Aktualisierungsfunktion verwendet wird.

Der Ausschnitt der Triangulierung, der durch den Flipping Algorithmus ersetzt wird, ist sternenförmig. Während der Ermittlung dieses Ausschnitts müssen alle Dreiecke des Sterns und die Nachbarn des Sterns gesperrt werden. Die schrittweise Sperrung der Dreiecke geht nach dem Prinzip der Tiefensuche vor. Damit begegnen wir benachbarten Dreiecken unter Umständen doppelt und versuchen diese zu sperren. Hierzu haben wir einen rekursiven Mutex eingeführt, der sich vom gleichen Thread auch mehrfach sperren lässt. Damit entfällt eine Prüfung und Fallunterscheidung doppelt gesehener Dreiecke,

sowohl beim Sperren, als auch beim Entsperren. Dieses Detail lässt sich natürlich auch anders lösen.

Code 4.11 (Mutex Variable mit rekursivem `lock`).

```
int lock() {
  pthread_t my_t = pthread_self();
  if (my_t == lock_id) {locks++; return OWNER;}
  while (!mutex.try_lock())
    { if (my_t < lock_id) return BLOCKED; }
  locks++;
  lock_id = my_t;
  return LOCKED; }

int unlock() {
  if (pthread_self() != lock_id) {exit(1);}
  locks--;
  if (locks == 0)
    {
      lock_id = 0;
      mutex.unlock();
    }
  return locks; }
```

Im Codebeispiel sehen wir, dass bei wiederholter Sperrung durch den gleichen Thread `(my_t == lock_id)` die Variable `locks` inkrementiert wird. Diese wird beim Entsperren dekrementiert. Der Mutex ist erst frei, wenn `locks == 0` gilt. Diese Funktionen sind leicht modifizierte Varianten der Mutex Variable mit Thread Hierarchie aus Abschnitt 2.3.5.

Bei der Parallelisierung der Delaunay Triangulierung haben wir bekannte Methoden aus dem vorherigen Abschnitt zur konvexen Hülle auf vorhandenen sequentiellen Code der LEDA Bibliothek angewandt. Unsere Methoden betreffen das Design der Datenstruktur, die Struktur einer Aktualisierungsfunktion und die Strategie zur Deadlock Vermeidung. In der Summe waren beim Flipping Algorithmus wegen des Redesigns der Datenstruktur und der Restrukturierung der Aktualisierungsfunktion viele Änderungen am Code vorzunehmen. Im Unterschied dazu konnten wir beim konvexe Hülle Algorithmus große Teile der Implementierung identisch übernehmen. Ob eine Transformation von sequentiellem in parallelen Code einfach machbar ist hängt von der konkreten sequentiellen Implementierung ab.

4.5.2. Laufzeit und Verbesserungen

Der von uns parallelisierte Flipping Algorithmus ist zunächst kein randomisiert inkrementeller Algorithmus. Erst durch die beschriebenen Modifikationen passt er in unser Framework. Dadurch ergeben sich Nachteile. Die Lokalisierungsfunktion ist mit einer erwarteten Laufzeit von $O(n^{3/2})$ bei einer gleichverteilten und randomisierten Eingabe zu

komplex. Die Aktualisierungsfunktion hat nach Guibas et al. [52] eine erwartete Laufzeit von $O(n)$. Daraus ergibt sich eine erwartete Laufzeit von $O(n^2)$.

Großes Verbesserungspotenzial liegt in der Suchfunktion. Anstelle eines naiven Ansatzes ist eine Suchstruktur zu verwenden. Eine häufig aufzufindende Idee lautet diese aus der Historie der Delaunay Triangulierungen zu konstruieren wie schon bei der konvexen Hülle. In der Literatur gibt es hierzu verschiedene Ansätze, siehe wieder bei Guibas et al. [52], bei Devillers [55, 56] oder bei Kohout et al. [57]. Letztere entwerfen bereits einen randomisiert inkrementellen Algorithmus mit Hinblick auf die Parallelisierung. Mit einer Suchstruktur lässt sich die erwartete Laufzeit auf $O(n \log n)$ reduzieren.

4.6. Offene Punkte

Neben der Implementierung und Parallelisierung weitere randomisiert inkrementeller Algorithmen, beispielsweise zu Voronoi Diagrammen, bieten sich zwei weitere Themen zur Vertiefung an:

Zum Einen die systematische Überprüfung und Implementierung von Suchstrukturen. Im Beispiel der konvexen Hülle konnten wir in der Lokalisierungsfunktion eine Suchstruktur verwenden. Die Struktur beschleunigt die Lokalisierung und verringert die Anzahl der Threadkollisionen, da lesende Lokalisierung und schreibende Aktualisierung wenig Schnittfläche haben. Gegenbeispiel ist die Delaunay Triangulierung, wo sich beide Funktionen behindern können und die Lokalisierung eine zu hohe Laufzeitkomplexität hat. Von einer systematischen Verwendung von Suchstrukturen versprechen wir uns Leistungszuwächse.

Zum Anderen lässt die bislang einfache Klasse des parallelen Solvers Raum für Weiterentwicklung. Speziell die Randomisierung der Algorithmen hat womöglich Verbesserungspotenzial. Wir haben einen experimentellen Solver implementiert, der eine Eingabe mit einer gegebene Relation vorsortiert, danach über die Threads verteilt und die verteilte Eingabe wieder permutiert. Es geht zum einen um die Frage wie viel Zufall der Algorithmus benötigt, um die erwartete Laufzeit zu erreichen und zum anderen wie sich Kollisionen der Threads verringern lassen, wenn man die zufällige Wahl beeinflusst. Aus Zeitgründen haben wir diesen Ansatz nicht weiterverfolgt. Ein andere weiterführender Ansatz, der sich mit der Verarbeitung sehr großer Datenmengen durch randomisiert inkrementelle Algorithmen beschäftigt und sich dabei mit auftretenden Wahrscheinlichkeiten beschäftigt, ist in [61] beschrieben.

4.7. Zusammenfassung

Wir haben unser Framework für die Parallelisierung randomisiert inkrementeller Algorithmen vorgestellt. Unser Frameworks kapselt die Verwaltung von Threads und die Balancierung der Threads in einer `Solver` Klasse. Die Threads werden durch den Solver effizient verwaltet und die Anzahl der Threads skaliert.

Die Implementierung der Algorithmen ist in eine threadsichere Datenstruktur ausgelagert. Wir haben verschiedene Beispiele zur Implementierung einer solchen Datenstruktur

gegeben und Regeln für ein gutes Design aufgestellt. Da diese Datenstrukturen Deadlock Vermeidung praktizieren müssen, haben wir unsere leicht benutzbare Lösung zur Deadlock Vermeidung präsentiert.

Grundsätzlich gilt, dass die Parallelisierung inkrementeller Algorithmen komplexer ist und dazu potenziell mehr sequentieller Code überarbeitet werden muss, als dies bei Divide und Conquer Alogrithmen der Fall ist.

5. Experimente

Das zweite Ziel dieser Arbeit ist der Nachweis von Leistungsgewinnen durch Parallelisierung unter Verwendung der vorgestellten Frameworks aus den Kapiteln 3 und 4. Wie Eingangs beschrieben ist die Leistung einer Implementierung der Quotient aus Arbeit (der Problemlösung durch den Algorithmus) und Rechenzeit. Der Faktor der Beschleunigung zwischen der sequentiellen und der parallelen Ausführung einer Impelmentierung ist die erzielte Leistungssteigerung.

Wir haben eine große Menge an Laufzeittest mit den randomisiert inkrementellen und Divide und Conquer Algorithmen durchgeführt, die wir in den Kapiteln 3 und 4 kurz beschrieben haben. Mit den Testläufen haben wir die Laufzeiten gemessen und die Beschleunigungsfaktoren ermittelt. Der Beschleunigungsfaktor einer Implementierung, ausgeführt mit n Threads, bezieht sich dabei immer auf die Laufzeit der gleichen Implementierung, ausgeführt mit *einem* Thread. Die theoretische Obergrenze für einen Beschleunigungsfaktor ist die verwendet Anzahl der Threads n. Auch ein hoher Beschleunigungsfaktor kann den Faktor n nur annähern, nie erreichen, da die notwendige Synchronisation zwischen den Threads, aber auch deren Starts und Stops, immer zu Reibungsverlusten führen. Mit Hilfe der Beschleunigungsfaktoren werden wir die Effizienz unserer Frameworks belegen.

Im nächsten Abschnitt 5.1 beschreiben wir detailliert den Versuchsaufbau und im darauffolgen Abschnitt 5.2 die Instrumente zur Auswertung der Ergebnisse. Danach präsentieren wir die Testergebnisse: in Abschnitt 5.3 zu den Datenstrukturen, in Abschnitt 5.4 zu den Divide und Conquer Algorithmen und in Abschnitt 5.5 zu den randomisiert inkrementellen Algorithmen.

5.1. Versuchsaufbau

Die Testreihen wurden auf einem Linux Ubuntu System der Version 11.10 und einer modernen Intel Core CPU mit 4 Kernen durchgeführt. Beim Betriebssystem handelt es sich um eine Standardinstallation ohne Anpassungen oder Optimierungen jeglicher Art. Um die Ergebnisse nicht zu verfälschen wurde lediglich sichergestellt, dass der Rechenserver für die Durchführung der Experimente exklusiv reserviert ist.

Zur Kompilierung der Testprogramme und der LEDA Bibliothek in der Version 6.2 [7] wurde der g++ Compiler der Version 4.6 verwendet. Dieser Arbeit liegt eine modifizierte Version der LEDA Bibliothek bei, die insbesondere den threadsicheren LEDA Graphen enthält. Die Details des System sind Tabelle 5.1 aufgeführt.

Die Experimente sind in Testreihen organisiert. Eine Testreihe untersucht eine oder mehrere Implementierungen, die mit verschiedenen Parameter gestartet werden. Diese Parameter sind: die Anzahl der zu verwendenden Threads, die Größe der Eingabemenge, die Anordnung der Eingabe (bspw.: sortiert/unsortiert) und für Divide und Conquer

Prozessor	Intel(R) Core(TM) i5-2500 CPU @ 3.30GHz
Anzahl Kerne	4
Cache L3	6 MB
Arbeitsspeicher	16 GB
Betriebssystem	Linux Ubuntu, Version 11.10
Kernel	3.0.0-14-generic, 64bit
Compiler	g++ 4.6
Bibliotheken	LEDA 6.2

Tabelle 5.1.: Systemkonfiguration, Hard- und Software

Algorithmen das Limit (siehe 3.3). Zu jedem Parameter ist eine Liste von Werten definiert. Eine Testreihe führt die Implementierungen nacheinander mit allen definierten Parameterwerten in allen möglichen Kombinationen aus und misst die Laufzeit jedes einzelnen Durchlaufs. Zur besseren Stabilität der Messergebnisse wiederholt eine Testreihe jede Konfiguration der Eingabeparametern zwanzig mal. Alle im weiteren vorgestellten Ergebnisse sind arithmetische Mittel von zwanzig Durchläufen.

Die Wahl der Eingabeparameter ist im wesentlichen vom Algorithmus abhängig. Nur bei der verwendeten Threadanzahl verwenden wir grundsätzlich folgende Parameter: $\{1, 2, 3, 4, 6, 8, 12, 16\}$. Da unsere Hardware nur vier Rechenkerne zur Verfügung stellt werden wir keine Beschleunigung größer oder gleich Faktor vier messen. Wir testen aber bewusst mit einer höheren Anzahl Threads, um das Verhalten zu beobachten. Eine Testreihe wird über ein tc-shell Script gestartet.

Durch die Anzahl der Algorithmen, die Anzahl der Eingabeparameter, der jeweilige Anzahl definierter Eingabewerte und die zwanzigfache Wiederholung entsteht eine hohes Datenaufkommen. Dieses beherrschen wir durch die Verwendung der XML Sprache. Jede Testreihe erzeugt eine XML Datei. Das XML Schema einer Testreihe ist wie folgt definiert:

Code 5.1 (XML Schema einer Testreihe)**.**

```
<?xml version="1.0" encoding="utf-16"?>
<xsd:schema attributeFormDefault="unqualified"
  elementFormDefault="qualified" version="1.0"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="data" type="dataType" />
  <xsd:complexType name="dataType">
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" name="time" type="timeType" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="timeType">
    <xsd:attribute name="input" type="xsd:string" />
```

```
    <xsd:attribute name="thread" type="xsd:int" />
    <xsd:attribute name="implementation" type="xsd:string" />
    <xsd:attribute name="size" type="xsd:string" />
  </xsd:complexType>
</xsd:schema>
```

Jeder einzelne Durchlauf einer Implementierung wir durch ein XML Tag `<time>` beschrieben, das als Wert die Laufzeit enthält. In den Tags enthalten sind die Eingabeparameter: Threadanzahl im Tag `<thread>`, Eingabeanordnung im Tag `<input>`, Eingabegröße im Tag `<size>` und der Name der Implementierung im Tag `<implementation>`. Die strukturierte Ablage der Testergebnisse zahlt sich bei der Auswertung aus.

Den Frameworks und den implementierten Algorithmen liegt eine Referenz bei. Diese wurde mit dem Werkzeug Doxygen [77] generiert.

5.2. Auswertung

Die Auswertung der Testreihen erfolgt durch ein HTML Formular mit eingebettetem Javascript Code. Das Formular lädt dynamisch die XML Dateien der Versuche und wertet diese aus. Das Nachladen der XML Dateien erfolgt durch eine Ajax Anfrage unter Verwendung der Java Bibliothek Prototype [73]. Die XML Dateien werden entsprechend einer Konfiguration ausgewertet, die wiederum als XML Datei vorliegt. In der Konfigurationsdatei werden auszuwertende Parameter definiert und auf angezeigte Beschriftungen abgebildet. Das HTML Formular kann mit einem aktuellen Firefox Browser geöffnet werden. Im Folgenden ist ein Ausschnitt des XML Schemas zur Datenauswertung abgedruckt:

Code 5.2 (XML Schema zur Auswertung einer Testreihe).

```
<?xml version="1.0" encoding="utf-16"?>
<xsd:schema attributeFormDefault="unqualified"
  elementFormDefault="qualified" version="1.0"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="expsetup" type="expsetupType" />
  <xsd:complexType name="expsetupType">
    <xsd:sequence>
      <xsd:element name="inputs" type="inputsType" />
      <xsd:element name="threads" type="threadsType" />
      <xsd:element name="implementations" type="implementationsType" />
      <xsd:element name="sizes" type="sizesType" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="sizesType">
    <xsd:sequence>
```

```
      <xsd:element maxOccurs="unbounded" name="size" type="sizeType" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="sizeType">
    <xsd:attribute name="nr" type="xsd:string" />
    <xsd:attribute name="name" type="xsd:string" />
  </xsd:complexType>

  other types omitted ...
</xsd:schema>
```

Die Konfigurationsdatei definiert innerhalb der vier Tags `<inputs>` die Eingabeanordnungen, `<threads>` die Threadanzahlen, `<implementations>` die Implementierungen und `<sizes>` die Eingabegrößen die auszuwerten und mit einer passenden Beschriftung auszugeben sind. Beispielsweise werden innerhalb des Tags `<sizes>` einzelne `<size>` Tags aufgelistet, die jeweils zwei Attribute enthalten. Das Attribut ‘‘`nr`‘‘ entspricht dem Attribut `<size>` von `<time>` Tags der Datendatei. Das Attribut ‘‘`name`‘‘ definiert dessen angezeigte Beschriftung in der Ausgabe.

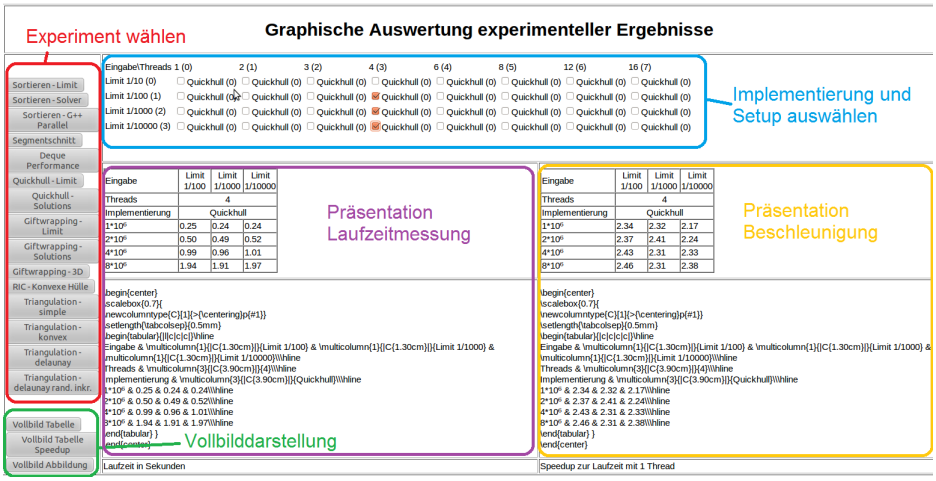


Abbildung 5.1.: Auswertung - HTML Maske & Tabellen; Rot: Auswahl der Testreihe; Blau: Implementierung nach Eingabegröße und Anzahl THreads; Violett: Auswertung der Laufzeiten; Orange: Auswertung der Beschelunigungsfaktoren; Grün: Vollbilddarstellung

Das HTML Formular wertet die Ergebnisse dynamisch aus. In Abbildung 5.1 ist das Formular abgebildet. Der Benutzer wählt erst aus dem Menü links (rot) die Testreihe. Danach werden im oberen Bereich (blau) die getesteten Implementierungen nach Anzahl der verwendeten Threads und Eingabegrößen aufgelistet. Zu den ausgewählten Implementierungen werden die Laufzeiten und die daraus resultierenden Beschleunigungsfaktoren ermittelt. Die Auswertungen der Laufzeiten (violett) und der Beschleunigungsfaktoren

(orange) werden sowohl tabellarisch als auch als Latex Code präsentiert. Die Tabellenspalten unterteilen sich in Ergebnisse zur Eingabeanordnung, Anzahl der Threads und Implementierung, in dieser Reihenfolge. Die Zeilen enthalten Ergebnisse zu wachsenden Eingabegrößen. Die Laufzeiten werden immer in Sekunden angegeben. Der Beschleunigungsfaktor bezieht sich immer auf die Laufzeit der identischen Implementierung, ausgeführt mit einem Thread. Unterhalb dieser Anzeige ist ein Diagramm eingeblendet, das die Beschleunigungsfaktoren graphisch visualisiert. Die angezeigten Tabellen und der Graph können im Vollbildmodus betrachtet werden. Dazu steht das Menü links unten (grün) zur Verfügung. Im Vollbildmodus können einfach durch die Druckfunktion des Browsers PDF Dateien erzeugt werden.

Das Diagramm visualisiert die Entwicklung der Beschleunigungsfaktoren in Abhängigkeit zur Eingabegröße, siehe Abbildung 5.2. Das Diagramm ist mit der Java Bibliothek Protovis [74] implementiert. Die Protovis Bibliothek ist mittlerweile in der Bibliothek D3 [75] aufgegangen. eine weitere Alternative ist die RGraph [76] Bibliothek. Grundsätzlich werden wir nur einen Ausschnitt der Daten hier veröffentlichen, da deren Menge zu groß ist. Die vollständigen Datensätze liegen auf dem dieser Arbeit zugehörigen Datenträger bei.

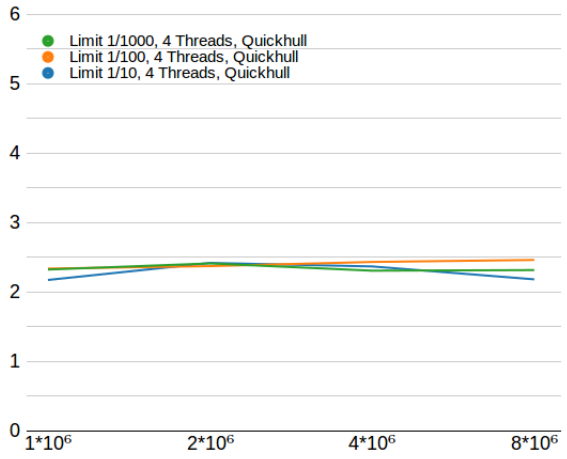


Abbildung 5.2.: Auswertung - Diagramm

Da die Experimente auf einem Rechner mit 4 Rechenkernen durchgeführt wurden, liegt die theoretische Maximalbeschleunigung durch Parallelisierung bei dem Faktor 4. Durch unvermeidliche Synchronisationsaufwand wird der gemessene Beschleunigungsfaktor darunter liegen. Auf Grund der Erfahrungswerte, die wir bei der Durchfürung der Experimente erlangt haben, nennen wir Beschleunigungsfaktoren größer 3,5 sehr gut, Faktoren größer 3 gut und Faktoren größer 2,5 befriedigend.

5.3. Datenstrukturen

In unserem ersten Experiment vergleichen wir die Leistung der Deque Implementierungen, die wir in Abschnitt 2.5 vorgestellt haben. Wir führen auf die Datenstrukturen jeweils eine bestimmte Anzahl n an Operationen aus. Eine Operation wird zufällig gewählt und ist eine `push()` oder `pop` Operation am Anfang (`front`) oder Ende (`back`) der Deque. Wir messen die Ausführungsdauer der n Operationen mit einer verschiedenen Anzahl an Threads. Die Ergebnisse sind in Tabelle 5.2 aufgeführt.

