Projekt: Parallelrechnerevaluation

Thema:Entwicklung eines Werkzeugs zur
Kommunikationsanalyse von MPI-Implementierungen

von:Paul Lindt
Betreuer:Julian Kunkel

Inhaltsverzeichnis

Motivation

Ziel der Projektes war es ein Werkzeug zu Entwickeln, welcher Aussagen über die Leistungsfähigkeit der Kommunikation sowie ihre Veränderung in Abhängigkeit von der Menge der zu kommunizierenden Daten erlauft.

Dadurch sollen Vergleiche unterschiedlicher Implementierungen des MPI-Standarts ermöglicht, sowie die Suche nach Fehlern in erleichtert werden.

Da die Ausführung von Programmen auf Clustern immer mit Kosten verbunden ist, sollten dabei die dem Programm zur Verfügung gestellten Ressourcen möglichst effizient eingesetzt werden.

Funktionsweise

Das Programm MNB analysiert zu beginn in welcher Umgebung es gestartet wurde. Mit Umgebung ist dabei, die Anzahl der Ranks, sowie ihre Verteilung auf Nodes gemeint. Dieses Wissen erlaubt es MNB einen Scheduler-Rank zu bestimmen, welcher in der Folge die Entscheidungen darüber, welcher Rank, welchen Test ausführt trifft.

Als nächstes wird die Konfigurationsdatei eingelesen, in welcher Festgelegt ist, welche Tests und in welcher Form auszuführen sind.

Bei der eigentlichen Ausführung versucht dann der Scheduler-Rank für alle Tests der Konfigurationsdatei eine gültige Kombination von Nodes und Ranks zu finden, und durch entsprechende Nachrichten an die anderen Ranks, welche in der Folge Tester heißen, ausführen zu lassen. Nachfolgen sind die einzelnen Bereiche genauer erklärt.

Environment

Environment beschreibt die Umgebung, die MPI dem Programm zur Verfügung stellt.

Da wir von MPI direkt nur die Menge der ausführenden Prozesse erfahren können muss zu beginn die Zuordnung von Prozessen/Ranks zu Hosts/Nodes geklärt werden. Dies geschieht dadurch dass jeder Prozess sich mit gethostname() den Namen des Rechners auf dem er ausgeführt wird besorgt. Anschließend tauschen alle Prozesse ihre Namen durch Aufruf von MPI_Gather untereinander aus. Auf Grundlage der Hostnamen erstellt jeder Prozess anschließend eine Liste von Nodes, von denen wiederum jeder über eine Liste der darauf laufenden Ranks verfügt.

```
typedef struct{
int numberOfNodes;
int numberOfRanks;
int schedulerRank;
MNB_Node * schedulerNode;
MNB_Node * localNode;
struct MNB_List * testerNodes;

}MNB_Environment;
```

Text 1: MNB_Environment realisirung

```
typedef struct{
char * hostname;
struct MNB_List * ranks;
enum MNB_NodeStatus status;
} MNB_Node;
```

Text 2: MNB_Node realisierung

Diese Liste ist nun zunächst Grundlage für die Auswahl des Schedulers, um die Verschwendung von Ressourcen zu minimieren fällt die Wahl auf den ersten Rank, des Nodes mit den wenigsten Ranks. Im weiteren Verlauf wird diese Liste auch fürs Scheduling selbst eingesetzt. Um Problemen beim dabei vorzubeugen wird nun zum einen der Scheduler-Rank aus der „ranks“ liste seines Nodes, sowie der Node Selber aus der Liste entfernt.

Programm intern finden die Repräsentation der Umgebungsinformationen durch die MNB_Environment Struktur statt, welche wiederum MNB_Node für die Darstellung der einzelnen Nodes einsetzt

Konfiguration

Die Information über die gewollten Tests müssen in Form einer Konfigurationsdatei bereitgestellt werden. Die Syntax dieser Datei wird durch libconfig, welche zum Einlesen benutzt wird, bestimmt. Die Konfigurationsdatei enthält Informationen über die Tests, die zum Testen eingesetzten Datenmengen sowie die Anforderungen an Nodes und Ranks mit denen die Tests ausgeführt werden sollen.