Eingabe	Zufallszahlen											
Threads	2			4			8			16		
Implementierung	Deque mit 2 Mutex	thread-basierte Deque	lock-freie Deque	Deque mit 2 Mutex	thread-basierte Deque	lock-freie Deque	Deque mit 2 Mutex	thread-basierte Deque	lock-freie Deque	Deque mit 2 Mutex	thread-basierte Deque	lock-freie Deque
$25*10^5$	0.35	1.55	0.43	0.26	1.96	0.43	0.22	0.47	0.43	0.22	0.42	0.43
$50*10^5$	0.32	1.60	0.43	0.26	1.99	0.42	0.23	0.61	0.43	0.22	0.46	0.42
$75*10^5$	0.33	1.62	0.43	0.26	2.08	0.42	0.23	0.73	0.42	0.22	0.55	0.42
$100*10^5$	0.34	1.68	0.42	0.26	2.11	0.42	0.23	0.71	0.43	0.22	0.54	0.42

Eingabe	Zufallszahlen											
Threads	2			4			8			16		
Implementierung	Deque mit 2 Mutex	thread-basierte Deque	lock-freie Deque	Deque mit 2 Mutex	thread-basierte Deque	lock-freie Deque	Deque mit 2 Mutex	thread-basierte Deque	lock-freie Deque	Deque mit 2 Mutex	thread-basierte Deque	lock-freie Deque
$25*10^5$	0.31	0.11	1.02	0.42	0.09	1.02	0.49	0.37	1.02	0.50	0.42	1.03
$50*10^5$	0.68	0.22	2.04	0.84	0.18	2.05	0.96	0.58	2.00	0.98	0.77	2.06
$75*10^5$	1.00	0.33	3.04	1.25	0.26	3.08	1.43	0.74	3.07	1.46	0.97	3.09
$100*10^5$	1.29	0.43	4.08	1.67	0.34	4.13	1.90	1.01	4.03	1.95	1.32	4.12

Tabelle 5.2.: Testergebnisse der Deque Implementierungen: Oben: Beschleunigungsfaktor nach Implementierungen und Eingabegrößen; Unten: Laufzeit nach Implementierungen und Eingabegrößen;

Wir vergleichen die naive Implementierung mit 2 Mutex Variablen (2.5.2.1), die threadbasierte Deque (2.5.2.2) und die lockfreie Deque (2.5.2.3). Die Anzahl der zufälligen Operationen ist ein Eingabeparameter aus: $\{25*10^5, 50*10^5, 75*10^5, 100*10^5\}$. In Tabelle 5.2 sind oben die Beschleunigungsfaktoren und unten die gemessenen Laufzeiten abgebildet.

Die naive Implementierung kann nicht überzeugen. Ihr Beschleunigungsfaktor bei 2 Threads liegt zwischen 0.32 und 0.35 was eine Verlangsamung bedeutet. Der Wert verschlechtert sich hin zu 16 Threads auf 0.22. Entsprechend kann man in Tabelle 5.2 unten eine Steigerung der Laufzeit mit der Erhöhung der Threadzahl ablesen. Immerhin belegt die naive Implementierung mit ihrer Laufzeit im gemessenen Rahmen den 2. Platz.

Die threadbasierte Deque ist die einzige Implementierung, die eine Beschleunigung erzielt. Bei 2 Threads liegt diese bei Faktor 1,55-1,68 bei 4 Threads hat sie den Höchststand mit Faktor 1,96-2,11. Mit mehr Threads sinkt die Leistung beachtlich und der Faktor ist wiederum kleiner 1. Ein Beschleunigungsfaktor von 2 bei 4 Threads ist nicht optimal, aber die threadbasierte Implementierung ist die Einzige, die eine Beschleunigung erzielt. Außerdem hat diese Implementierung in den Tests durchweg die besten Laufzeiten.

Die lockfreie Deque reagiert nicht auf die Erhöhung der Threadzahl größer 2. Beschleu-

nigung und Laufzeit bleiben bei Verwendung von mehr Threads konstant unverändert, bieten dabei aber bei einem Beschleunigungsfaktor von 0,42-0,43 nur einen Bruchteil der Leistung zur Ausführung mit einem Thread.

Die Testergebnisse liefern eine eindeutige Präferenz für die zu verwendende Implementierung. Leider konnte aber auch die effizienteste Implementierung, die threadbasierte Deque, mit ihrer Beschleunigung nicht vollständig überzeugen.

5.4. Divide and Conquer Experimente

Im Rahmen der Experimente zu Divide und Conquer Algorithmen untersuchen wir neben der Effizienz des Divide und Conquer Frameworks aus 3.3 noch folgende drei weitere Gesichtspunkte genauer:

1. Wir haben den Limit Paramter variiert und sein Optimum hinsichtlich der Eingabegröße ermittelt.

2. Wir haben die Wirkung verschiedener speziallisierter sequentieller Solver untersucht.

3. Wir haben die Leistung unserer Sortieralgorithmen mit der der MCSTL Bibliothek verglichen.

Der Limit Parameter definiert innerhalb unseres Frameworks die Größe ab der Jobs nicht mehr weiter aufgeteilt und parallel bearbeitet werden, sondern sequentiell gelöst werden müssen. Er entlasstet somit den Verteilungsmechanismus der Jobs indem er die Erzeugung einer großen Menge trivialer Jobs unterbindet. Wir werden in Teilabschnitt 5.4.1 die optimale Größe des Limitparameters hinsichtlich des Beschleunigungsfaktors ermitteln.

Wir möchten nicht nur die Effizienz unserer Arbeit anhand guter Beschleunigungsfaktoren belegen, sondern auch absolute Laufzeiten überprüfen. Hierzu vergleichen wir in Teilabschnitt 5.4.2 die Laufzeit unserer Quick- und Mergesort Implementierung mit der MCSTL Bibliothek [9].

Im Kontext der konvexe Hülle Algorithmen in Teilabschnitt 5.4.3 machen wir von der Möglichkeit gebrauch spezielle, effizientere sequentielle Solver statt des Standardsolvers zu verwenden. Die konvexe Hülle Algorithmen bieten sich für dieses Experiment an, da eine Fülle sehr unterschiedlicher Lösungen bekannt ist. Wie werden untersuchen, wie sich der Beschleunigungsfaktor bei unterschiedlichen Lösungen verhält und wie sich die Laufzeit beeinflußen lässt.

Wir beenden den Abschnitt mit Experimenten zu Triangulierungsalgorithmen 5.4.4 und einer Zusammenfassung unserer Ergebnisse 5.4.5.

5.4.1. Der Limit Parameter

Wir untersuchen das Verhalten der Laufzeit und des Beschleunigungsfaktors zweier Sortieralgorithmen und zweier konvexe Hülle Algorithmen hinsichtlich Ihrer Veränderung,

wenn wir den Limit Parameter in Relation zur Eingabegröße verändern. D.h. wir testen den Limit Parameter mit den Werten $1/10, 1/100, 1/1000$ und $1/10000$ in Relation zur Eingabegröße. Wir beginnen mit dem Quicksort und dem Mergesort Algorithmus (5.4.1.1) und fahren fort mit dem Giftwrapping und Quickhull Algorithmus (5.4.1.2).

5.4.1.1. Sortieralgorithmen

Die Sortieralgorithmen testen wir mit Zufallszahlen vom Typ Integer. Wir generieren Eingabemengen der Größen $10^6, 10^7, 10^8$ und 10^9. Den Limit Parameter geben wir immer in Relation zur Eingabegröße an. Wir beginnen mit der Quicksort Implementierung.

Eingabe	Limit 1/10			Limit 1/100			Limit 1/1000			Limit 1/10000		
Threads	4	8	16	4	8	16	4	8	16	4	8	16
Implementierung	Quicksort											
10^6	3.40	3.16	2.87	*3.86*	3.58	3.18	3.72	3.41	3.21	3.24	2.70	2.52
10^7	3.25	3.28	3.26	3.57	3.54	3.50	*3.58*	3.54	3.50	3.55	3.46	3.45
10^8	3.22	3.39	3.45	3.55	3.55	3.55	*3.60*	3.59	3.57	3.57	3.59	3.58
10^9	3.29	3.39	3.49	3.57	3.58	3.56	*3.60*	3.59	3.57	3.58	3.59	3.58

Eingabe	Limit 1/10			Limit 1/100			Limit 1/1000			Limit 1/10000		
Threads	4	8	16	4	8	16	4	8	16	4	8	16
Implementierung	Quicksort											
10^6	0.02	0.02	0.03	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.03	0.03
10^7	0.23	0.23	0.23	0.21	0.21	0.22	0.21	0.21	0.22	0.22	0.22	0.22
10^8	2.63	2.50	2.46	2.38	2.38	2.38	*2.35*	2.36	2.37	2.38	2.37	2.38
10^9	27.49	26.68	25.89	25.15	25.10	25.24	*24.92*	24.98	25.13	25.10	25.01	25.07

Tabelle 5.3.: Testergebnisse Limit Quicksort: Oben: Beschleunigungsfaktor nach Limits und Eingabegrößen; Unten: Laufzeit nach Limits und Eingabegrößen; Die besten Werte pro Eingabegröße sind kursiv

Die Ergebnisse zu den Quicksort Experimenten werden in Tabelle 5.3 aufgeführt. Die besten Werte pro Eingabegröße sind kursiv dargestellt. Wir erzielen durchweg gute bis sehr gute Beschleunigungsfaktoren, außer mit 10^6 Zufallszahlen fiel die Beschleunigung etwas geringer aus. Für die Eingabegröße 10^6 sind die absoluten Laufzeiten zu gering, um die Beschleunigung zuverlässig zu messen. Die Beschleunigungsfaktoren für die Eingabegrößen 10^7, 10^8 und 10^9 haben eine Spannweite von 3.22 bis 3.60. Die maximale Beschleunigung für diese Größen erzielen wir mit 4 Threads und dem Limit 1/1000.

Vergleicht man nur die Beschleunigungsfaktoren miteinander, die mit 8 beziehungsweise 16 Threads ermittelt wurden, so entstehen Konstellationen in denen wiederum das Limit 1/1000 die besten Beschleunigungsfaktoren liefert.

Die Laufzeittabelle gibt insgesamt ein ähnliches Bild ab wie die Beschleunigungstabelle. Die Laufzeiten mit dem Limit 1/100 liegen eng beieinander. Die schnellste Ausführung ist wiederum mit 4 Threads und einem Limit von 1/1000 gelungen.

Die Experimente zum Mergesort Algorithmus wurden analog zu den Quicksort Messungen durchgeführt und sind in Tabelle 5.4 aufgeführt. Der Mergesort Algorithmus erzielt überwiegend gute Beschleunigungsfaktoren, kommt aber an die Spitzenwerte des Quicksort Algorithmus leider nicht ran. Die Beschleunigungsfaktoren für Eingabegrößen größer

Eingabe	Limit 1/10			Limit 1/100			Limit 1/1000			Limit 1/10000		
Threads	4	8	16	4	8	16	4	8	16	4	8	16
Implementierung	Mergesort											
10^6	3.00	2.97	2.93	3.12	*3.14*	3.06	3.09	3.07	2.97	2.87	2.79	2.69
10^7	2.87	3.01	3.06	2.97	3.02	3.04	2.99	3.04	*3.06*	2.98	3.01	3.04
10^8	2.91	3.07	3.10	2.98	3.03	3.07	3.02	3.06	*3.09*	3.02	3.07	3.08
10^9	3.06	3.23	3.20	3.13	3.16	3.21	3.16	3.18	*3.22*	3.17	3.20	3.22

Eingabe	Limit 1/10			Limit 1/100			Limit 1/1000			Limit 1/10000		
Threads	4	8	16	4	8	16	4	8	16	4	8	16
Implementierung	Mergesort											
10^6	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.05
10^7	0.42	0.41	0.40	0.41	0.41	0.41	0.41	0.41	0.40	0.42	0.41	0.41
10^8	4.68	4.43	*4.38*	4.56	4.49	4.43	4.55	4.49	4.44	4.58	4.51	4.48
10^9	50.33	*47.66*	48.01	49.17	48.67	47.95	49.06	48.62	48.15	49.16	48.64	48.38

Tabelle 5.4.: Testergebnisse Limit Mergesort: Oben: Beschleunigungsfaktor nach Limits und Eingabegrößen; Unten: Laufzeit nach Limits und Eingabegrößen; Die besten Werte pro Eingabegröße sind kursiv

10^6 liegen mit einer Spanne von 2.87 bis 3.22 relativ nahe beieinander. Die Laufzeiten für die Größe 10^6 sind wiederum zu gering für verlässliche Beschleunigungsmessungen. Die maximalen Beschleunigungsfaktoren erzielen wir mit 16 Threads und einem Limit von 1/1000. Die Laufzeitmessung gibt dazu ein konsistentes Bild ab. Die Spitzenwerte hingegen bei einem Limit von 1/10 gemessen.

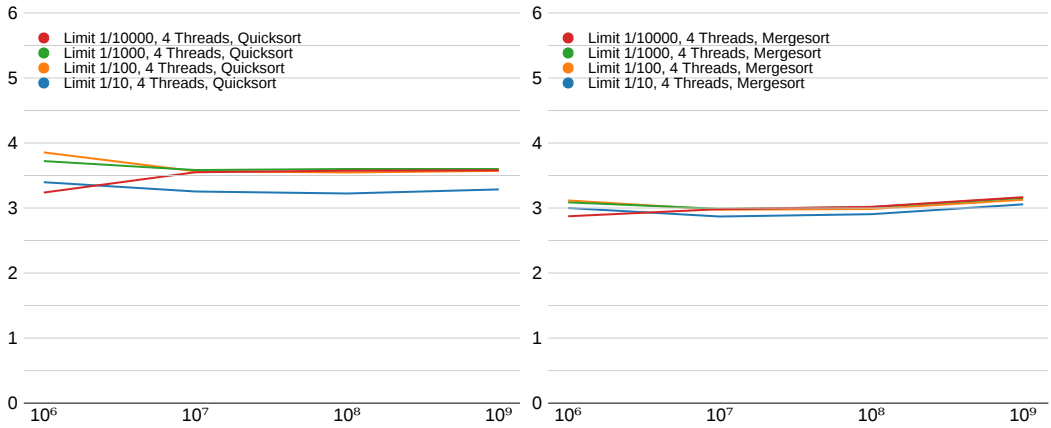


Abbildung 5.3.: Analyse Limit Sortieralgorithmen: X-Achse: Eingabegröße; Y-Achse: Beschleunigungsfaktor; Links: Auswertung des Quicksort Algorithmus mit 4 Threads; Rechts: Auswertung des Mergesort Algorithmus mit 4 Threads;

In Abbildung 5.3 sehen wir einen Vergleich der Beschleunigungsfaktoren verschiedener Limit Parameter mit 4 Threads, aufgeteilt nach Implementierungen. Es ist zu sehen, dass zwischen den Limits $1/100, 1/1000$ und $1/10000$ nur geringe Unterschiede liegen. Das Limit 1/10 ist leicht abgeschlagen.

5.4.1.2. Konvexe Hülle Algorithmen

Die konvexe Hülle Algorithmen testen wir mit zufallsgenerierten Punkten, die gleichmäßig in der Ebene verteilt sind. Wir generieren Eingabemengen der Größen $1*10^6, 2*10^6, 4*10^6$ und $8 * 10^6$. Den Limit Parameter geben wir wieder in Relation zur Eingabegröße an. Wir beginnen mit der Quickhull Implementierung.

Eingabe	Limit 1/10			Limit 1/100			Limit 1/1000			Limit 1/10000		
Threads	4	8	16	4	8	16	4	8	16	4	8	16
Implementierung	Quickhull											
$1*10^6$	2.17	2.16	2.23	2.34	2.34	*2.37*	2.32	2.30	2.25	2.17	2.15	2.18
$2*10^6$	2.42	2.56	*2.63*	2.37	2.45	2.46	2.41	2.40	2.36	2.24	2.37	2.44
$4*10^6$	2.37	2.43	2.46	2.43	*2.61*	2.51	2.31	2.35	2.31	2.33	2.52	2.49
$8*10^6$	2.18	2.40	2.35	2.46	2.46	*2.56*	2.31	2.32	2.42	2.38	2.52	2.55

Eingabe	Limit 1/10			Limit 1/100			Limit 1/1000			Limit 1/10000		
Threads	4	8	16	4	8	16	4	8	16	4	8	16
Implementierung	Quickhull											
$1*10^6$	0.25	0.25	0.24	0.25	0.25	0.25	0.24	0.25	0.25	0.24	0.24	0.24
$2*10^6$	0.52	0.50	0.48	0.50	0.49	0.48	0.49	0.49	0.50	0.52	0.49	0.48
$4*10^6$	0.99	0.97	0.95	0.99	*0.92*	0.95	0.96	0.94	0.96	1.01	0.94	0.95
$8*10^6$	2.06	1.88	1.92	1.94	1.94	1.86	1.91	1.91	*1.83*	1.97	1.86	*1.83*

Tabelle 5.5.: Testergebnisse Limit Quickhull: Oben: Beschleunigungsfaktor nach Limits und Eingabegrößen; Unten: Laufzeit nach Limits und Eingabegrößen; Die besten Werte pro Eingabegröße sind kursiv

Die Ergebnisse zu den Quickhull Experimenten sind in Tabelle 5.5 aufgeführt. Die besten Werte pro Eingabegröße sind kursiv dargestellt. Die ermittelten Beschleunigungsfaktoren sind befriedigend, ihre Spanne reicht dabei von 2.15 bis 2.63. Die erreichten Bestwerte pro Eingabegröße sind dabei über verschiedene Konfigurationen verteilt, wobei das Limit 1/100 3 Bestwerte vereint. Auch die Laufzeittabelle gibt kein klares Bild ab. So liegen die Laufzeiten bei den Eingabegrößen 10^6 und 10^7 nahe beieinander, bei den Größen 10^8 und 10^9 erreichen die Limits 1/100, 1/1000 und 1/10000 je einen Spitzenwert.

Die Ergebnisse zu den Gidtwrapping Experimenten sind in Tabelle 5.6 aufgeführt. Wir erzielen überwiegend gute Beschleunigungsfaktoren, außer für das Limit 1/10, wo die Werte befriedigend ausfallen. Die Beschleunigungsfaktoren haben eine Spannweite von 2.71 bis 3.28. Mit 4 Threads und dem Limit 1/1000 werden durchgängig die besten Beschleunigungen erzielt. Mit der Konfiguration 4 Threads und Limit 1/1000 erreichen wir auch die besten Laufzeiten.

Auch wenn wir nur die Beschleunigungsfaktoren miteinander vergleichen, die wir mit 8 bzw. 16 Threads berechnet haben, so können wir wieder die besten Beschleunigungsfaktoren bei dem Limit 1/1000 ablesen.

In Abbildung 5.4 sehen wir einen Vergleich der Beschleunigungsfaktoren verschiedener Limit Parameter mit 4 Threads, aufgeteilt nach Implementierungen. Aus der Abbildung zum Quickhull Algorithmus lässt sich keine Tendenz ablesen, zu unterschiedlich sind die Entwicklungen. Zum Giftwrapping lässt sich feststellen, dass die Unterschiede zwischen den Limits 1/100, 1/1000 und 1/10000 gering ausfallen, das Limit 1/10 ist hingegen schon

Eingabe	Limit 1/10			Limit 1/100			Limit 1/1000			Limit 1/10000		
Threads	4	8	16	4	8	16	4	8	16	4	8	16
Implementierung	Gift Wrapping											
$1*10^6$	2.98	3.00	2.71	*3.26*	3.15	2.97	*3.26*	3.15	3.00	3.21	3.09	2.99
$2*10^6$	2.96	2.93	2.80	3.24	3.15	2.99	*3.28*	3.18	3.03	3.23	3.15	3.01
$4*10^6$	2.98	2.96	2.80	3.25	3.14	2.96	*3.26*	3.18	3.04	3.23	3.15	3.04
$8*10^6$	2.99	2.98	2.84	3.24	3.13	2.95	*3.26*	3.15	3.02	*3.26*	3.15	3.04

Eingabe	Limit 1/10			Limit 1/100			Limit 1/1000			Limit 1/10000		
Threads	4	8	16	4	8	16	4	8	16	4	8	16
Implementierung	Gift Wrapping											
$1*10^6$	0.63	0.62	0.69	0.58	0.59	0.63	*0.57*	0.59	0.62	0.59	0.61	0.63
$2*10^6$	1.34	1.36	1.42	1.23	1.26	1.33	*1.21*	1.25	1.31	1.23	1.27	1.33
$4*10^6$	2.84	2.87	3.03	2.60	2.70	2.87	*2.59*	2.66	2.78	2.62	2.69	2.78
$8*10^6$	6.06	6.07	6.39	5.58	5.76	6.12	5.54	5.73	5.98	*5.53*	5.71	5.92

Tabelle 5.6.: Testergebnisse Limit Giftwrapping: Oben: Beschleunigungsfaktor nach Limits und Eingabegrößen; Unten: Laufzeit nach Limits und Eingabegrößen; Die besten Werte pro Eingabegröße sind kursiv

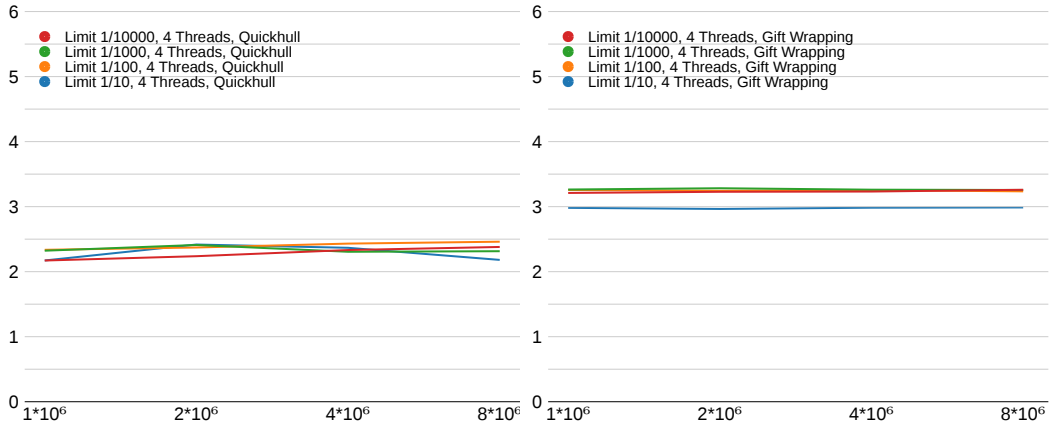


Abbildung 5.4.: Analyse Limit konvexe Hülle Algorithmen: X-Achse: Eingabegröße; Y-Achse: Beschleunigungsfaktor; Links: Auswertung des Quickhull Algorithmus mit 4 Threads; Rechts: Auswertung des Giftwrapping Algorithmus mit 4 Threads;

leicht abgeschlagen.

5.4.1.3. Zusammenfassung

Das Limit 1/1000 zur Eingabegröße erzielte in unseren Experimenten die meisten Höchstwerte bei den Beschleunigungsfaktoren, insbesondere bei den Implementierungen Quicksort, Mergesort und Giftwrapping. Quickhull bildet die Ausnahme. Auch wenn die Beschleunigungsfaktoren der Limits 1/100 und 1/10000 nahe dran liegen, lässt sich aus den Tabellen fast überwiegend ein Vorteil für das Limit 1/1000 erkennen. Der Vorteil für das Limit 1/1000 ist nicht nur an der Anzahl der Spitzenwerte erkennbar, sondern

auch, wenn man die Beschleunigungsfaktoren einer gegeben Anzahl Threads vergleicht. Im weiteren werden wir für alle Divide und Conquer Experimente ein Limit von 1/1000 zur Eingabegröße verwenden.

5.4.2. Sortieralgorithmen

Die Sortieralgorithmen testen wir mit zufällig generierten Mengen von Integern. Die Eingabemengen haben eine Größe von 10^6, 10^7, 10^8 und 10^9. Wir vergleichen zunächst unserer Implementierungen des Quicksort (Abschnitt 3.4.2) und des Mergesort Algorithmus (Abschnitt 3.4.1) mit den Implementierungen aus der MCSTL Bibliothek [9]. Danach Untersuchen wir die Wirkung verschiedener Solver und des parallelen Partitionierens.

5.4.2.1. Leistung Sortieralgorithmen und Vergleich MCSTL

In Tabelle 5.7 sind die Ergebnisse unserer Experimente abgebildet. Unsere Quicksort Implementierung liefert in den dargestellten Experimenten überwiegend sehr gute Beschleunigungsfaktoren, wogegen unsere Mergesort Implementierung 'nur' gute Beschleunigungen zeigt. Zudem fällt im Vergleich zwischen den beiden Implementierung auf, dass die tatsächliche Laufzeit des Mergesort Algorithmus um circa Faktor 2 schlechter ist als die des Quicksort Algorithmus.

Eingabe	Zufallszahlen											
Threads	4				8				16			
Implementierung	Quick-sort	Merge-sort	Quick-sort - MCSTL	Merge-sort - MCSTL	Quick-sort	Merge-sort	Quick-sort - MCSTL	Merge-sort - MCSTL	Quick-sort	Merge-sort	Quick-sort - MCSTL	Merge-sort - MCSTL
10^6	*3.75*	3.17	1.64	3.38	3.51	3.17	1.55	1.63	3.08	3.10	1.45	1.45
10^7	*3.58*	2.98	1.77	3.43	*3.57*	3.05	1.77	2.33	3.51	3.08	1.70	2.10
10^8	*3.58*	3.03	1.76	3.48	*3.58*	3.07	1.70	2.97	3.56	3.11	1.73	2.96
10^9	*3.59*	3.18	1.81	3.52	*3.59*	3.20	1.71	3.20	3.57	3.24	1.70	3.18

Eingabe	Zufallszahlen											
Threads	4				8				16			
Implementierung	Quick-sort	Merge-sort	Quick-sort - MCSTL	Merge-sort - MCSTL	Quick-sort	Merge-sort	Quick-sort - MCSTL	Merge-sort - MCSTL	Quick-sort	Merge-sort	Quick-sort - MCSTL	Merge-sort - MCSTL
10^6	0.02	0.04	0.04	0.02	0.02	0.04	0.04	0.04	0.02	0.04	0.04	0.04
10^7	0.21	0.43	0.41	0.21	0.21	0.42	0.41	0.31	0.21	0.42	0.43	0.35
10^8	*2.33*	4.69	4.73	2.39	*2.33*	4.63	4.89	2.80	2.34	4.58	4.82	2.81
10^9	*24.75*	50.94	49.90	25.64	*24.76*	50.50	52.66	28.14	24.94	49.90	52.99	28.35

Tabelle 5.7.: Vergleich Quicksort MCSTL: Oben: Beschleunigungsfaktor nach Implementierungen und Eingabegrößen; Unten: Laufzeit nach Implementierungen und Eingabegrößen;

Im Gegensatz dazu liefert die MCSTL Quicksort Implementierung in keiner Konfiguration befriedigenden Ergebnisse. Die MCSTL Mergesort Implementierung zeigt bei einer Ausführung mit 4 Threads gute Beschleunigungswerte. Mit 8, bzw. 16 Threads nimmt die Implementierung erst bei einer Eingabegröße von 10^9 gute Werte, darunter fällt der Leistungsgewinn drastisch ab. Auch zwischen den Laufzeiten der MCSTL

Implementierungen liegt ein Unterschied von Faktor 2, hier zugunsten der Mergesort Implementierung.

Im Vergleich zwischen unserer und der MCSTL Implementierung sind unserer Quicksort und die MCSTL Mergsort Implementierung auf Augenhöhe bezüglich Beschleunigung und Laufzeit. Die anderen beiden Implementierungen sind abgeschlagen. Allerdings hat unserer Quicksort Implementierung immer die bessere Beschleunigung, insbesondere bei der Verwendung von 8 und 16 Threads, und bei großen Eingabemengen auch die deutlich kürzere Laufzeit, z.B. 24.75 zu 25.64 Sekunden bei 4 Threads und 10^9 Integern.

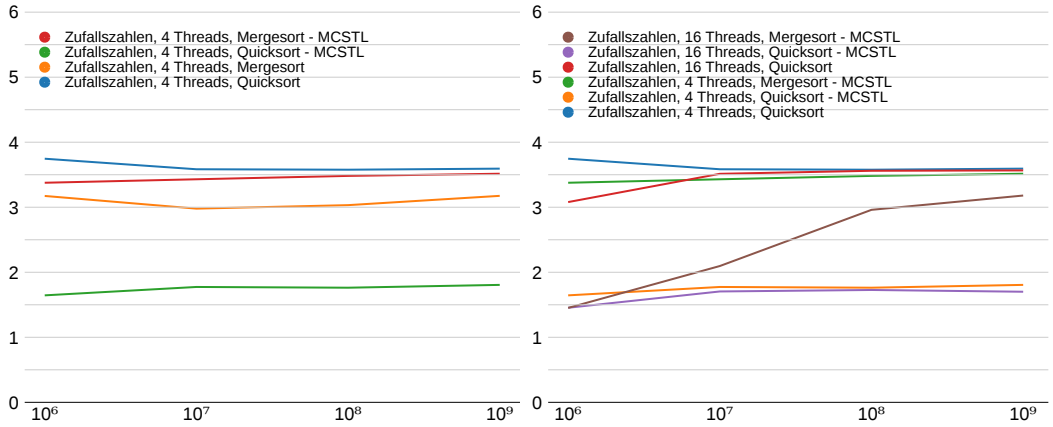


Abbildung 5.5.: Vergleich mit MCSTL: X-Achse: Eingabegröße; Y-Achse: Beschleunigungsfaktor; Links: Vergleich Sortieralgorithmen mit 4 Threads; Rechts: Vergleich Quicksort mit MCSTL mit 4 und 16 Threads;

In Abbildung 5.5 sehen wir den Vergleich aus einer anderen Perspektive. Im linken Diagramm sehen wir die Beschleunigung der verschiedenen Implementierungen mit 4 Threads. Dort liegt unserer Quicksort Implementierung an der Spitze, gefolgt von MCSTL Mergesort. Im rechten Diagramm ist der Vergleich zwischen Quicksort und den MCSTL Implementierungen mit 4 und 16 Threads abgebildet. Bis auf MCSTL Mergesort mit 4 Threads sind die MCSTL Implementierungen teils deutlich abgeschlagen.

5.4.2.2. Standardsolver und paralleles Partitionieren

In Tabelle 5.8 vergleichen wir die Auswirkungen der parallelen Partitionierung aus Abschnitt 2.6.2 und die Anwendung eines speziellen Solvers anstelle des Standardsolvers auf Leistung der Quicksort Implementierung. Die Quicksort Implementierung, die auf beide Optimierungen verzichtet, bezeichnen wir mit 'St. Sol. + seq. Job' die Implementierung 'Standard Sol.' bezieht die parallele Partitionierung mit ein. Die Implementierung 'spezieller Sol.' stellt die höchste Ausbaustufe dar, die wir üblicherweise verwenden.

Betrachten wir zunächst die Tabelle 5.8 oben mit den Beschleunigungsfaktoren. Zwischen den Beschleunigungsfaktoren der naiven Variante ('St. Sol. + seq. Job') und der Variante mit paralleler Partitionierung ('Standard Sol.') besteht für die Eingabegrößen 10^7 bis 10^9 eine Differenz von 0.25 bis 0.35 zugunsten der parallelen Partitionierung.

Eingabe	Zufallszahlen								
Threads	4			8			16		
Implementierung	spezieller Sol.	Standard Sol.	St. Sol. + seq. Job	spezieller Sol.	Standard Sol.	St. Sol. + seq. Job	spezieller Sol.	Standard Sol.	St. Sol. + seq. Job
10^6	3.73	3.71	2.97	3.41	3.45	2.87	3.23	3.26	2.75
10^7	3.57	3.60	3.25	3.56	3.59	3.22	3.51	3.56	3.21
10^8	3.60	3.61	3.34	3.60	3.60	3.34	3.57	3.59	3.33
10^9	3.59	3.60	3.32	3.58	3.60	3.35	3.57	3.58	3.32

Eingabe	Zufallszahlen								
Threads	4			8			16		
Implementierung	spezieller Sol.	Standard Sol.	St. Sol. + seq. Job	spezieller Sol.	Standard Sol.	St. Sol. + seq. Job	spezieller Sol.	Standard Sol.	St. Sol. + seq. Job
10^6	0.02	0.02	0.03	0.02	0.02	0.03	0.02	0.03	0.03
10^7	0.21	0.25	0.28	0.21	0.25	0.28	0.21	0.25	0.28
10^8	2.33	2.73	2.95	2.33	2.73	2.95	2.34	2.75	2.96
10^9	24.63	28.17	30.59	24.68	28.20	30.27	24.77	28.31	30.54

Tabelle 5.8.: Vergleich Quicksort Solver und paralleles Partitionieren: Oben: Beschleunigungsfaktor nach Implementierungen und Eingabegrößen; Unten: Laufzeit nach Implementierungen und Eingabegrößen;

Die Verwendung des speziellen Solvers im Vergleich zum Standardsolver bringt hingegen keine weitere Verbesserung des Beschleunigungsfaktors. Die Werte sind nahezu identisch.

Eine Auswertung der Laufzeittabelle 5.8 unten ergibt, dass sich die bessere Beschleunigung durch die Benutzung der parallelen Partitionierung analog auf die Laufzeit auswirkt. Weiter zeigt sich, dass die Verwendung des speziellen Solvers zwar keine höhere Beschleunigung einbringt, aber die absoluten Laufzeiten erheblich besser sind. Unter Beachtung der Eingabegrößen 10^7 bis 10^9 ist die Implementierung mit speziellem Solver um circa 15% schneller als die Implementierung mit Standardsolver und sogar um circa 20% schneller als die naive Variante.

In Abbildung 5.6 werden die Beschleunigungsfaktoren der verschiedenen Varianten nochmals verglichen. Zu erkennen ist, dass die naive Variante zwar gute Werte erzielt, aber dennoch zu den anderen beiden Varianten mit sehr guten Werten abgeschlagen ist.

5.4.2.3. Zusammenfassung

Der Vergleich mit MCSTL hat zum einen gezeigt, dass unser Divide und Conquer Framework in der Lage ist bei effizienter Implementierung der Jobs kompetitive Laufzeiten zu erzeugen. Und zum anderen, was viel wichtiger ist, die Eigenschaft durchweg gute bis sehr gute Beschleunigungen zu erzielen, auch wenn mehr Threads verwendet werden als Rechenkerne vorhanden sind und auch wenn die konkrete Implementierungen, wie unsere Mergesort Implementierung, nicht die besten Laufzeiten erreicht.

Weiter haben wir gezeigt, dass sich durch zwei Maßnahmen die Effizienz des Frameworks weiter steigern lässt. Die Verwendung paralleler Funktionen in den einzelnen Jobs erhöht nochmals den Beschleunigungsfaktor. Das Ersetzen des sequentiellen Standard-

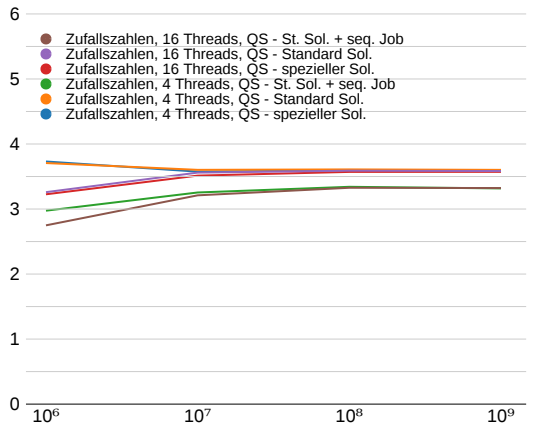


Abbildung 5.6.: Vergleich Quicksort Solver: X-Achse: Eingabegröße; Y-Achse: Beschleunigungsfaktor; Vergleich der Solver des Quickhull Algorithmus mit 4 und 16 Threads;

solvers durch eine spezielle, dem Algorithmus angepasste, Variante kann die Laufzeit erheblich verbessern. Das Framework bietet ein hohes Maß an Flexibilität um verschiedene Leistungsreserven auszuschöpfen.

5.4.3. Konvexe Hülle Algorithmen

Wir haben in Abschnitt 3.5.1 den Giftwrapping Algorithmus vorgestellt und in Abschnitt 3.5.2 den Quickhull Algorithmus, deren experimentelle Ergebnisse wir hier besprechen. Wir beginnen mit dem Giftwrapping Algorithmus in Abschnitt 5.4.3.1, danach folgt Quickhull in Abschnitt 5.4.3.2 und weitere Varianten des Quickhull Algorithmus in Abschnitt 5.4.3.3. Wir vergleichen die speziellen, sequentiellen Solver mit dem Standardsolver und untersuchen die Auswirkungen des parallelen Partitionierens beider Implementierungen in Abschnitt 5.4.3.4. Die Implementierungen berechnen die konvexe Hülle in 2D. Im letzten Abschnitt 5.4.3.5 präsentieren wir auch die Ergebnisse für die Giftwrapping Implementierung in 3D.

Aus früheren Experimenten wissen wir, dass die verschiedenen konvexe Hülle Algorithmen auf bestimmte Anordnungen der Eingabepunkte sehr unterschiedlich reagieren. Wir werden daher die Experimente mit folgenden Eingabeanordungen durchführen:

- "Punkte im Quadrat": zufallsverteilte Punkte in einem Quadrat. Sehr kleine konvexe Hülle.
- "Punkte in Kreisnähe": zufallsverteilte Punkte in relativer Nähe zu einem Kreis. Größere Hülle.
- "Punkte auf Kreis": Die Punkte liegen mit gegebener Genauigkeit auf einem Kreis. Annähernd alle Punkte sind Teil der Hülle.

Die Ursache für das unterschiedliche Laufzeitverhalten der Algorithmen beruht auf der Größe der resultierenden konvexen Hülle. Die Ausgabegröße hat bezüglich der asymptotischen Laufzeit bei keinem der vorgestellten Algorithmen einen Einfluss, auf die gemessenen Laufzeiten hingegen schon. Des weiteren variieren wir, wie schon bekannt, die Anzahl der Threads und die Eingabegrößen. Wir verwenden Punktmengen der Mächtigkeit $1 * 10^6$, $2 * 10^6$, $4 * 10^6$ und $8 * 10^6$.

5.4.3.1. Giftwrapping Lösungen

Wir haben den Job, der den Giftwraping Alogrithmus implementiert, in Abschnitt 3.5.1 vorgestellt. Diesen Job kombinieren wir mit verschiedenen sequentiellen Solverlösungen mit dem vorrangigen Ziel die absolute Laufzeit zu optimieren, wobei wir die Beschleunigungsfaktoren nicht außer Acht lassen. Als sequentielle Solver verwenden wir den Giftwrapping Algorithmus, den Grahams Scan und den randomisiert inkrementellen Algorithmus.

Eingabe	Punkte im Quadrat			Punkte in Kreisnähe			Punkte auf Kreis		
Threads	4								
Implementierung	Gift Wrap-ping	Graham Scan	random-isiert Inkre-mentell	Gift Wrap-ping	Graham Scan	random-isiert Inkre-mentell	Gift Wrap-ping	Graham Scan	random-isiert Inkre-mentell
$1*10^6$	*3.26*	3.10	2.84	*3.38*	3.28	3.25	3.34	3.30	*3.40*
$2*10^6$	*3.26*	3.09	2.78	*3.41*	3.31	3.24	3.38	3.33	*3.44*
$4*10^6$	*3.24*	3.07	2.77	*3.42*	3.30	3.24	3.34	3.25	*3.41*
$8*10^6$	*3.23*	2.97	2.74	*3.44*	3.27	3.22	*3.29*	3.17	3.27

Eingabe	Punkte im Quadrat			Punkte in Kreisnähe			Punkte auf Kreis		
Threads	4								
Implementierung	Gift Wrap-ping	Graham Scan	random-isiert Inkre-mentell	Gift Wrap-ping	Graham Scan	random-isiert Inkre-mentell	Gift Wrap-ping	Graham Scan	random-isiert Inkre-mentell
$1*10^6$	0.57	0.42	*0.30*	0.91	0.65	*0.59*	1.05	*0.87*	1.59
$2*10^6$	1.21	0.90	*0.64*	2.25	1.53	*1.34*	2.22	*1.85*	4.50
$4*10^6$	2.60	1.98	*1.36*	5.43	3.53	*3.00*	4.71	*4.03*	12.22
$8*10^6$	5.55	4.44	*2.96*	12.44	7.84	*6.58*	9.98	*8.67*	31.61

Tabelle 5.9.: Vergleich verschiedener Giftwrapping Lösungen: Die Giftwrapping Jobs werden von verschiedenen sequentiell Solvern gelöst. Diese sind in der Reihe "Implementierung" aufgeführt; Die Threadanzahl ist 4; Die Eingabeanordnung variiert; Oben: Beschleunigungsfaktor nach Implementierungen und Eingabegrößen; Unten: Laufzeit nach Implementierungen und Eingabegrößen;

In Tabelle 5.9 überprüfen wir das Verhalten der Solver bei verschiedenen Eingabeanordnungen und 4 Threads. Die besten Ergebnisse pro Eingabeanordnung und Eingabegröße sind kursiv dargestellt.

Die sequentiellen Solver Giftwrapping und Graham Scan liefern ausschließlich gute Beschleunigungsfaktoren; der randomisiert inkrementelle Solver hingegen beschleunigt nur mit der Punkteanordnung auf dem Kreis gut, sonst nur befriedigend. Für die Eingabean-

ordnungen “Punkte im Quadrat” und “Punkte in Kreisnähe” beschleunigt der Giftwrapping Solver am schnellsten, für “Punkte auf Kreis” gewinnt der randomisiert inkrementelle Solver.

Die absoluten Laufzeiten liefern ein fast gegensätzliches Bild ab. Die beste Leistung für die Anordnungen “Punkte im Quadrat” und “Punkte in Kreisnähe” zeigt der randomisiert inkrementelle Solver, für “Punkte auf Kreis” gewinnt der Grahams Scan. Der Giftwrapping Solver enttäuscht in jeder Kategorie. Der Graham Scan zeigt eine durchweg solide Leistung und der randomisiert inkrementelle Solver ist für zwei Anordnungen stark, fällt für “Punkte in Kreisnähe” jedoch drastisch ab.

Besonders interessant ist der Vergleich zwischen dem Giftwrapping Solver und den beiden anderen. Dieser zeigt, dass sich durch die Kombination eines Divide und Conquer Algorithmus mit sequentiellen Solvern, die völlig andere Algorithmen implementieren, sowohl bessere Beschleunigungsfaktoren als auch bessere Laufzeiten erzielen lassen.

Eingabe	Punkte im Quadrat								
Threads	4			8			16		
Implementierung	Gift Wrap-ping	Graham Scan	random-isiert Inkre-mentell	Gift Wrap-ping	Graham Scan	random-isiert Inkre-mentell	Gift Wrap-ping	Graham Scan	random-isiert Inkre-mentell
$1*10^6$	3.26	3.10	2.84	3.14	2.98	2.70	2.99	2.81	2.41
$2*10^6$	3.26	3.09	2.78	3.16	3.01	2.63	3.03	2.83	2.43
$4*10^6$	3.24	3.07	2.77	3.17	2.98	2.68	3.02	2.85	2.42
$8*10^6$	3.23	2.97	2.74	3.14	2.87	2.64	2.99	2.73	2.46

Eingabe	Punkte im Quadrat								
Threads	4			8			16		
Implementierung	Gift Wrap-ping	Graham Scan	random-isiert Inkre-mentell	Gift Wrap-ping	Graham Scan	random-isiert Inkre-mentell	Gift Wrap-ping	Graham Scan	random-isiert Inkre-mentell
$1*10^6$	0.57	0.42	0.30	0.59	0.43	0.32	0.62	0.46	0.36
$2*10^6$	1.21	0.90	0.64	1.25	0.93	0.67	1.30	0.99	0.73
$4*10^6$	2.60	1.98	1.36	2.66	2.04	1.41	2.79	2.14	1.56
$8*10^6$	5.55	4.44	2.96	5.72	4.60	3.06	6.00	4.83	3.29

Tabelle 5.10.: Vergleich verschiedener Giftwrapping Lösungen: Die Giftwrapping Jobs werden von verschiedenen sequentiell Solvern gelöst. Diese sind in der Reihe ”Implementierung” aufgeführt; Die Threadanzahl variiert; Die Punkte sind im Quadrat gleichverteilt; Oben: Beschleunigungsfaktor nach Implementierungen und Eingabegrößen; Unten: Laufzeit nach Implementierungen und Eingabegrößen;

In Tabelle 5.10 untersuchen wir die Auswirkung der Anzahl verwendeter Threads auf Beschleunigung und Laufzeit. Wir ziehen hierzu nur die Anordnung “Punkte im Quadrat” heran.

Zieht man die Ergebnisse eines beliebigen Solvers heran, zeigt sich, dass dessen Leistung mit Zunahme der Threads abnimmt. Dieses Bild ergibt sich für jeden Solver sowohl bezüglich der Beschleunigungsfaktoren als auch der Laufzeiten. Eine Erhöhung der Threadanzahl über die Anzahl der verfügbaren Rechenkerne ist hier also kontraproduktiv.

In Abbildung 5.7 vergleichen wir nochmals die Beschleunigungsfaktoren des Giftwrap-

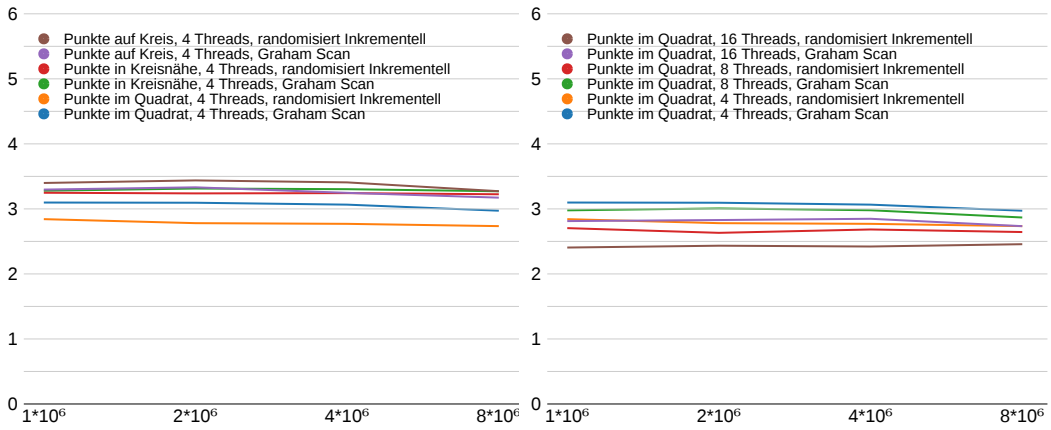


Abbildung 5.7.: Analyse Giftwrapping Solver: X-Achse: Eingabegröße; Y-Achse: Beschleunigungsfaktor; Links: Auswertung des Giftwrapping Algorithmus mit den Solvern Graham Scan und randomisiert Inkrementell für 4 Threads und verschiedenen Eingaben; Rechts: Auswertung des Giftwrapping Algorithmus mit den Solvern Graham Scan und randomisiert Inkrementell für "Punkte im Quadrat" und 4,8,16 Threads;

ping Algorithmus mit den Solvern "randomisiert inkrementell" und "Grahma Scan". Auf der linken Seite sehen wir den Vergleich der Solver bei verschiedenen Eingabeanordnungen. Auf der rechten Seite sehen wir den Einfluss einer wachsenden Anzahl von Threads auf die Beschleunigungsfaktoren. Die Ergebnisse sind oben beschrieben.

5.4.3.2. Quickhull Lösungen

Den Quickhull Job haben wir in Abschnitt 3.5.2 beschrieben. So wie wir die Solver des Giftwrapping Algorithmus im vorigen Teilabschnitt verglichen haben, gehen wir nun auch bei den Quickhull Lösungen vor. Als sequentielle Solver verwenden wir den Quickhull Algorithmus, den Grahams Scan und den randomisiert inkrementellen Algorithmus.

In Tabelle 5.11 überprüfen wir das Verhalten der Solver bei verschiedenen Eingabeanordnungen und 4 Threads. Falls klar zu ermitteln, sind die besten Ergebnisse pro Eingabeanordnung und Eingabegröße kursiv dargestellt.

Bei der Betrachtung der Beschleunigungsfaktoren fällt direkt auf, dass die verschiedenen Solver die Beschleunigung für die Anordnung "Punkte im Quadrat" nicht und für "Punkte in Kreisnähe" wenig beeinflussen. Für die Anordnung "Punkte auf Kreis" zeigt der Quickhull Solver die beste Leistung. Die Beschleunigungsfaktoren für die Anordnung "Punkte im Quadrat" sind zudem nur befriedigend, für die restlichen Anordnungen messen wir eine gute Beschleunigung.

Da die sequentiellen Solver auf die Beschleunigung für "Punkte im Quadrat" offenbar keinen Einfluss haben, dominiert das Teilen der Jobs das Verhalten des Algorithmus. Das Verhalten ist durch die Konstruktion des Quickhull Algorithmus erklärbar. Beim Teilen eines Quickhull Jobs werden alle Punkte in 3 Teilmengen eingeteilt. Davon wer-

Eingabe	Punkte im Quadrat			Punkte in Kreisnähe			Punkte auf Kreis		
Threads	4								
Implementierung	Quick-hull	Graham Scan	random-isiert Inkre-mentell	Quick-hull	Graham Scan	random-isiert Inkre-mentell	Quick-hull	Graham Scan	random-isiert Inkre-mentell
$1*10^6$	2.32	2.33	2.34	3.24	3.27	3.23	3.30	3.29	3.34
$2*10^6$	2.22	2.22	2.22	3.26	3.27	3.23	3.36	3.30	3.35
$4*10^6$	2.16	2.17	2.17	3.29	3.28	3.24	*3.40*	3.28	3.30
$8*10^6$	2.09	2.09	2.09	3.28	3.30	3.27	*3.41*	3.31	3.23

Eingabe	Punkte im Quadrat			Punkte in Kreisnähe			Punkte auf Kreis		
Threads	4								
Implementierung	Quick-hull	Graham Scan	random-isiert Inkre-mentell	Quick-hull	Graham Scan	random-isiert Inkre-mentell	Quick-hull	Graham Scan	random-isiert Inkre-mentell
$1*10^6$	0.25	0.24	0.24	1.75	1.25	*1.19*	4.15	*1.51*	2.23
$2*10^6$	0.49	0.49	0.48	3.19	2.80	*2.61*	9.89	*3.21*	5.93
$4*10^6$	1.01	1.01	1.01	6.07	6.20	*5.67*	23.32	*6.82*	15.31
$8*10^6$	2.04	2.04	2.04	*12.08*	13.36	12.21	54.33	*14.30*	38.33

Tabelle 5.11.: Vergleich verschiedener Quickhull Lösungen: Die Quickhull Jobs werden von verschiedenen sequentiell Solvern gelöst. Diese sind in der Reihe "Implementierung" aufgeführt; Die Threadanzahl ist 4; Die Eingabeanordnung variiert; Oben: Beschleunigungsfaktor nach Implementierungen und Eingabegrößen; Unten: Laufzeit nach Implementierungen und Eingabegrößen;

den 2 in der Rekursion verwendet, die 3. Teilmenge wird verworfen, da ihre Punkte im Inneren der Hülle liegen. Bei der Anordnung "Punkte im Quadrat" liegen die allermeisten Punkte im Inneren und werden sehr schnell ausgefiltert. Es werden insgesamt nur wenige Jobs angelegt und der Rekursionsbaum des Quickhull Algorithmus degeneriert. Da unsere Parallelisierung primär auf der parallelen Bearbeitung von Jobs basiert, kann unser Framework seine Stärken hier nicht ausspielen. Auch das parallele Ausführen der Partitionierungsfunktion während des Teilens der Jobs hat nicht zu guten Ergebnissen geführt. Den Effekt der parallelen Partitionierung untersuchen wir detailliert in Teilabschnitt (5.4.3.3).

Die Laufzeitmessung zeigt den gleichen Effekt. Es gibt keine Unterschiede für die Anordnung "Punkte im Quadrat". Für "Punkte in Kreisnähe" kann der randomisiert inkrementelle Solver Vorteile verbuchen. Den Vergleich für "Punkte auf Kreis" gewinnt, wie schon beim Giftwrapping Algorithmus, der Graham Scan Solver. "Punkte auf Kreis"

In Tabelle 5.12 untersuchen wir wieder den Einfluss der Anzahl verwendeter Threads auf Beschleunigung und Laufzeit. Wir ziehen hierzu nur die Anordnung "Punkte im Quadrat" heran.

Es zeigt sich, dass für größere Eingaben ($4*10^6$,$8*10^6$) die Beschleunigung leicht steigt und die Laufzeit leicht sinkt. Einfluss einer höheren Anzahl Threads ist wiederum gering.

In Abbildung 5.8 vergleichen wir nochmals die Beschleunigungsfaktoren des Quickhull Algorithmus mit den Solvern "randomisiert inkrementell" und "Grahma Scan". Auf der linken Seite sehen wir den Vergleich der Solver bei verschiedenen Eingabeanord-

Eingabe	Punkte im Quadrat								
Threads	4			8			16		
Implementierung	Quick-hull	Graham Scan	random-isiert Inkre-mentell	Quick-hull	Graham Scan	random-isiert Inkre-mentell	Quick-hull	Graham Scan	random-isiert Inkre-mentell
$1*10^6$	2.32	2.33	2.34	2.26	2.28	2.28	2.23	2.22	2.23
$2*10^6$	2.22	2.22	2.22	2.31	2.29	2.29	2.17	2.25	2.22
$4*10^6$	2.16	2.17	2.17	2.33	2.36	2.35	2.28	2.30	2.28
$8*10^6$	2.09	2.09	2.09	2.28	2.29	2.26	2.25	2.24	2.25

Eingabe	Punkte im Quadrat								
Threads	4			8			16		
Implementierung	Quick-hull	Graham Scan	random-isiert Inkre-mentell	Quick-hull	Graham Scan	random-isiert Inkre-mentell	Quick-hull	Graham Scan	random-isiert Inkre-mentell
$1*10^6$	0.25	0.24	0.24	0.25	0.25	0.25	0.26	0.26	0.26
$2*10^6$	0.49	0.49	0.48	0.47	0.47	0.47	0.50	0.48	0.48
$4*10^6$	1.01	1.01	1.01	0.94	0.93	0.93	0.96	0.95	0.96
$8*10^6$	2.04	2.04	2.04	1.88	1.87	1.89	1.90	1.91	1.90

Tabelle 5.12.: Vergleich Quickhull Lösungen: Die Quickhull Jobs werden von verschiedenen sequentiell Solvern gelöst. Diese sind in der Reihe "Implementierung" aufgeführt; Die Threadanzahl variiert; Die Punkte sind im Quadrat gleichverteilt; Oben: Beschleunigungsfaktor nach Implementierungen und Eingabegrößen; Unten: Laufzeit nach Implementierungen und Eingabegrößen;

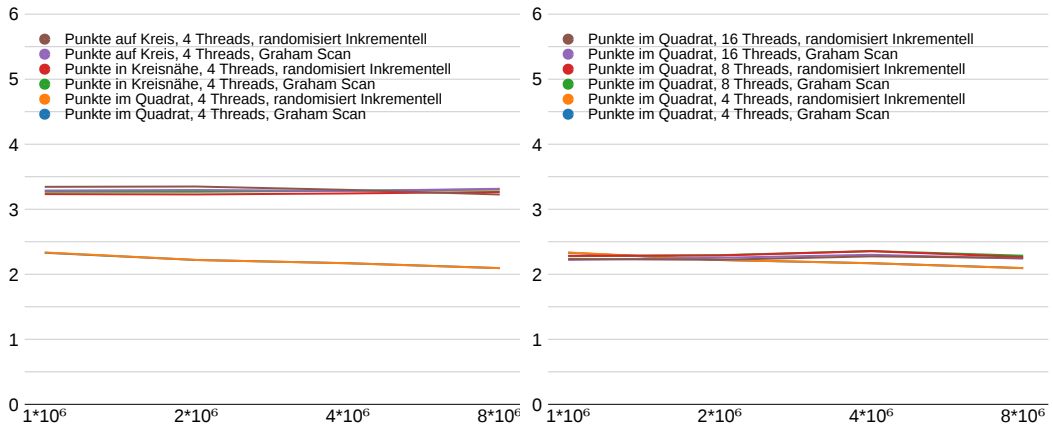


Abbildung 5.8.: Analyse Quickhull Solver: X-Achse: Eingabegröße; Y-Achse: Beschleunigungsfaktor; Links: Auswertung des Quickhull Algorithmus mit den Solvern Graham Scan und randomisiert Inkrementell für 4 Threads und verschiedenen Eingaben; Rechts: Auswertung des Quickhull Algorithmus mit den Solvern Graham Scan und randomisiert Inkrementell für "Punkte im Quadrat" und 4,8,16 Threads;

nungen. Beide Implementierungen haben sehr ähnliche Beschleunigungsfaktoren für die

unterschiedliche Eingabeanordnungen. Für die "Punkte im Quadrat" ist die Beschleunigung deutlich schlechter. Auf der rechten Seite sehen wir, dass ein wachsenden Anzahl von Thread die Beschleunigung des Quickhull Algorithmus praktisch nicht messbar beeinflusst. Wie wir oben schon festgestellt haben dominiert das Teilen der Jobs für die Anordnung "Punkte im Quadrat" die Laufzeit. Mehr Threads können die Teilung der Jobs offenbar nicht beschleunigung.

5.4.3.3. Quickhull - weitere Lösungen

Zum Quickhull Job mit seinen verschiedenen sequentiellen Solvern aus dem letzten Teilabschnitt stehen uns weitere Alternativen zur Verfügung. Zum einen haben wir die Option nur die obere bzw. untere Hülle zu berechnen, zum anderen Verfügen wir über eine Implementierung, die auf Listen basiert, das heißt die Punktmengen werden in Listen zwischengespeichert und nicht in Feldern, was technische Vorteile bei der Partitionierung in 3 Teilmengen bringt.

Eingabe	Punkte im Quadrat			Punkte in Kreisnähe			Punkte auf Kreis		
Threads	4								
Implementierung	nur obere Hülle	Quick-hull	listen-basiert	nur obere Hülle	Quick-hull	listen-basiert	nur obere Hülle	Quick-hull	listen-basiert
$1*10^6$	1.76	2.32	1.31	2.99	3.24	2.66	3.19	3.30	3.01
$2*10^6$	1.76	2.22	1.33	2.99	3.26	2.79	3.27	3.36	3.20
$4*10^6$	1.79	2.16	1.35	2.98	3.29	2.85	3.33	3.40	3.29
$8*10^6$	1.71	2.09	1.34	2.98	3.28	2.87	3.33	3.41	3.35

Eingabe	Punkte im Quadrat			Punkte in Kreisnähe			Punkte auf Kreis		
Threads	4								
Implementierung	nur obere Hülle	Quick-hull	listen-basiert	nur obere Hülle	Quick-hull	listen-basiert	nur obere Hülle	Quick-hull	listen-basiert
$1*10^6$	0.18	0.25	0.21	0.95	1.75	1.30	2.16	4.15	3.60
$2*10^6$	0.35	0.49	0.41	1.75	3.19	2.33	5.10	9.89	8.72
$4*10^6$	0.70	1.01	0.81	3.37	6.07	4.54	11.95	23.32	21.02
$8*10^6$	1.42	2.04	1.62	6.69	12.08	9.40	27.87	54.33	49.78

Tabelle 5.13.: Vergleich alternativer Quickhull Lösungen: Die Quickhull Jobs werden von verschiedenen sequentiell Solvern gelöst. Diese sind in der Reihe "Implementierung" aufgeführt; Die Threadanzahl ist 4; Die Eingabeanordnung variiert; Oben: Beschleunigungsfaktor nach Implementierungen und Eingabegrößen; Unten: Laufzeit nach Implementierungen und Eingabegrößen;

In Tabelle 5.13 vergleichen wir den listenbasierten Ansatz mit der Implementierung aus Teilabschnitt 5.4.3.2 und betrachten das Verhalten bei der Berechnung der oberen Hülle.

Zunächst vergleichen wir die "obere Hülle" mit der vollständigen "Quickhull" Berechnung. Für die Anordnung "Punkte im Quadrat" messen wir für die "obere Hülle" eine noch schlechter Beschleunigung als "Quickhull". Quickhull hat einen Beschleunigungsvorteil, da die obere und untere Hülle parallel gelöst wird und sich so die degenerierten Rekursionsbäumen geringer auswirken. Entsprechend sind die Laufzeiten der "oberen Hülle" nicht

wie eventuell erwartet um den Faktor 2 schneller. Für die Anordnungen "Punkte in Kreisnähe" und "Punkte auf Kreis" ist die Beschleunigung gut und die Laufzeit zu "Quickhull" halbiert.

Der listenbasierte Ansatz liefert durchweg bessere Laufzeiten ab. Gleichzeitig sind die Beschleunigungsfaktoren nur für die Anordnung "Punkt auf Kreis" gut, für die Anordnung "Punkte im Quadrat" dagegen schlecht. Die listenbasierte Implementierung kann weiter analysiert und optimiert werden.

5.4.3.4. Standardsolver und paralleles Partitionieren

Wir untersuchen nun für die Giftwrapping und Quickhull Jobs den Einfluss der parallelen Partitionierung und des Standardsolvers im Vergleich zum speziellen Solver.

Eingabe	Punkte im Quadrat		Punkte in Kreisnähe		Punkte auf Kreis	
Threads	4					
Implementierung	Gift Wrap-ping	Standard-solver	Gift Wrap-ping	Standard-solver	Gift Wrap-ping	Standard-solver
$1*10^6$	3.26	3.28	3.38	3.39	3.34	3.36
$2*10^6$	3.26	3.26	3.41	3.43	3.38	3.42
$4*10^6$	3.24	3.26	3.42	3.43	3.34	3.40
$8*10^6$	3.23	3.24	3.44	3.43	3.29	3.39

Eingabe	Punkte im Quadrat		Punkte in Kreisnähe		Punkte auf Kreis	
Threads	4					
Implementierung	Gift Wrap-ping	Standard-solver	Gift Wrap-ping	Standard-solver	Gift Wrap-ping	Standard-solver
$1*10^6$	0.57	0.59	0.91	0.94	1.05	1.09
$2*10^6$	1.21	1.24	2.25	2.31	2.22	2.29
$4*10^6$	2.60	2.65	5.43	5.58	4.71	4.78
$8*10^6$	5.55	5.67	12.44	12.83	9.98	9.94

Tabelle 5.14.: Vergleich Giftwrapping spezieller Solver/Standardsolver: Die Threadanzahl ist 4; Die Eingabeanordnung variiert; Oben: Beschleunigungsfaktor nach Implementierungen und Eingabegrößen; Unten: Laufzeit nach Implementierungen und Eingabegrößen;

In Tabelle 5.14 vergleichen wir den sequentiellen Giftwrapping Solver, den wir auch in den anderen Experimenten verwenden, mit der Leistung des Standardsolvers. Die Beschleunigungsfaktoren sind durchweg gut, die Werte sind beinahe gleich mit einem sehr leichten Vorteil für den speziellen Solver. Die Laufzeittabelle zeigt das gleiche Bild.

In Tabelle 5.15 vergleichen wir den speziellen Solver des Quickhull Algorithmus mit dem Standardsolver und untersuchen die Auswirkungen des parallelen Partitionierens.

Die Beschleunigungsfaktoren des speziellen und das Standardsolver sind sich sehr ähnlich. Bei den Laufzeiten lässt sich ein Vorteil für den speziellen Solver ausmachen, insbesondere für die Anordnung "Punkte auf Kreis".

Eingabe	Punkte im Quadrat			Punkte in Kreisnähe			Punkte auf Kreis		
Threads	4								
Implementierung	Quick-hull	Standard-solver	seq. Partiti-on	Quick-hull	Standard-solver	seq. Partiti-on	Quick-hull	Standard-solver	seq. Partiti-on
$1*10^6$	2.32	2.31	1.72	3.24	3.30	3.18	3.30	3.34	3.28
$2*10^6$	2.22	2.22	1.79	3.26	3.31	3.22	3.36	3.38	3.34
$4*10^6$	2.16	2.16	1.71	3.29	3.33	3.21	3.40	3.39	3.36
$8*10^6$	2.09	2.09	1.64	3.28	3.31	3.22	3.41	3.42	3.40

Eingabe	Punkte im Quadrat			Punkte in Kreisnähe			Punkte auf Kreis		
Threads	4								
Implementierung	Quick-hull	Standard-solver	seq. Partiti-on	Quick-hull	Standard-solver	seq. Partiti-on	Quick-hull	Standard-solver	seq. Partiti-on
$1*10^6$	0.25	0.25	0.33	1.75	1.91	1.78	4.15	4.46	4.19
$2*10^6$	0.49	0.48	0.60	3.19	3.51	3.23	9.89	10.58	9.94
$4*10^6$	1.01	1.01	1.28	6.07	6.71	6.20	23.32	25.04	23.62
$8*10^6$	2.04	2.04	2.61	12.08	13.38	12.28	54.33	57.68	54.45

Tabelle 5.15.: Vergleich Quickhull spezieller Solver/Standardsolver und sequentielles Partitionieren; Die Threadanzahl ist 4; Die Eingabeanordnung variiert; Oben: Beschleunigungsfaktor nach Implementierungen und Eingabegrößen; Unten: Laufzeit nach Punkteanordnung und Eingabegrößen;

Das sequentielle Partitionieren ist wie erwartet langsamer als die parallele Variante. Besondere Vorteile bringt das parallele Partitionieren bei der Anordnung "Punkte im Quadrat", sowohl bezüglich der Beschleunigungsfaktoren, als auch der Laufzeiten. Bei der Anordnung "Punkt auf Kreis" ist der Gewinn durch paralleles Partitionieren minimal.

5.4.3.5. Giftwrapping 3D

Der Giftwrapping Algorithmus aus Abschnitt 3.5.1 liegt uns auch in einer Implementierung zur Berechnung in 3D vor.

In Tabelle 5.16 sehen wir die Ergebnisse der Giftwrapping Experimente in 3D. Es ist zu sehen, dass die Beschleunigungsfaktoren nie befriedigende Werte erreichen. Die Ursachen sind weiter zu analysieren.

5.4.3.6. Zusammenfassung

Wir haben in diesem Abschnitt die Leistung unsere Framewoks an Hand des Quickhull und des Giftwrapping Algorithmus untersucht. Wir haben mit beiden Implementierungen überwiegend gute bis sehr gute Beschleunigungsfaktoren gemessen.

Das Framework liefert, wie zu erwarten, in den Fällen eine durchschnittliche Leistung ab, in denen die Rekursionsbäume des parallelisierten Algorithmus degenerieren. Unser Ansatz auch einzelne Funktionen zu parallelisieren (bspw. parallele Partitionierung, siehe Abschnitt 2.6) bringt klare Verbesserungen, kann das Problem aber nicht vollständig beheben.

Eingabe	Punkte im Würfel			Punkte auf Paraboloid			Punkte auf Kugel		
Threads	4	8	16	4	8	16	4	8	16
Implementierung	Gift Wrapping in 3D								
$1*10^5$	1.76	1.05	0.61	1.52	1.09	0.98	1.81	1.08	0.61
$2*10^5$	1.78	1.06	0.60	1.54	1.09	1.00	1.82	1.08	0.62
$4*10^5$	1.78	1.06	0.60	1.54	1.09	1.03	1.82	1.08	0.62
$8*10^5$	1.79	1.06	0.60	1.50	1.08	1.01	1.83	1.09	0.62

Eingabe	Punkte im Würfel			Punkte auf Paraboloid			Punkte auf Kugel		
Threads	4	8	16	4	8	16	4	8	16
Implementierung	Gift Wrapping in 3D								
$1*10^5$	1.43	2.40	4.16	2.62	3.67	4.05	1.37	2.30	4.03
$2*10^5$	2.85	4.80	8.51	5.42	7.66	8.32	2.73	4.60	8.01
$4*10^5$	5.72	9.63	16.93	11.60	16.30	17.36	5.47	9.21	16.16
$8*10^5$	11.41	19.28	34.13	26.92	37.33	39.87	10.95	18.37	32.22

Tabelle 5.16.: Vergleich Giftwrapping spezieller Solver/Standardsolver: Oben: Beschleunigungsfaktor nach Implementierungen und Eingabegrößen; Unten: Laufzeit nach Implementierungen und Eingabegrößen;

Weiter haben wir unsere parallen Implementierungen mit verschiedenen sequentiellen Solvern kombiniert. Wir haben gezeigt, dass sich mit den speziellen sequentiellen Solvern die tatsächliche Laufzeit nochmals reduzieren lässt.

Zudem haben die Experimente gezeigt, dass es im Allgemeinen kontraproduktiv ist, die Anzahl der verwendeten Threads deutlich über die verfügbaren Rechenkerne zu steigern.

Die Giftwrapping Implementierung in 3D hat enttäuscht. Weitere Analysen sind notwendig. Ein Ansatz ist hier die Untersuchung der verwendeten geometrischen Objekte.

5.4.4. Triangulierungsalgorithmen

Wir haben in Abschnitt 3.6 verschiedene Traingulierungsalgorithmen vorgestellt. Wir stellen hier die experimentellen Ergebnisse vor. Wir beginnen mit der Unterteilung eines Dreiecks 5.4.4.1, fahren vor mit der Triangulierung durch den Giftwrapping Algorithmus 5.4.4.2 und schließen die Testreihe mit der Delaunay Triangulierung ab 5.4.4.3.

Für die Algorithmen liegen verschiedenen Implementierungen vor. Diese unterscheiden sich in der verwendeten Datenstruktur. Wir verwenden den threadsichern Leda Graphen aus Abschnitt 3.6.1.1 ("Leda Graph"), eine kantenbasierte Datenstruktur aus Abschnitt 3.6.1.2 ("Edge Struktur") und eine dreiecksbasierte Struktur 3.6.1.3 ("Dualer Graph").

Die Eingabepunkte sind in den folgenden Experimenten immer zufallsverteilt in einem Quadrat. Weiter variieren wir die Anzahl der Threads und die Eingabegrößen. Wir verwenden Punktmengen der Mächtigkeit $1 * 10^6$, $2 * 10^6$, $4 * 10^6$ und $8 * 10^6$.

5.4.4.1. Unterteilung eines Dreiecks

Die Triangulierung durch rekursive Unterteilung eines Dreiecks haben wir in Abschnitt 3.6.1 diskutiert.

Eingabe	Punkte im Quadrat								
Threads	4			8			16		
Implementierung	Leda Graph	Edge Struktur	Dualer Graph	Leda Graph	Edge Struktur	Dualer Graph	Leda Graph	Edge Struktur	Dualer Graph
$1*10^6$	3.13	3.14	3.20	1.10	1.27	1.64	0.83	0.95	1.50
$2*10^6$	3.09	3.16	3.21	1.17	1.35	1.71	0.88	1.01	1.55
$4*10^6$	3.09	3.19	3.21	1.27	1.43	1.78	0.95	1.06	1.60
$8*10^6$	3.21	3.19	3.20	1.36	1.53	1.90	1.02	1.13	1.66

Eingabe	Punkte im Quadrat								
Threads	4			8			16		
Implementierung	Leda Graph	Edge Struktur	Dualer Graph	Leda Graph	Edge Struktur	Dualer Graph	Leda Graph	Edge Struktur	Dualer Graph
$1*10^6$	1.87	1.62	1.56	5.32	3.99	3.05	7.08	5.34	3.34
$2*10^6$	4.01	3.42	3.35	10.66	8.02	6.28	14.11	10.69	6.92
$4*10^6$	8.70	7.18	7.04	21.17	16.03	12.67	28.28	21.60	14.13
$8*10^6$	17.77	15.31	15.05	41.99	31.80	25.38	56.06	43.10	28.98

Tabelle 5.17.: Vergleiche Lösungen einfache Triangulierung: Oben: Beschleunigungsfaktor nach Implementierungen und Eingabegrößen; Unten: Laufzeit nach Implementierungen und Eingabegrößen;

In Tabelle 5.17 präsentieren wir die Testergebnisse der verschiedenen Implementierungen. Die Beschleunigungsfaktoren erreichen mit 4 Threads gute Werte. Die besten Werte erzielt dabei die Implementierung mit dem dualen Graph. Mit mehr als 4 Threads verringert sich der Beschleunigungsfaktor für alle Implementierungen drastisch.

Die Laufzeiten zeigen das gleiche Bild: Die Implementierung mit dem dualen Graphen ist am schnellsten. Die weitere Erhöhung der Threadanzahl ist kontraproduktiv.

5.4.4.2. Giftwrapping

Die Weiterentwicklung des nun bekannten Giftwrapping Algorithmus zu einem Triangulierungsalgorithmus haben wir in Abschnitt 3.6.2 beschrieben. Uns liegen Implementierungen mit dem Leda Graphen und der kantenbasierten Datenstruktur vor.

In Tabelle 5.18 sehen wir die Ergebnisse der Laufzeitexperimente. Mit 4 Threads liefert der Leda Graph befriedigende und die kantenbasierte Struktur gute Ergebnisse. Wieder reduzieren sich die Beschleunigungsfaktoren dramatisch bei Verwendung zusätzlicher Threads. Die Laufzeiten verhalten sich analog.

5.4.4.3. Delaunay Triangulierung

Den Divide und Conquer Algorithmus zur Berechnung der Delaunay Triangulierung haben wir in Abschnitt 3.6.3 eingeführt. Wir implementieren ihn mit dem Leda Graphen und der kantenbasierten Datenstruktur.

In Tabelle 5.19 werden die Testergebnisse der Delaunay Triangulierung dargestellt. Die Beschleunigungsfaktoren mit 4 Threads sind mit der kantenbasierten Struktur gut,

Eingabe	Punkte im Quadrat					
Threads	4		8		16	
Implementierung	Leda Graph	Edge Struktur	Leda Graph	Edge Struktur	Leda Graph	Edge Struktur
$1*10^6$	2.83	3.19	0.71	0.86	0.53	0.67
$2*10^6$	2.84	3.19	0.73	0.89	0.55	0.66
$4*10^6$	2.86	3.20	0.76	0.93	0.57	0.68
$8*10^6$	2.88	3.19	0.80	0.98	0.59	0.70

Eingabe	Punkte im Quadrat					
Threads	4		8		16	
Implementierung	Leda Graph	Edge Struktur	Leda Graph	Edge Struktur	Leda Graph	Edge Struktur
$1*10^6$	1.00	0.85	3.97	3.14	5.31	4.04
$2*10^6$	2.07	1.78	8.04	6.40	10.76	8.60
$4*10^6$	4.30	3.74	16.11	12.81	21.69	17.64
$8*10^6$	8.93	7.91	31.97	25.74	43.53	36.04

Tabelle 5.18.: Vergleiche Lösungen Giftwrapping Triangulierung: Oben: Beschleunigungsfaktor nach Implementierungen und Eingabegrößen; Unten: Laufzeit nach Implementierungen und Eingabegrößen;

Eingabe	Punkte im Quadrat					
Threads	4		8		16	
Implementierung	Leda Graph	Edge Struktur	Leda Graph	Edge Struktur	Leda Graph	Edge Struktur
$1*10^6$	2.75	3.33	0.85	0.98	0.67	0.84
$2*10^6$	2.77	3.27	1.01	1.16	0.75	0.94
$4*10^6$	2.79	3.15	1.28	1.45	0.95	1.15
$8*10^6$	2.62	2.81	1.60	1.75	1.26	1.41

Eingabe	Punkte im Quadrat					
Threads	4		8		16	
Implementierung	Leda Graph	Edge Struktur	Leda Graph	Edge Struktur	Leda Graph	Edge Struktur
$1*10^6$	2.96	2.49	9.56	8.47	12.18	9.87
$2*10^6$	6.62	5.70	18.08	16.07	24.36	19.81
$4*10^6$	16.11	14.47	35.07	31.41	47.09	39.76
$8*10^6$	46.64	43.85	76.50	70.66	97.23	87.36

Tabelle 5.19.: Vergleiche Lösungen Delaunay Triangulierung: Oben: Beschleunigungsfaktor nach Implementierungen und Eingabegrößen; Unten: Laufzeit nach Implementierungen und Eingabegrößen;

mit dem Leda Graphen nur befriedigend. Die Benutzung weiterer Threads ist für die Beschleunigung sehr schädlich. Die Laufzeitmessungen geben das gleiche Bild ab.

5.4.4.4. Zusammenfassung

Wir haben in diesem Abschnitt die Leistung unsere Framewoks mit verscheidenen Traingulierungsalgortihmen untersucht. Dabei haben wir überwiegend gute Beschleunigungsfaktoren gemessen.

Mit den beiden leichteren, speziellen Datenstrukturen, der kanten- und der dreiecksbasierten Datenstruktur, erzielen wir fast ausschließlich gute Beschleunigungsfaktoren. Der deutlich komplexere und flexible Leda Graph schneidet etwas schlechter ab, zeigt aber ein berfriedigendes Ergebnis.

Zudem haben die Experimente gezeigt, dass es im Allgemeinen kontraproduktiv ist, die Anzahl der verwendeten Threads deutlich über die verfügbaren Rechenkerne zu steigern.

5.4.5. Zusammenfassung

Wir haben mit dem Divide und Conquer Framework eine breite Palette von Experimenten durchgeführt.

Zuerst haben wir den optimalen Wert für den Limit Parameter experimentell bestimmt, siehe 5.4.1. Der Limit Parameter definiert die Problemgröße unterhalb derer Teilprobleme sequentiell gelöst werden. Damit ist er für alle weiteren Divide und Conquer Experimente entscheident. Es hat aich gezeigt, dass der Faktor 1/1000 in Relation zur Eingabegröße in den meisten Fällen das beste Ergebnis erzielt. Experimentiert haben wir hierzu mit den Algorithmen Quicksort, Mergesort, Giftwrapping und Quickhull.

Wir haben jeweils mehrere Algorithmen aus den Bereichen Sortieren, konvexe Hülle und Triangulierung implementiert. Mit diesen Implementierungen haben wir gezeigt, dass es möglich ist mit unserem Framework gute bis sehr gute Beschleunigungsfaktoren zu erzielen. Weiter zeigt der Vergleich mit der MCSTL Implementierung verschiedener Sortieralgorithmen, dass effizient implementierte Jobs auch kompetitive Laufzeiten erzielen.

Unser Konzept für die Lösung kleiner Problemgrößen andere Algorithmen heranzuziehen hat sich bewährt. Mit der Benutzung spezieller sequentieller Solver lässt sich die Laufzeit weiter reduzieren. Die tatsächliche gewonnene Leistung ist dabei vom sequentiellen Algorithmus abhängig.

Auch die Parallelisierung der Divide Funktion unter Zuhilfenahme zum Beispiel der parallelen Partitionierung hat sich als Leistungssteigerung erwiesen. Leider ist die Leistungssteigerung nicht hoch genug, um auch die Fälle vollständig abzufedern bei denen der Rekursionsbaum degeneriert, siehe Quickhull 5.4.3.4. Hier liegt weiteres Verbesserungspotential vor.

Die threadsichere Aufbereitung des Leda Graphen hat sich in den Experimenten zur Triangulierung als leistungsfähig erwiesen. Die Implementierung ist dabei zwar bei weitem nicht so effizient wie spezialisierte Datenstrukturen, aber als universale und flexible Struktur hat der Leda Graph primär andere Vorzüge.

Die Experimente haben gezeigt, dass es im Allgemeinen kontraproduktiv für unser Framework ist, die Anzahl der verwendeten Threads deutlich über die verfügbaren Rechenkerne zu steigern.

5.5. Randomisiert Inkrementelle Experimente

Wir untersuchen hier die Effizienz des Frameworks für randomisiert inkrementelle Algorithmen. Insbesondere interessieren wir uns für die Beschleunigungsfaktoren der getesteten Algorithmen.

Wir beginnen mit dem randomisiert inkrementellen Algorithmus zur Berechnung der konvexen Hülle in Teilabschnitt 5.5.1. Zu diesem Algorithmus liegen uns zwei Implementierungen vor, die sich in der Deadlockvermeidungsstrategie unterscheiden, siehe Abschnitt 4.4.2. Wir stellen die Threadhierarchie und die Mutexhierarchie gegenüber.

In Teilabschnitt 5.5.2 diskutieren wir die inkrementelle Delaunay Triangulierung und schließen mit einer Zusammenfassung 5.5.3.

5.5.1. Konvexe Hülle Algorithmen

Den konvexe Hülle Algorithmus haben wir in Abschnitt 4.4 eingeführt. Wir verfügen über 2 Implementierungen, die sich in der Deadlockvermeidungsstrategie unterscheiden. Die Strategien heißen Threadhierarchie und Mutexhierarchie, entsprechend benennen wir die Implementierungen. Für die folgenden Experimente arbeiten wir mit verschiedenen Anordnungen der Eingabepunkte: gleichverteilte Punkte im Quadrat ("Punkte im Quadrat"), Punkte in relativer Nähe zu einem Kreis ("Punkte in Kreisnähe") und Punkte mit gegebener Genauigkeit auf einem Kreis ("Punkte auf Kreis"). Als Eingabe verwenden wir Punktmengen der Mächtigkeit $1 * 10^6$, $2 * 10^6$, $4 * 10^6$ und $8 * 10^6$.

Eingabe	Punkte im Quadrat		Punkte in Kreisnähe		Punkte auf Kreis	
Threads	4					
Implementierung	Thread-hierarchie	Mutex-hierarchie	Thread-hierarchie	Mutex-hierarchie	Thread-hierarchie	Mutex-hierarchie
$1*10^6$	*3.87*	3.70	*2.77*	2.71	2.69	2.66
$2*10^6$	*3.69*	3.40	*3.04*	2.99	2.85	2.84
$4*10^6$	*3.89*	3.51	*3.26*	3.20	2.98	2.96
$8*10^6$	*3.67*	3.64	3.34	3.35	2.99	3.03

Eingabe	Punkte im Quadrat		Punkte in Kreisnähe		Punkte auf Kreis	
Threads	4					
Implementierung	Thread-hierarchie	Mutex-hierarchie	Thread-hierarchie	Mutex-hierarchie	Thread-hierarchie	Mutex-hierarchie
$1*10^6$	0.03	0.03	1.13	1.07	3.07	2.95
$2*10^6$	0.06	0.07	2.24	*2.14*	7.90	*7.62*
$4*10^6$	0.12	0.13	4.56	*4.36*	19.65	*19.10*
$8*10^6$	0.23	0.25	9.32	*8.83*	47.77	*46.06*

Tabelle 5.20.: Vergleiche Konvexe Hülle Thread-/Mutexhierarchie: Oben: Beschleunigungsfaktor nach Implementierungen und Eingabeanordnung; Unten: Laufzeit nach Implementierungen und Eingabeanordnung;

In Tabelle 5.20 vergleichen wir die Implementierung der Mutexhierarchie mit der Threadhierarchie. Die besten Werte pro Eingabegröße und -anordnung sind kursiv dargestellt.

Die Beschleunigungsfaktoren der Threadhierarchie sind für die Anordnungen “Punkte im Quadrat” und “Punkte in Kreisnähe” gut bis sehr gut, die Mutexhierarchie liefert überwiegend gute Resultate. Bei der Anordnung “Punkte auf Kreis” fallen beide Implementierungen ab und zeigen nur befriedigende Werte. Die Beschleunigungsfaktoren der Threadhierarchie übertreffen in den meisten Fällen die der Mutexhierarchie.

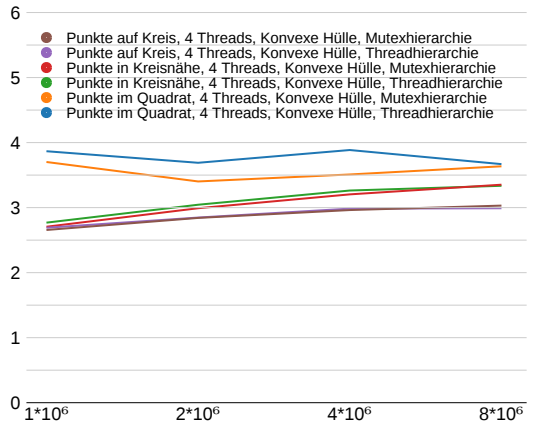


Abbildung 5.9.: Analyse Quickhull Solver: X-Achse: Eingabegröße; Y-Achse: Beschleunigungsfaktor; Links: Auswertung des Quickhull Algorithmus mit den Solvern Graham Scan und randomisiert Inkrementell für 4 Threads und verschiedenen Eingaben;

In Abbildung 5.9 stellen wir nochmal die Laufzeiten der beiden Hierarchien bei verschiedenen Eingabeanordnungen gegenüber. Wir können für die Threadhierarchie ein leichten Vorteil bei den Anordnungen “Punkte im Quadrat” und “Punkte in Kreisnähe” ablesen. Im Falle der “Punkte auf Kreis” liegen beide ungefähr gleichauf.

Die absoluten Laufzeiten sind bei der Anordnung “Punkte im Quadrat” sehr nahe beieinander. Bei den anderen beiden Anordnungen verbucht die Mutexhierarchie einen Vorteil, wenn auch die Laufzeiten für “Punkte auf Kreis” bei beiden Implementierungen schon sehr hoch sind. Insgesamt ist die Bandbreite der Laufzeiten der verschiedenen Anordnungen hoch. Der Algorithmus ist sehr schnell für die Anordnung “Punkte im Quadrat” bis hin zu langsam für “Punkte auf Kreis”.

In Tabelle 5.21 sehen wir die Entwicklung der Beschleunigungsfaktoren und Laufzeiten der Threadhierarchie für höhere Anzahlen von Threads. Die Leistung sinkt in allen Fällen unter das Niveau von 4 verwendeten Threads ab.

5.5.2. Delaunay Triangulierung

Wir präsentieren hier die Testergebnisse des randomisiert inkrementellen Algorithmus zur Delaunay Triangulierung, den wir in Abschnitt 4.5 diskutiert haben. Wir testen den Algorithmus mit verschiedenen Anordnungen und den Eingabegrößen $1 * 10^5$, $2 * 10^5$, $4 * 10^5$ und $8 * 10^5$.

Eingabe	Punkte im Quadrat			Punkte in Kreisnähe			Punkte auf Kreis		
Threads	4	8	16	4	8	16	4	8	16
Implementierung	Konvexe Hülle, Threadhierarchie								
$1*10^6$	3.87	2.77	2.39	2.77	1.28	1.13	2.69	1.42	1.35
$2*10^6$	3.69	2.85	2.54	3.04	1.59	1.53	2.85	1.65	1.65
$4*10^6$	3.89	3.29	3.23	3.26	1.86	1.97	2.98	1.81	1.86
$8*10^6$	3.67	3.30	3.39	3.34	2.19	2.32	2.99	1.87	1.97

Eingabe	Punkte im Quadrat			Punkte in Kreisnähe			Punkte auf Kreis		
Threads	4	8	16	4	8	16	4	8	16
Implementierung	Konvexe Hülle, Threadhierarchie								
$1*10^6$	0.03	0.04	0.05	1.13	2.44	2.77	3.07	5.82	6.10
$2*10^6$	0.06	0.08	0.09	2.24	4.30	4.47	7.90	13.63	13.66
$4*10^6$	0.12	0.14	0.14	4.56	7.99	7.55	19.65	32.41	31.46
$8*10^6$	0.23	0.26	0.25	9.32	14.18	13.38	47.77	76.50	72.44

Tabelle 5.21.: Konvexe Hülle, Threadhierarchie: Oben: Beschleunigungsfaktor nach Threadanzahl und Eingabeanordnung; Unten: Laufzeit nach Threadanzahl und Eingabeanordnung;

Eingabe	Punkte im Quadrat			Punkte in Kreisnähe			Punkte auf Kreis		
Threads	4	8	16	4	8	16	4	8	16
Implementierung	Delaunay Triangulation, rand. inkr.								
$1*10^5$	2.47	1.66	1.23	3.19	2.28	1.91	3.32	2.42	2.14
$2*10^5$	2.89	2.05	1.56	3.39	2.57	2.38	3.44	2.73	2.67
$4*10^5$	3.19	2.42	1.93	3.46	2.79	2.74	3.48	3.01	3.13
$8*10^5$	3.35	2.77	2.33	3.49	2.95	3.04	3.49	3.25	3.48

Eingabe	Punkte im Quadrat			Punkte in Kreisnähe			Punkte auf Kreis		
Threads	4	8	16	4	8	16	4	8	16
Implementierung	Delaunay Triangulation, rand. inkr.								
$1*10^5$	1.81	2.70	3.63	7.44	10.41	12.45	9.26	12.69	14.34
$2*10^5$	4.88	6.89	9.04	25.02	33.06	35.68	41.64	52.49	53.66
$4*10^5$	13.52	17.81	22.32	66.71	82.57	84.06	176.35	203.79	195.87
$8*10^5$	38.20	46.12	54.85	142.54	168.35	163.48	730.33	783.87	733.43

Tabelle 5.22.: Inkrementelle Delaunay Triangulierung:
Oben: Beschleunigungsfaktor nach Threadanzahl und Eingabeanordnung; Unten: Laufzeit nach Threadanzahl und Eingabeanordnung;

In Tabelle 5.22 sehen wir die Testergebnisse der Delaunay Triangulierung. Die Beschleunigungsfaktoren mit 4 Threads sind für alle Punkteanordnungen überweigend gut. Mit 8 und 16 Threads ist die Beschleunigung wiederum geringer. Bei den Laufzeiten zeigt sich die quadratische asymptotische Laufzeit des Algorithmus; die Implementierung ist für größere Eingabemengen langsam.

5.5.3. Zusammenfassung

Wir haben gezeigt, dass sich mit unserem randomisiert inkrementellen Framework mit verschiedenen Implementierungen gute Beschleunigungswerte erzielen lassen.

Weiter haben wir im Kontext der konevexe Hülle Experimente gezeigt, dass beide

Deadlockvermeidungsstrategien (Thread/-Mutexhierarchie), die wir in Abschnitt 4.4.2 vorgestellt haben, gute Ergenisse liefern können.

Die gezeigte Implementierungen zur Delaunay Triangulierung hat eine ungenügende Laufzeit. Der Algorithmus kann durch Suchstrukturen erweitert werden, wie bereits in Abschnitt 4.6 vorgeschlagen.

5.6. Zusammenfassung

Im diesem Kapitel haben wir zunächst die Leistungsfähigkeit verschiedener threadsicherer Deque Implementierungen getestet, siehe Abschnitt 5.3. Wir haben gezeigt, dass die Implementierung der threadbasierten Deque aus Abschnitt 2.5.2.2 am effizientesten arbeitet. Wir setzen diese Implementierung zur Speicherverwaltung im Leda Graphen ein.

Im Anschluss haben wir in Abschnitt 5.4 zahlreiche Experimente mit unserem Divide und Conquer Framework durchgeführt.

Wir haben zuerst den optimalsten Wert für den Limit Parameter aus Abschnitt 5.4.1 auf Basis der Algorithmen Quicksort, Mergesort, Giftwrapping und Quickhull experimentell ermittelt. Der optimalste Wert des Parameters ist 1/1000 in Relation zur Eingabegröße.

Mit Implementierungen verschiedener Algorithmen aus den Bereichen Sortieren, konvexe Hülle und Triangulierung haben wir gezeigt, dass unser Framework gute bis sehr gute Beschleunigungswerte erzielen kann. Dabei braucht unser Framework auch den Vergleich mit anderen Ansätzen der Parallelisierung nicht zu scheuen. In Experimenten mit Sortieralgorithmen aus der MCSTL Bibliothek zeigt unser Framework bessere Laufzeiten und Beschleunigungswerte.

Unser Konzept für die Lösung kleiner Problemgrößen andere Algorithmen heranzuziehen hat sich bewährt. Mit der Benutzung spezieller sequentieller Solver lässt sich die Laufzeit weiter reduzieren. Die tatsächliche gewonnene Leistung ist dabei vom sequentiellen Algorithmus abhängig.

Auch die Parallelisierung der Divide Funktion unter Zuhilfenahme zum Beispiel der parallelen Partitionierung hat sich als Leistungssteigerung erwiesen. Leider ist die Leistungssteigerung nicht hoch genug, um auch die Fälle vollständig abzufedern bei denen der Rekursionsbaum degeneriert, siehe Quickhull 5.4.3.4. Hier liegt weiteres Verbesserungspotential vor.

Die threadsichere Aufbereitung des Leda Graphen hat sich in den Experimenten zur Triangulierung als leistungsfähig erwiesen. Die Implementierung ist dabei zwar bei weitem nicht so effizient wie spezialisierte Datenstrukturen, aber als universale und flexible Struktur hat der Leda Graph primär andere Vorzüge.

Zuletzt haben wir in Abschnitt 5.5 die randomisiert inkrementellen Algorithmen untersucht. Auch hier haben wir gezeigt, dass unser Framework in der Lage ist mit verschiedenen Implementierungen gute Beschleunigungswerte zu erzielen. Die vorgestellten Deadlockvermeidungsstrategien aus Abschnitt 4.4.2 zeigen im Vergleich beide gute Resultate. Eine Implementierung von Suchstrukturen für die Delaunay Triangulierung ist zu einer Verbesserung der tatsächlichen Laufzeit unbedingt nötig.

Die Experimente haben gezeigt, dass es im Allgemeinen kontraproduktiv für unsere Frameworks ist, die Anzahl der verwendeten Threads deutlich über die verfügbaren Rechenkerne zu steigern.

6. Zusammenfassung

Mit dieser Arbeit verfolgen wir zwei Ziele. Das erste Ziel ist die Entwicklung von Frameworks zur vereinfachten Parallelisierung von Divide und Conquer Algorithmen und inkrementell randomisierte Algorithmen. Das zweite Ziel ist der Nachweis von Leistungsgewinn durch die beschleunigte, parallele Ausführung von Algorithmen.

Wir fassen zunächst die Entwicklung der Frameworks für Divide und Conquer Algorithmen und inkrementell randomisierte Algorithmen kurz zusammen. Zu beiden Kategorien haben wir je ein Framework zur einfachen Parallelisierung vorgestellt. Beide sind durch ihren modularen Aufbau und der Verwendung von C++ Templates generisch.

In Kapitel 3 haben wir das Framework zur Parallelisierung von Divide und Conquer Algorithmen präsentiert. Das Framework besteht aus einem Solver, der die Struktur des Algorithmus implementiert, und einem Job, der eine konkrete Problemlösung implementiert. Die Jobs sind die Knoten des Rekursionsbaumes des Divide und Conquer Algorithmus. Die Ausführung der Jobs und damit der Rekursionsbaum wird automatisch parallelisiert. Durch die Unterstützung paralleler Funktionen auch innerhalb der Jobs sind weiter ausgearbeitete Parallelisierungen möglich. In Beispielen haben wir gezeigt, das sich vorhandener sequentieller Code leicht in das Framework einfügen lässt, sowohl zur Formulierung der Jobs als auch zur Lösung trivialer Problemgrößen. Triviale Jobs können mit beliebigen sequentiellen Algorithmen kombiniert und gelöst werden.

Die Verwaltung der Threads ist vollständig im gegebenen Solver gekapselt. So ist das Framework leicht zu bedienen. Das Framework skaliert gut über die Anzahl der Threads. Die Effizienz des Frameworks ist durch den positiven Vergleich mit der MCSTL Bibliothek belegt. Damit haben wir unsere Designziele erreicht.

Das Framework zur Parallelisierung randomisiert inkrementeller Algorithmen haben wir in Kapitel 4 vorgestellt. Das Frameworks kapselt die Verwaltung von Threads und die Balancierung der Threads in einer `Solver` Klasse, die von einem konkreten Algorithmus unabhängig ist. Der Solver konstruiert parallel eine threadsichere Datenstruktur. Die Datenstruktur implementiert die Funktionen, um sich selbst inkrementell zu erweitern und repräsentiert gleichzeitig die Lösung zu einem algorithmischen Problem. Die parallele Konstruktion von Datenstrukturen macht Synchronisationstechniken notwendig. Zur Implementierung der threadsicheren Datenstrukturen haben wir verschiedene Beispiele gegeben und Regeln für ein gutes Design aufgestellt. Dazu haben wir unsere leicht benutzbare Lösung zur Deadlock Vermeidung präsentiert.

Grundsätzlich gilt, dass zur Parallelisierung inkrementeller Algorithmen komplexer ist und potenziell mehr sequentieller Code überarbeitet werden muss, als dies bei Divide und Conquer Alogrithmen der Fall ist. Trotzdem konnten wir durch eine Modularisierung des Frameworks die Implementierung der Algorithmen vereinfachen und typische Probleme wie Deadlock Vermeidung und Lastverteilung entschärfen.

Durch das gelungene Design der Frameworks, konnten wir das erste Ziel der Arbeit erfüllen.

Das zweite Ziel ist der Nachweis von hartem Leistungsgewinn durch Parallelisierung. Wir messen den Gewinn der Parallelisierung durch Berechnung eines Beschleunigungsfaktors in Relation zur sequentiellen Ausführung. Beide Frameworks haben wir an verschiedenen Algorithmen aus dem Bereich Computational Geometry erprobt. Im Einzelnen sind dies:

- Sortieralgorithmen: Merge- und Quicksort.
- Konvexe Hülle: Giftwrapping, Quickhull und randomisiert inkrementeller Algorithmus.
- Triangulierung: Unterteilung von Dreiecken, Giftwrapping.
- Delaunay Triangulierung: Divide und Conquer und randomisiert inkrementeller Algorithmus.

Zur parallelen Implementierung dieser Algorithmen haben wir uns in Kapitel 2 verschiedene Werkzeuge und Techniken erarbeitet, die uns innerhalb unseres Framewoks als Ergänzung dienen. Dazu gehören verschiedene Synchronisationsmethoden (Gegenseitiger Ausschluss, lockfreie Synchronisation), threadsicheren Datenstrukturen (lockfreie Deque, threadbasierte Deque, Leda Graph) und parallelen Basisfunktionen (Extremwertsuche, Partitionieren).

Einen Ausschnitt der Ergebnisse unserer zahlreichen Experimente haben wir in Abschnitt 5 präsentiert.

Mit der Divide und Conquer Implementierungen verschiedener Algorithmen aus den Bereichen Sortieren, konvexe Hülle und Triangulierung haben wir gute bis sehr gute Beschleunigungswerte erzielt. Wir haben zudem unsere Implementierung des Quicksort Algorithmus mit der MCSTL Implementierung verglichen und erzielen bessere Beschleunigungswerte und Laufzeiten. Unsere erweiterten Konzepte zur Lösung kleiner Problemgrößen und zur Parallelisierung auf Funktionsebene zeigen in den meisten Fällen Wirkung bei der Verbesserung der Laufzeit oder Beschleunigungsfaktoren.

Das Framework für randomisiert inkrementelle Algorithmen haben wir mit Implementierungen zur konvexen Hülle und der Delaunay Triangulierung getestet. Das Framework ist in der Lage mit verschiedenen Implementierungen gute Beschleunigungswerte zu erzielen. Eine Implementierung von Suchstrukturen für die Delaunay Triangulierung ist zu einer Verbesserung der absoluten Laufzeit nötig.

Wir haben gezeigt, dass unsere Frameworks gute bis sehr gute Beschleunigungsfaktoren möglich machen.

Danksagung

Ich möchte meiner Familie, besonders meiner Frau Maria Antonietta für ihre Unterstützung bei Erstellung dieser Arbeit und meiner Tochter Sophia Elena für Ihre Motivationskunst, danken.
Weiter danke ich Steffen Knapp für seine wertvollen Eingaben und Prof. Stefan Näher für die Möglichkeit und die fachliche Unterstützung zur Promotion.

Literaturverzeichnis

[1] Richard M. Stallman and the GCC Developer Community
Using the GNU Compiler Collection,
GNU Press. 7, 21

[2] M. H. Austern
Generic programming and the STL,
Addison-Wesley, 2001.

[3] Ulrich Drepper, Ingo Molnar.
The Native POSIX Thread Library for Linux,
http://www.redhat.com/whitepapers/developer/POSIX_Linux_Threading.pdf 7, 12

[4] Stefan Näher, Daniel Schmitt.
A Framework for Multi-Core Implementations of Divide and Conquer Algorithms and its Application to the Convex Hull Problem,
Proceedings of the 20th Annual Canadian Conference on Computational Geometry, 2008. 46

[5] Stefan Näher, Daniel Schmitt.
Multi-core Implementations of Geometric Algorithms,
Efficient Algorithms 2009: 261-274. 46

[6] Kurt Mehlhorn, Stefan Näher.
The LEDA Platform for Combinatorial and Geometric Computing.
Cambridge University Press, 1999. 7, 46, 71, 78, 80, 83

[7] Algorithmic Solutions.
The LEDA User Manual, Version 6.3.
http://www.algorithmic-solutions.info/leda_manual/manual.html 34, 83, 89

[8] T. Cormen, C. Leiserson, R. Rivert, C. Stein
Introduction to Algorithms, Second Edition.
MIT Press 53, 55, 70

[9] Felix Putze, Peter Sanders, Johannes Singler
MCSTL: The Multi-Core Standard Template Library.
Technical Report, MCSTL Version 0.70-beta 8, 45, 48, 70, 95, 100

[10] *MCSTL: The Multi-Core Standard Template Library Homepage.*
http://algo2.iti.kit.edu/singler/mcstl/ 8

[11] David A. Bader, Bernard M.E. Moret, Peter Sanders
Algorithm Engineering for Parallel Computation.
Experimental Algorithmics, 2002 - Springer 7

[12] Peter Sanders
Algorithm Engineering – An Attempt at a Definition.
Efficient Algorithms, 2009, Lecture Notes in Computer Science, 321-340 7

[13] *The Message Passing Interface (MPI) standard.*
http://www.mcs.anl.gov/research/projects/mpi/ 8

[14] *Message Passing Interface Forum.*
http://www.mpi-forum.org/ 8

[15] *OpenMP Homepage.*
http://openmp.org/wp/ 8

[16] *Pthread API Dokumentation.*
http://cursuri.cs.pub.ro/ apc/2003/resources/pthreads/uguide/document.htm 7, 8, 11, 12, 46, 71

[17] *Pthread API Dokumentation.*
http://pubs.opengroup.org/onlinepubs/009695399/basedefs/pthread.h.html 12

[18] *Boost Homepage.*
http://www.boost.org/ 9

[19] *Online Buch zur Boost C++ Bibliothek.*
http://en.highscore.de/cpp/boost/ 9

[20] *Java SE 7 Online API Documentation*
http://docs.oracle.com/javase/7/docs/api/index.html 10

[21] *Threading Building Blocks Homepage*
http://threadingbuildingblocks.org/ 9

[22] *Intel(R) Threading Building Blocks Reference Manual*
Document Number 315415-016US, Intel Corporation. 9

[23] *Homepage des Microsoft Developer Networks*
http://msdn.microsoft.com/de-de/default.aspx 9

[24] E. G. Coffman, M. Elphick, A. Shoshani
System Deadlocks.
Computing Surveys, Vol 3, No 2, 1971 13

[25] M. Jones
What really happend on Mars?.
http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/mars_pathfinder.html 14

[26] H. Sundell, Philippas Tsigas
Lock-Free and Practical Deques using Single-Word Compare-And-Swap,
Technical Report no. 2004-02, Göteborg University 2004. 24, 28, 32, 33

[27] H. Sundell
Lock-Free and Practical Non-Blocking Data Structures,
Ph.D. Thesis, Göteborg University 2004. 28, 33, 39

[28] M. Michael, M. Scott
Correction of a memeory managment method for lock free data structures,
CS Department, University of Rochester, Tech. Rep., 1995. 33

[29] J. Valois
Lock-Free data structures,
Ph.D. dissertation, Rensselaer Polytechnic Institute, Troy, New York, 1995. 33

[30] P. Tsigas, Y. Zhang. A Simple,
Fast and Scalable Non-Blocking Concurrent FIFO queue for Shared Memory Multiprocessor Systems.
13th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '01), ACM. 28

[31] H. Sundell, P. Tsigas.
Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems.
Proceedings of IPDPS 03 in Nice (France) 22-26 April 2003. Rewarded with the Best Paper Award in the Algorithms category. 28

[32] H. Sundell, P. Tsigas.
Scalable and Lock-Free Concurrent Dictionaries.
Proceedings of SAC2004 in Nicosia (Cyprus) 14-17 March 2004. 28

[33] H. Sundell.
Wait-Free Reference Counting and Memory Management.
Technical Report no. 2004-10. Department of Computing Science. Chalmers University of Technology, November 2004. 28

[34] J. Valois
Lock-Free Linked Lists using Compare and Swap,
Rensselaer Polytechnic Institute, Troy, New York. 33

[35] M. Michael, M. Scott
Simple, Fast and Practical Non-Blocking and Blocking Concurrent Queue Algorithms,
CS Department, University of Rochester, Tech. Rep. 32

[36] M. Michael
CAS-based lock-free algorithm for shared deques,

Proceedings of the 9th International Euro-Par Conference, Springer Verlag, 2003. 32

[37] M. Herlihy
Wait-free synchronisation,
ACM Transactions on Programming Languages and Systems, vol. 11, p124-149, 1991. 20, 25

[38] N. Arora, R. Blumofe, C. Plaxton
Thread scheduling for multiprogrammed multiprocessors,
ACM Symposium on Parallel Algorithms and Architectures, p119-129, 1998. 32

[39] M. Greenwald
Non-blocking synchronization and system design,
Ph.D. dissertation, Stanford University, Palo Alto, CA, 1999. 32

[40] P. Martin, M. Moir, G. Steel
DCAS-based concurrent deques supporting bulk allocation,
Sun Microsystems, TEch. Rep. TR-2002-111, 2002. 32

[41] P. Martin, M. Moir, G. Steel et al.
DCAS is not a Silver Bullet for Nonblocking Algorithm Design,
SPAA'04, Barcelona, 2004. 25

[42] M. M. Michael
Safe memory reclamation for dynamic lock-free objects using atomic reads and writes,
In Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing, pages 21–30, July 2002. 25

[43] M. M. Michael
ABA Prevention Using Single-Word Instructions, IBM Research Division Report RC23089 (W0401-136) January 29, 2004 25, 26

[44] Danny Hendler, Nir Shavit, Lena Yerushalmi
A scalable lock-free stack algorithm,
SPAA '04 Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures, Pages 206 - 215 26

[45] Damian Dechev, Peter Pirkelbauer, Bjarne Stroustrup
Lock-free Dynamically Resizable Arrays,
OPODIS'06 Proceedings of the 10th international conference on Principles of Distributed Systems, Pages 142-156 26

[46] R. Seidel
Backward Analysis of Randomized Geometric Algorithms,
Trends in Discrete and Computational Geometry, 37-68, volume 10 of Algorithms and Combinatorics, 1992. 71

[47] K. L. Clarkson
New Applications of Random Sampling in Computational Geometry,
Discrete and Computational Geometry 4, 195-222, 1987. 71

[48] K. L. Clarkson
Algorithms for Diametral Pairs and Convex Hulls that are Optimal, Randomized and Incremental,
Proc. 4th ACM Symposium on Computational Geometry, 12-17, 1988. 71

[49] K. L. Clarkson, Peter W. Shor
Applications of Random Sampling in Computational Geometry II,
Discrete and Computational Geometry 4, 387-421, 1989. 58, 71

[50] Steven Fortune
Voronoi Diagrams and Delaunay Triangulations,
Handbook of discrete and computational geometry, 377 - 388, 1997. 69, 83

[51] Peter Su, Robert L. Scott Drysdale
A Comparison of Sequential Delaunay Triangulation Algorithms,
Comput. Geom. Theory Appl, 61-70, 1995. 69, 83

[52] Leonidas J. Guibas, Donald E. Knuth, Micha Sharir
Randomized Incremental Construction of Delaunay and Voronoi Diagrams,
Proceedings of the 17th int. colloquium on Automata, languages and programming, 414 - 431, 1990 83, 87

[53] Leonidas J. Guibas, Jorge Stolfi
Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams,
ACM Transactions on Graphics, Vol. 4, No 2, April 1985, Pages 74-123. 83

[54] Dani Lischinski
Incremental Delaunay Triangulation,
In Graphics Gems IV, Paul S. Heckbert, editor, Academic Press, 1994. 83

[55] Oliver Devillers
Improved Incremental Randomized Delaunay Triangulation,
In Proc. 14th Annu. ACM Sympos. Comput. Geom., pages 106-115, 1998. 87

[56] Oliver Devillers
The Delaunay Hierarchy,
J. Found. Comput. Sci. 13:163-180, 2002. 87

[57] Josef Kohout, Ivana Kolingerovà
Parallel Delaunay Triangulation Based on Circum-Circle Criterion,
SCCG '03 Proceedings of the 19th spring conference on Computer graphics, ACM New York, NY, USA 2003. 87

[58] M. Kallay
The Complexity of Incremental Convex Hull Algorithms.
Info. Proc. Letters 19, 197, 1984

[59] K. Mulmuley
A Fast Planar Partition Algorithm, I.
Proc. of the 29th FOCS, 1988.

[60] K. Mulmuley
A Fast Planar Partition Algorithm, II.
SCG '89 Proceedings of the fifth annual symposium on Computational geometry.

[61] N. Amenta, S. Choi, G. Rote
Incremental Constructions con BRIO.
SCG '03 Proceedings of the nineteenth annual symposium on Computational geometry. 87

[62] P. Tsigas, Y. Zhang
A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise 10000. 40

[63] A. Bykat
Convex hull of a finite set of points in two dimensions.
IPL, 7:296-298, 1978. 58

[64] W. F. Eddy
A new convex hull algorithm for planar sets.
ACM Trans. Math. Softw. 3, 1977. 58

[65] R.L. Graham
An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set.
Information Processing Letters 1, 132-133 (1972) 58

[66] A.M. Andrew
Another Efficient Algorithm for Convex Hulls in two dimensions.
Information Processing Letters 9, 216-219 (1979) 58

[67] F. P. Preparata, S. J. Hong
Convex hulls of finite sets of points in two and three dimensions.
Communications of the ACM, Volume 20 Issue 2, Feb. 1977 55

[68] Robert D. Blumofe, Charles E. Leiserson
Scheduling Multithreaded Computations by Work Stealing.
Journal of the ACM, Vol. 46, No 5, September 1999, pp. 720 - 748. 48, 70

[69] Umut A. Acar, Guy E. Blelloch, Robert D. Blumofe
The Data Locality of Work Stealing.
SPAA 2000, Bar Harbor, Maine USA 48, 70

[70] Guy E. Blelloch et al.
Scheduling Threads for Constructive Cache Sharing on CMPs.
SPAA 07, June 9-11, 2007, San Diego, California, USA 48

[71] Ivan J. Balaban
An optimal algorithm for finding segment intersections.
11th Computational Geometry, Vancouver, B. C. Canada 70

[72] L. Guibas, J. Stolfi
Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams.
ACM Transactions on Graphics, 4: 75-123, 1985 69

[73] *Prototype Javascript Framework.*
http://www.prototypejs.org/ 91

[74] *Protovis - A graphical approach to visualization.*
http://mbostock.github.com/protovis/ 93

[75] *d3.js - Data-Driven Documents.*
http://mbostock.github.com/d3/ 93

[76] *RGraph - HTML5 Javascript charts library.*
http://www.rgraph.net/ 93

[77] *Doxygen - Generate documentation from source code.*
http://www.doxygen.org 91, 133

Abbildungsverzeichnis

Tabellenverzeichnis

Anhang A.

Auszug aus aus der Dokumentation des Softwarepakets

Die Dokumentation wurde durch Doxygen [77] generiert und ist in englischer Sprache verfasst.

A.1. Multithread Frameworks for Divide and Conquer and incremental Algorithms with Geometric Background

Author

Daniel Schmitt

Version

1.0

Date

A.1.1. Installation Notes

This document is the reference manual for the Software Package "Multithread Frameworks for Divide and Conquer and incremental Algorithms with Geometric Background".

This packages brings to you C++ source code using the pthread library. It further requires a LEDA Installation that is compiled with multithread support.

The installation is simple. Copy the mcfad directory to your home directory or any other location you prefer. The package is not precompiled. Try to compile the programs in the mcfad/demos directory. For a succesful compilation open the file demos/Make.pro and set the following variables:

- LROOT the leda directory
- MCROOT the mcfad directory
- VERS an alternativ g++ compiler version
- WORD the flag -DWORD_LENGTH_64, if and only if you use a 64 bit system

After the options are set Call make in the mcfad/demos directory or any subdirectory and try to execute a demo programm.

To use the package in your code just include the necessary files in your source code. To compile your code have a look at mcfad/demos/Make.pro again.

Do not forget to set the following compiler options:

- set the LEDA include path
- set the LEDA library path
- set the mcfad include path
- link the thread library with the -pthread option
- define the flag -DLEDA_MULTI_THREAD for leda multi thread support
- define the flag -DWORD_LENGTH_64, if and only if you use a 64 bit system

It is only a reference manual! If you want to learn more read the thesis!

A.2. Examples - Framework Usage

A.2.1. The Divide and Conquer Framework

The framework consists of two parts: the solver and the job. A Job represents a problem instance, implements divide and conquer algorithm and also represents the solution.

The parallel solver manages the threads, administrates the job tree and distributes the work equally. There are different types of solvers, also serial solvers, see the dc_*_solver* classes.

A.2.1.1. Using the Framework - Quicksort

In the following example we instantiate a quicksort job, a parallel and a serial solver and .

1. The type iterator is a random access iterator. The variables s and e decribe the input as interval [s..e[.
2. Instantiation of the quicksort job with the input. The size of the job is e - s.
3. The limit gives the minimum size for jobs that are solved parallel. Test runs showed that 1/100 of the maximum size is an adequate value.
4. We instantiate either the standard serial solver 'dc_serial_solver_2' or a specialized serial solver 'qs_solver'. Specialized solvers can be more efficient, but they must be implemented for every job type.
5. At last we instantiate the parallel solvers. Parameters are the number of threads, maximal job size for parallel dolved jobs and the serial solver.
6. A call to the solvers solve functions starts the computation. Parameter is the job.

```
qsjob<iterator> qs(iterator s, iterator e);
int limit = (e - s) / 100;
//dc_serial_solver_2<iterator> > serial;
qs_solver<iterator> serial;
dc_parallel_solver_divide_2<iterator> > solver(thread_num, limit, serial);
solver.solve(qs);
```

A.2.1.2. Job Implementation - Quicksort

A job represents a problem and its solution. In the quicksort example the input set is given as an interval between two random access iterators. When the job is solved - that means the child jobs are solved and the 'merge' function returns - the interval is sorted.

The solver uses the following functions:

- 'size' returns the size of the job, here the elements of the iterator interval.
- 'is_leaf' checks if a problem instance is trivial, if the job ismaller than 3.
- 'handle_leaf' solves trivial problem instances.
- 'divide' partitions the input set of a quicksort job into two subsets which are the input of two new jobs. The solver can assign a number of threads to a job (secondary threads). If available, the divide function uses the additional threads to partition the set in parallel. Function get_threads() returns the number of assigned threads
- 'merge' merges jobs, here trivial.

```
// The Standard Quicksort Job Class.
// The qsjob class implements the Quicksort Algorithm for
// the use with the divide and conquer framework dc_frame.h.
// It inherits the interface MC_JOB.
// The template argument iterator must be random access iterator.
// There must be defined the '<' operator for the input type.
// The function divide uses the parallel partiton object partition
// if threads are available.
// The merge function is trivial and can be omitted.
template <class iterator>
class qsjob : public MC_JOB
{
  typedef qsjob<iterator> job;
  public:
    qsjob(){};
    // Initialize the interval [l..r[ to sort.
    qsjob(iterator l, iterator r) : left(l), right(r) {};

    // Random access iterators, specifying the interval
    // [left..right[ to sort.
    iterator left;
    iterator right;

    // The input type, E must implement the '<' operator.
    typedef typename iterator::value_type E;

    int       size()
    { return right - left; }

    bool      is_leaf()
    { return size() <= 3; }
```

```
        void      handle_leaf()
        { // Handle sets up to 3 values ... }

        // Divide step of the Quicksort algorithm.
        void divide(job& j1, job& j2)
        {
        // Init variables
        // ...

        // partitioning, parallel if possible and necessary
        if (get_threads() > 1 && size() > 10000)
          {
            // partition parallel with partition object.
            // ...
          }
        else
          {
            // partition serial
            // ...
          }

        // Configure child jobs
        // j is the position of the pivot element
        j1.left = left;
        j1.right = j;
        j2.left = j;
        j2.right = right;
        return;
      }

      // Trivial function to comply with solver object specifications
      void      merge(job& , job& ) {}
};
```

A.2.1.3. The Job Interface

Each divide and conquer should inherit the base class MC_JOB. It declares the virtual functions 'size', 'is_leaf' and 'handle_leaf'. The signatures of 'divide' and 'merge' depend on the choosen solver and are omitted. In the class are defines get/set functions for the number of secondary threads defined.

```
// The Base Class for Divide and Conquer Jobs.
// Each parallel solver objects expect jobs to come at
// least with the functions defined in this class.
// It is recommended to derive all job classes from MC_JOB.
// The basic functions are pure virtual.
// Parallel solver objects use the get/set_threads functions to
// supplement primary threads with additional secondary threads.
class MC_JOB
{
private:
  // Total number of secondary threads to use
  int num_threads;
```

```
public:
  MC_JOB(){num_threads = 1;};
  virtual ~MC_JOB(){};

  // Get/set for the total number of secondary threads to use.
  void set_threads(int i) {num_threads = i;} ;
  int get_threads() {return num_threads;};

  // The size of the job/Amount of work.
  virtual int          size() = 0;
  // True, if the job has a trivial size
  virtual   bool      is_leaf() = 0;
  // Solves jobs of trivial size
  virtual   void      handle_leaf() = 0;
};
```

A.2.2. The Randomized Incremental Framework

The randomized incremental framework consists of two parts, respectively classes. First a concurrent datastructure, that implements the a randomized incremental algorithm. Second the solver that administrates the threads and manages the input. There are incremental solvers as part of this package, the concurrent datastructure is to be programmed following the spezification, see subsection.

The concurrent datastructure must also implement a deadlock avoidance strategy. We implemented different mutexes with the Thread Hierachy Strategy. By that the detection of potential deadlocks is hidden in the mutexes, so that the datastructure must only provide a back-off function. Their use is shown in a later subsection.

A.2.2.1. Using the Framework - Convex Hull

Our example is the randomized incremental computation of the convex hull, that is part of this package. The code that calls the framework is simple. The only complex section can be the precomputation for the initialization of the concurrent data structure. In this case we must compute three non collinear points. We omit that part:

1. The function inc_convex gets the input data by a pair of random access iterators and the number of threads.

2. We compute three non collinear points to initialize the class hull_struct. This class is the concurrent datastructure that represents the convex hull.

3. We initialize an incremental solver. The default prallel solver is the class inc_-parallel_solver. Template parameters are the type of the input iterator and the concurrent datastructure. The constructor awaits the number of threads to use.

4. The solve function gets the datastructure and the input. It starts the parallel computation of the hull.

5. return_hull extracts from the concurrent datastructure the convex hull as a dobled linked list.

```
template <class iterator>
list<typename iterator::value_type> inc_convex(iterator s,
  iterator e, int thread_num)
  {
    typedef typename iterator::value_type POINT;
    list<POINT> CH;
    // Compute three non collinear points a,b,c
    // ...
    hull_struct<POINT> res_data(a,b,c);
    inc_parallel_solver<iterator,hull_struct<POINT> > ps(thread_num);
    ps.solve(res_data, s, e);

    CH = res_data.return_hull();
    return CH;
  }
```

A.2.2.2. Deadlock Avoidance with Thread Hierachy

An important task for the thread safe concurrent access of data is to handle deadlocks. Deadlocks can occure if a number of threads (processes) share the same objects (ressources) and may lock more than one object at a time.

Our strategy is to detect potential deadlocks with certain conditions during a lock operation. The return value of a lock operation indicates if the lock was taken successfully or was cancelled due to a deadlock violation (or the calling thread alredy owns the mutex). The strategy is named deadlock avoidance, the deadlock conditions are modelled with a thread hierachy. The package comes with different mutexes that implement a thread hierachy.

An algorithm that uses these mutexes must only catch the return values of the lock operation and in case of a potential deadlock call a backoff function. The backoff must release all locks and rollback the datastructure changes into a consistent state before.

```
// The return values of a lock or trylock operation.
enum lock_op {BLOCKED = 0, OWNER = -1,LOCKED = 1};

// A Mutex Class, using a Leda Spinlock Mutex to implement
// Thread Hierachy for Deadlock Avoidance.
// The mutex checks thread priorities with the thread id.
// If the mutex is locked and the deadlock condition does
// not hold, threads do busy waiting.
class da_mutex_spin
{
public:
  da_mutex_spin();
  ~da_mutex_spin();

// Locks the mutex.
// Returns OWNER if the calling thread already owns the lock,
```

```
  // BLOCKED if a potential deadlock is detected, LOCKED if succesful.
  int lock();

  // True if the calling thread already locks the mutex.
  bool my_lock();

  // True if the mutex is locked.
  bool locked();

  // Tries to lock the mutex.
  // Returns OWNER if the calling thread already owns the lock,
  // BLOCKED if blocked, LOCKED if succesful.
  int try_lock();

  // Unlocks the mutex.
  void unlock();
  };
```

A.2.2.3. A Concurrent Datastructure - Convex Hull

The concurrent datastructure of this example is named hull_struct. It represents the convex hull of the added points and provides an efficient search structure for unprocessed points. Details of the algorithm are omitted here.

The hull_struct is a container. It consist of three main parts:

1. The elements of type hull_element. The elements form a doubled linked list on the outside (the convex hull) and a search structure on the inside. The elements are lockable with a deadlock avoidance mutex.

2. The locate function. It locates for a point p a hull edge (hull_element) that sees p. The function uses the search structure of the hull_struct. The edges of the search structure are only read and therefore not locked. The return value true indicates that no deadlock occured.

3. The update function. It adds two new hull edges to the structure, the tangents of p and the former hull, if p lies outside. Otherwise it terminates immediately. To update the hull with new edges all changed edges are locked first, started at the located edge. In case of a deadlock the locks are released and update returns false.

Here is a shortend but still long code section of the convex hull example. For a short summary go to the next section.

```
// This class implements a thread safe datastructure and the
// parallel inremental convex hull algorithm.
// It uses thread hierachy to avoid deadlocks.
template <class point>
class hull_struct
{

  // An element of the hull respectively the search structure.
  // The element is a lockable directed edge.
```

```
  // The edges of the hull build a double linked list.
  struct hull_element
  {
    // Endpoints of the edge
    point   source;
    point   target;

    // True, if hull edge, else part of the search structure
    bool    outside;

    hull_element(const point& a, const point& b) :
                  source(a), target(b), outside(true) {}

    // Locks the mutex
    int  lock()     {return mutex.lock(); }
    // Tries to lock the mutex
    int try_lock() { return mutex.try_lock(); }
    // Unlocks the mutex
    void unlock()   { mutex.unlock();    }

    // The mutex
    da_mutex mutex;
    // Double linked list
    hull_element* succ;
    hull_element* pred;
  };

public:

  // An element of the structure
  typedef hull_element* elem_pointer;

  // Initiates the start triangle of the structure
  hull_struct(point& a, point& b, point& c)
  { //... }

  // Returns the hull as a double linked list.
  list<point> return_hull()
  { //... }

  // Locates a hull edge location that sees point p, if p is on the
  // outside. Otherwise location is Null.
  // Returns true if the operation was successfull, false if
  // a deadlock was avoided.
  // There is no lock operation in this case, so the return value is
  // always true.
  bool locate(point& p, elem_pointer& location)
  {
      int i = 0;
      location = 0;
      while (i < 3 && !right_turn(T[i]->source,T[i]->target,p) ) i++;
      if (i == 3) return true;

      hull_element* e = T[i];
```

```
    while (! e->outside)
    { hull_element* r0 = e->pred;
      if ( right_turn(r0->source,r0->target,p) ) e = r0;
      else { hull_element* r1 = e->succ;
            if ( right_turn(r1->source,r1->target,p) ) e = r1;
            else { e = nil; break; }
          }
    }

    location = e;
    return true;
}

// Extends the hull with point p, starting at edge location.
// Returns true if the operation was successfull, false if
// a deadlock was avoided.
bool update(point& p, elem_pointer& location)
{
    // if no edge was found, p is on the inside
    if (!location) return true;
    // Lock first edge
    if (location->lock() == BLOCKED) return false;

    hull_element* e = location;
    // check for race conditions
    if (! e->outside) { e->unlock(); return false; }

    hull_element* high = e;
    hull_element* low = e;

    // compute the upper tangent and lock the hull elements
    bool ok = true;

    do
    { high = high->succ;
       if (high->lock() == BLOCKED) { ok = false; break; }
    }
    while (orientation(high->source,high->target,p) <= 0);

    // backoff -> release all locks
    if (!ok)
    { while(high != low)
     { high = high->pred;
       high->unlock();
      }
     return false;
    }

    // compute the lower tangent
    // analog ...

    // add new tangents between low and high
    // ...
```

```
      // mark edges between low and high as "inside"
      // and define refinements and release all locks
      if (low != high) low->unlock();
      while (e != high)
      { hull_element* q = e->succ;
        e->pred  = e_l;
        e->succ = e_h;
        e->outside = false;
        e->unlock();
        e = q;
      }
      high->unlock();

    return true;
  }

  // Dummy
  void cleanup(elem_pointer&){};
  // Dummy
  void init(elem_pointer&){};

private:

  hull_element* T[3];
  hull_element* last_edge;
};
```

A.2.2.4. Spezification of Randomized Incremental Concurrent Datastructures

We summarize the interface of the concurrent datastructure. It must implement following four functions and one typedef:

- Define the type elem_pointer to be a pointer type to an element of the structure.
- Operation locate computes an element location of the datastructure to an input object p. The element location is in relation to the object p.
- Operation update extends the structure with object p by the use of element location.

Both locate and update must return false if a deadlock was avoided, true otherwise. In the deadlock case object p is later processed again.

There are two functions to acquire/release thread specific variables:

- Function init is called once per thread at its creation.
- Function cleanup is called once per thread at its deletion.

It is recommended to use these two functions with the pthread library functions 'pthread_setspecific' and 'pthread_getspecific'. They use the association of an object

pointer with a predefined key on a per thread level. The key is best defined in the constructor/destructor of the datastructure with the pthread functions 'pthread_key_-create' and 'pthread_key_delete'. In the previous example init and cleanup were not in use, just implemented as dummies.

```
// This class implements a thread safe datastructure for a
// randomized incremental algorithm.
// It is a container for structs of type specific_element
template <class data>
class concurrent_datastructure
{
  // An element of the structure
  struct specific_element;

public:

// Masks element of the structure for the framework
typedef specific_element* elem_pointer;

// Default Constructor
concurrent_datastructure();

// Locates element location as entry point for the update
// operation of data p.
// Returns true if the operation was successfull, false if
// a deadlock was avoided.
bool locate(data& p, elem_pointer& location);

// Extends the structure with data p, starting at element location.
// Returns true if the operation was successfull, false if
// a deadlock was avoided.
bool update(data& p, elem_pointer& location);

// Called one time by each thread before its first locate operation
void init(elem_pointer&);
// Called one time by each thread after its last update operation
void cleanup(elem_pointer&);
};
```

A.3. Class Index

A.3.1. Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

A.4. Class Index

A.4.1. Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

A.5. Class Documentation

A.5.1. dc_parallel_solver_2< Job > Class Template Reference

A Parallel Solver for Jobs that divide/merge Pairs of Jobs.
Inheritance diagram for dc_parallel_solver_2< Job >:

Public Member Functions

- dc_parallel_solver_2 (int thread_num_, int limit_, SOLVER< Job > &serial_)

  Initializes the solver with the number of threads to use, maximal size of parallel solvable job and a serial solver.
- virtual void solve (Job &j)

  Starts the execution of the divide and conquer algorithm given by the job.
- virtual void solve (list< Job ∗ > &L)

  Starts the execution of the divide and conquer algorithm given by the jobs in the list L in parallel.

Public Attributes

- float used_time_total

  Measured Real/Used Time of the Last Solve() Call.
- double real_time_total

  Measured Real/Used Time of the Last Solve() Call.

A.5.1.1. Detailed Description

template<class Job>class dc_parallel_solver_2< Job >

A Parallel Solver for Jobs that divide/merge Pairs of Jobs.
The documentation for this class was generated from the following file:

- frameworks/divide_conquer/dc_frame.h

A.5.2. dc_parallel_solver_3< Job > Class Template Reference

A Parallel Solver for Jobs that divide/merge Triples of Jobs.

Inheritance diagram for dc_parallel_solver_3< Job >:

Classes

- struct **threadArgs**

Public Member Functions

- dc_parallel_solver_3 (int thread_num_, int limit_, SOLVER< Job > &serial_)

  Initializes the solver with the number of threads to use, maximal size of parallel solvable job and a serial solver.
- virtual void solve (Job &j)

  Starts the execution of the divide and conquer algorithm given by the job.
- virtual void solve (list< Job ∗ > &L)

  Starts the execution of the divide and conquer algorithm given by the jobs in the list L in parallel.

Public Attributes

- float used_time_total

  Measured Real/Used Time of the Last Solve() Call.
- double real_time_total

  Measured Real/Used Time of the Last Solve() Call.

A.5.2.1. Detailed Description

template<class Job>class dc_parallel_solver_3< Job >

A Parallel Solver for Jobs that divide/merge Triples of Jobs.

The documentation for this class was generated from the following file:

- frameworks/divide_conquer/dc_frame.h

A.5.3. dc_parallel_solver_debug_2< Job > Class Template Reference

A Parallel Solver for Jobs that divide/merge Pairs of Jobs. Tree data is saved for Debugging.

Inheritance diagram for dc_parallel_solver_debug_2< Job >:

Public Member Functions

- dc_parallel_solver_debug_2 (int thread_num_, int limit_, SOLVER< Job > &serial_)

  Initializes the solver with the number of threads to use, maximal size of parallel solvable job and a serial solver.
- virtual void solve (Job &j)

  Starts the execution of the divide and conquer algorithm given by the job.
- virtual void solve (list< Job ∗ > &L)

  Starts the execution of the divide and conquer algorithm given by the jobs in the list L in parallel.
- void tree_info (int level=0)

  Prints information about the D&C tree until level is reached.

Public Attributes

- float used_time_total

  Measured Real/Used Time of the Last Solve() Call.
- double real_time_total

  Measured Real/Used Time of the Last Solve() Call.

A.5.3.1. Detailed Description

template<class Job>class dc_parallel_solver_debug_2< Job >

A Parallel Solver for Jobs that divide/merge Pairs of Jobs. Tree data is saved for Debugging.

The documentation for this class was generated from the following file:

- frameworks/divide_conquer/dc_frame.h

A.5.4. dc_parallel_solver_debug_3< Job > Class Template Reference

A Parallel Solver for Jobs that divide/merge Triples of Jobs. Tree data is saved for Debugging.

Inheritance diagram for dc_parallel_solver_debug_3< Job >:

Public Member Functions

- dc_parallel_solver_debug_3 (int thread_num_, int limit_, SOLVER< Job > &serial_)

  Initializes the solver with the number of threads to use, maximal size of parallel solvable job and a serial solver.
- virtual void solve (Job &j)

  Starts the execution of the divide and conquer algorithm given by the job.
- virtual void solve (list< Job ∗ > &L)

  Starts the execution of the divide and conquer algorithm given by the jobs in the list L in parallel.
- void tree_info (int level=0)

  Prints information about the D&C tree until level is reached.

Public Attributes

- float used_time_total

  Measured Real/Used Time of the Last Solve() Call.
- double real_time_total

  Measured Real/Used Time of the Last Solve() Call.

A.5.4.1. Detailed Description

template<class Job>class dc_parallel_solver_debug_3< Job >

A Parallel Solver for Jobs that divide/merge Triples of Jobs. Tree data is saved for Debugging.

The documentation for this class was generated from the following file:

- frameworks/divide_conquer/dc_frame.h

A.5.5. dc_parallel_solver_divide_2< Job > Class Template Reference

A Parallel Solver for Jobs that only divide into Pairs of Jobs.

Inheritance diagram for dc_parallel_solver_divide_2< Job >:

Public Member Functions

- dc_parallel_solver_divide_2 (int thread_num_, int limit_, SOLVER< Job > &serial_)

Initializes the solver with the number of threads to use, maximal size of parallel solvable job and a serial solver.

- virtual void solve (Job &j)

  Starts the execution of the divide and conquer algorithm given by the job.

- virtual void solve (list< Job ∗ > &L)

  Starts the execution of the divide and conquer algorithm given by the jobs in the list L in parallel.

Public Attributes

- float used_time_total

  Measured Real/Used Time of the Last Solve() Call.

- double real_time_total

  Measured Real/Used Time of the Last Solve() Call.

A.5.5.1. Detailed Description

template<class Job>class dc_parallel_solver_divide_2< Job >

A Parallel Solver for Jobs that only divide into Pairs of Jobs.

The documentation for this class was generated from the following file:

- frameworks/divide_conquer/dc_frame.h

A.5.6. dc_parallel_solver_divide_3< Job > Class Template Reference

A Parallel Solver for Jobs that only divide into Triples of Jobs.

Inheritance diagram for dc_parallel_solver_divide_3< Job >:

Public Member Functions

- dc_parallel_solver_divide_3 (int thread_num_, int limit_, SOLVER< Job > &serial_)

  Initializes the solver with the number of threads to use, maximal size of parallel solvable job and a serial solver.

- virtual void solve (Job &j)

  Starts the execution of the divide and conquer algorithm given by the job.

- virtual void solve (list< Job ∗ > &L)

  Starts the execution of the divide and conquer algorithm given by the jobs in the list L in parallel.

Public Attributes

- float used_time_total

  Measured Real/Used Time of the Last Solve() Call.
- double real_time_total

  Measured Real/Used Time of the Last Solve() Call.

A.5.6.1. Detailed Description

template<class Job>class dc_parallel_solver_divide_3< Job >

A Parallel Solver for Jobs that only divide into Triples of Jobs.
The documentation for this class was generated from the following file:

- frameworks/divide_conquer/dc_frame.h

A.5.7. dc_serial_it_solver< Job > Class Template Reference

A Iterativ Serial Solver for Jobs that divide/merge a List of Jobs.
Inheritance diagram for dc_serial_it_solver< Job >:

Public Member Functions

- void solve (Job &j)

  Starts the execution of the divide and conquer algorithm given by the job.

A.5.7.1. Detailed Description

template<class Job>class dc_serial_it_solver< Job >

A Iterativ Serial Solver for Jobs that divide/merge a List of Jobs.
The documentation for this class was generated from the following file:

- frameworks/divide_conquer/dc_frame.h

A.5.8. dc_serial_it_solver_2< Job > Class Template Reference

A Iterativ Serial Solver for Jobs that divide/merge Pairs of Jobs.
Inheritance diagram for dc_serial_it_solver_2< Job >:

Public Member Functions

- void solve (Job &j)

  Starts the execution of the divide and conquer algorithm given by the job.

A.5.8.1. Detailed Description

template<class Job>class dc_serial_it_solver_2< Job >

A Iterativ Serial Solver for Jobs that divide/merge Pairs of Jobs.
The documentation for this class was generated from the following file:

- frameworks/divide_conquer/dc_frame.h

A.5.9. dc_serial_solver< Job > Class Template Reference

A Recursive Serial Solver for Jobs that divide/merge a List of Jobs.
Inheritance diagram for dc_serial_solver< Job >:

Public Member Functions

- void solve (Job &j)

  Starts the execution of the divide and conquer algorithm given by the job.

A.5.9.1. Detailed Description

template<class Job>class dc_serial_solver< Job >

A Recursive Serial Solver for Jobs that divide/merge a List of Jobs.
The documentation for this class was generated from the following file:

- frameworks/divide_conquer/dc_frame.h

A.5.10. dc_serial_solver_2< Job > Class Template Reference

A Recursive Serial Solver for Jobs that divide/merge Pairs of Jobs.
Inheritance diagram for dc_serial_solver_2< Job >:

Public Member Functions

- void solve (Job &j)

  Starts the execution of the divide and conquer algorithm given by the job.

A.5.10.1. Detailed Description

template<class Job>class dc_serial_solver_2< Job >

A Recursive Serial Solver for Jobs that divide/merge Pairs of Jobs.
The documentation for this class was generated from the following file:

- frameworks/divide_conquer/dc_frame.h

A.5.11. dc_serial_solver_3< Job > Class Template Reference

A Recursive Serial Solver for Jobs that divide/merge Triples of Jobs.
Inheritance diagram for dc_serial_solver_3< Job >:

Public Member Functions

- void solve (Job &j)

  Starts the execution of the divide and conquer algorithm given by the job.

A.5.11.1. Detailed Description

template<class Job>class dc_serial_solver_3< Job >

A Recursive Serial Solver for Jobs that divide/merge Triples of Jobs.
The documentation for this class was generated from the following file:

- frameworks/divide_conquer/dc_frame.h

A.5.12. dc_serial_solver_divide_2< Job > Class Template Reference

A Recursive Serial Solver for Jobs that divide Pairs of Jobs.
Inheritance diagram for dc_serial_solver_divide_2< Job >:

Public Member Functions

- void solve (Job &j)

  Starts the execution of the divide and conquer algorithm given by the job.

A.5.12.1. Detailed Description

template<class Job>class dc_serial_solver_divide_2< Job >

A Recursive Serial Solver for Jobs that divide Pairs of Jobs.
The documentation for this class was generated from the following file:

- frameworks/divide_conquer/dc_frame.h

A.5.13. inc_parallel_solver< iterator, D > Class Template Reference

Incremental Parallel Solver for Random Access Iterators.

Public Member Functions

- inc_parallel_solver (int thread_num_=1)

  Initializes the solver with the number of threads to use.
- void solve (D &res, iterator s, iterator e)

  res is the datastructure to be build in parallel. Iterators [s...e[describe the input interval.

A.5.13.1. Detailed Description

template<class iterator, class D>class inc_parallel_solver< iterator, D >

Incremental Parallel Solver for Random Access Iterators.

The class constructs in parallel a datastructure of type D. D implements a randomized incremental algorithm. The template parameter iterator is a random access iterator. The input is evenly distributed on the created threads.

The documentation for this class was generated from the following file:

- frameworks/incremental/inc_frame.h

A.5.14. inc_parallel_solver_dar< iterator, D > Class Template Reference

Incremental Parallel Solver for Random Access Iterators with Dynamic Argument Reservation.

Public Member Functions

- inc_parallel_solver_dar (int thread_num_=1)

  Initializes the solver with the number of threads to use.
- void solve (D &res, iterator s_, iterator e_)

  res is the datastructure to be build in parallel. Iterators [s,e[describe the input.

A.5.14.1. Detailed Description

template<class iterator, class D>class inc_parallel_solver_dar< iterator, D >

Incremental Parallel Solver for Random Access Iterators with Dynamic Argument Reservation.

The class constructs in parallel a datastructure of type D. D implements a randomized incremental algorithm. The template parameter iterator is a random access iterator. The created threads are initialized with only a small subset of the input. Further arguments are assigned by the threads on demand.

The documentation for this class was generated from the following file:

- frameworks/incremental/inc_frame.h

A.5.15. inc_parallel_solver_exp< iterator, D > Class Template Reference

Incremental Parallel Solver for Random Access Iterators, Partially Derandomized.

Public Member Functions

- inc_parallel_solver_exp (int thread_num_=1)

  Initializes the solver with the number of threads to use.
- void solve (D &res, iterator s, iterator e)

  res is the datastructure to be build in parallel. Iterators [s,e[describe the input.

A.5.15.1. Detailed Description

template<class iterator, class D>class inc_parallel_solver_exp< iterator, D >

Incremental Parallel Solver for Random Access Iterators, Partially Derandomized.

The class constructs in parallel a datastructure of type D. D implements a randomized incremental algorithm. The template parameter iterator is a random access iterator. The input is sorted, evenly distributed on the created threads and permuted again.

The documentation for this class was generated from the following file:

- frameworks/incremental/inc_frame.h

A.5.16. inc_serial_solver< iterator, D > Class Template Reference

A Serial Solver for Randomized Incremental Datastructures.

Public Member Functions

- void solve (D &res, iterator s, iterator e)

  res is the datastructure to be build in parallel. Iterators [s,e[describe the input.

A.5.16.1. Detailed Description

template<class iterator, class D>class inc_serial_solver< iterator, D >

A Serial Solver for Randomized Incremental Datastructures.

The documentation for this class was generated from the following file:

- frameworks/incremental/inc_frame.h

A.5.17. MC_JOB Class Reference

The Base Class for Divide and Conquer Jobs.

Public Member Functions

- virtual int size ()=0

  The size of the job/Amount of work.
- virtual bool is_leaf ()=0

  True, if the job has a trivial size.
- virtual void handle_leaf ()=0

  Solves jobs of trivial size.
- **LEDA_MEMORY** (MC_JOB)

- void set_threads (int i)

  Get/set for the total number of secondary threads to use.
- int get_threads ()

  Get/set for the total number of secondary threads to use.

A.5.17.1. Detailed Description

The Base Class for Divide and Conquer Jobs.

Each parallel solver objects expect jobs to come at least with the functions defined in this class. It is recommended to derive all job classes from MC_JOB. The basic functions are pure virtual. Functions divide and merge are omitted because their signature depends on the solver. Parallel solver objects use the get/set_threads functions to supplement primary threads with additional secondary threads.

The documentation for this class was generated from the following file:

- frameworks/divide_conquer/dc_frame.h

A.5.18. SOLVER< Job > Class Template Reference

The Signature for a Solver Class.

Inheritance diagram for SOLVER< Job >:

Public Member Functions

- virtual ~SOLVER ()

  Destructor.
- virtual void solve (Job &)

  Starts the execution of the divide and conquer algorithm given by the job.
- bool **use_secondary** ()
- void **use_secondary** (bool b)

Protected Attributes

- bool **use_secondary_threads**

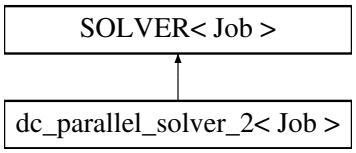
A.5.18.1. Detailed Description

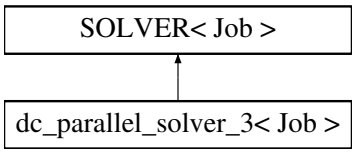
template<class Job>class SOLVER< Job >

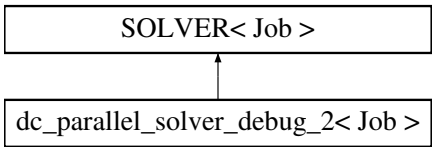
The Signature for a Solver Class.

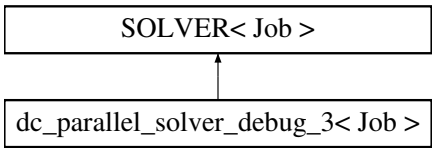
The documentation for this class was generated from the following file:

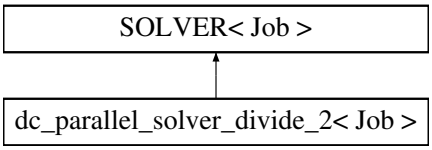
- frameworks/divide_conquer/dc_frame.h

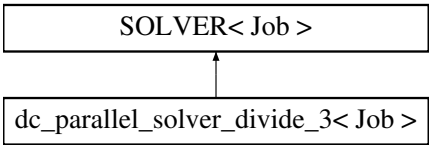

SOLVER< Job >
dc_parallel_solver_2< Job >

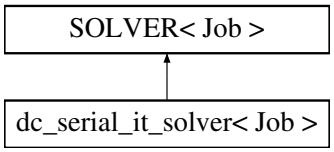

SOLVER< Job >
dc_parallel_solver_3< Job >

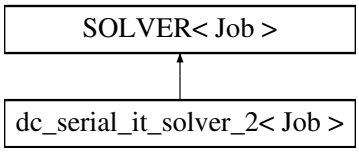

SOLVER< Job >
dc_parallel_solver_debug_2< Job >

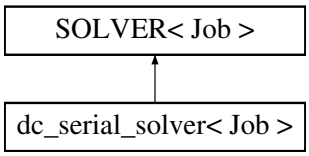

SOLVER< Job >
dc_parallel_solver_debug_3< Job >

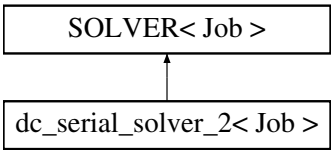

SOLVER< Job >
dc_parallel_solver_divide_2< Job >

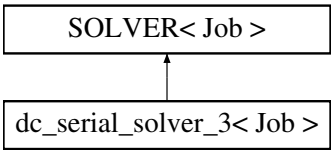

SOLVER< Job >
dc_parallel_solver_divide_3< Job >

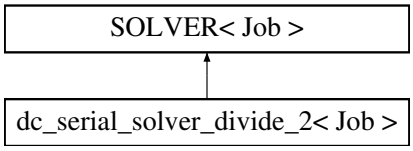

SOLVER< Job >
dc_serial_it_solver< Job >

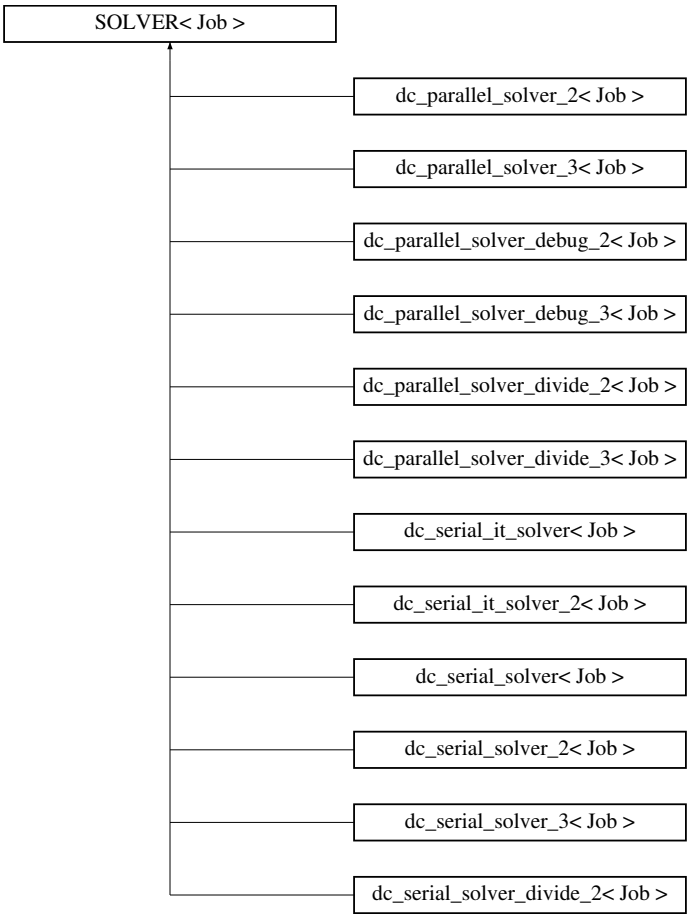

SOLVER< Job >
dc_serial_it_solver_2< Job >


SOLVER< Job >
dc_serial_solver< Job >


SOLVER< Job >
dc_serial_solver_2< Job >


SOLVER< Job >
dc_serial_solver_3< Job >


SOLVER< Job >
dc_serial_solver_divide_2< Job >


SOLVER< Job >
dc_parallel_solver_2< Job >
dc_parallel_solver_3< Job >
dc_parallel_solver_debug_2< Job >
dc_parallel_solver_debug_3< Job >
dc_parallel_solver_divide_2< Job >
dc_parallel_solver_divide_3< Job >
dc_serial_it_solver< Job >
dc_serial_it_solver_2< Job >
dc_serial_solver< Job >
dc_serial_solver_2< Job >
dc_serial_solver_3< Job >
dc_serial_solver_divide_2< Job >