Beispiel einer Konfigurationsdatei:

```
node_configurations =
{
    conf1 = (2, 10);
};

tests =
{
  test1 = { conf = "conf1";     sizeconf = ("1M","10M");     tests="reduceTo0";};
};
```

Zu sehen ist hier eine minimale Konfigurationdatei, welche noch um die Bereiche size_configurations sowie test_collections erweitert werden kann, um die Lesbarkeit, in Fällen wo es viele Kombinationen von zu testenden Node-Konfigurationen, Datenmengen und Testfunktionen gibt, zu verbessern. Die Syntax ist kaum erklärungsbedürftig, um Missverständnisse zu vermeiden sollte aber erwähnt werden, dass bei einer Node-Konfiguration die erste Zahl die Anzahl der Nodes und die zweite die gewünschte Anszahl der Ranks angibt.
Für jeden Test muss mindestens eine Node-Konfiguration, eine Datengröße sowie eine Testfunktion angegeben werden. Die Angaben zu Datenmengen können mit M für Mebibyte = 2^{20} sowie K für Kibibyte = 2^{10} abgekürzt werden.

Wie bereits erwähnt wird zum einlesen der Konfigurationsdatei libconfig eingesetzt. Diese wird benutzt um die Konfigurationsdatei in ihrem Aufbau und Inhalt durch MNB_Setting Strukturen nachzubilden. MNB_Setting verfügt neben dem Namen und einem Zeigen auf den Wert ein Feld für den Typ eines Eintrags, welcher insbesondere auch den Wert MNB_LIST annehmen kann, um durch einfache oder geschweifte Klammern zusammengefasste Einträge der Konfigurationsdatei zu repräsentieren.

```
typedef struct {
    void * data;
    int type;
    char * name;
} MNB_Setting;
```

Text 3: MNB_Setting

Der nächste Schritt ist es dann die Referenzen Node-Konfigurationenen, Größen und mögliche Testkombinationen, welche unterhalb des „tests“ Eintrags Benutzt werden aufzulösen. Dies geschieht indem für jeden als Wert vom Typ String versucht wird ein MNB_Setting mit gleichem Namen zu finden.

Da die MNB_Setting Strukturen, welche die Tests repräsentieren nun nur noch Werte enthalten, werden MNB_Test Strukturen mit diesen Werten gefüllt.

MNB_Tests enthalten neben ihrem Namen, die für die Ausführung unverzichtbaren Felder für den Namen der Testfunktion, die Datenmengen, sowie die Konfiguration von Nodes und Ranks mit welcher dieser Test durchgeführt wird.
Die so entstehende Liste von MNB_Tests wird wichtigster Bestandteil einer MNB_Configuration Struktur, welche die Konfiguration für alle anderen Elemente des Programms darstellt.

```
typedef struct {
    char * name;
    char * test;
    struct MNB_List * sizes;
    struct MNB_List * nodeConf;
    int minRepeats;
    int maxDuration;
} MNB_Test;
```

Text 4: MNB_Test

Ausführung

Nachdem dass Programm weiß, welche Ressourcen ihm zur Verfügung stehen, sowie welche Tests ausgeführt werden sollen, kann die eigentliche Arbeit beginnen. Zunächst einmal scheiden sich die Ausführungspfade, des Schedulers sowie der Ranks, welche die Tests ausführen und im weiteren verlauf Tester heißen sollen.

Für alle Ranks beginnt die weitere Ausführung eines Kommunikators, welcher alle Tester-Ranks enthällt. Dies ist notwendig, da für die Ausführung von jedem Test ein eigener Kommunikator benötigt wird. Da aber jeder Rank des übergeordneten Kommunikators die MPI_Comm_create(...) Funktion aufrufen muss, müsste der Scheduler ebenfalls an der Erstellung des Kommunikators für jeden Test beteiligt sein. Damit dieses nicht notwendig ist, bekommen die Tester einen eigenen Kommunikator.

Scheduler

Der Scheduler Sortiert zunächst-einmal die auszuführenden Tests nach ihren Anforderungen an Nodes und Ranks, sowie die Nodes nach der Menge der auf ihnen laufenden Ranks um das nun folgende Scheduling zu erleichtern.

Das Scheduling funktioniert inden versucht wird Tests mit möglichst großen Node und Rank Anforderung zu finden, für deren Ausführung perfekt passende Nodes zur Verfügung stehen. Wenn für keinen Test mehr solch eine perfekte Konfiguration gefunden werden kann, ist das Scheduling bereit auch Node-Konfigurationen zu suchen, die zwar nicht mehr Perfekt passen, bei dennen die Abweichung aber zu verkraften ist. Beispielsweise würde dann ein Test welcher auf 8 Ranks , verteilt auf 2 Nodes ausgeführt wirden soll, auch 2 Nodes mit je 5 Ranks zugewiesen bekommen, wobei aber natuerlich bei der Ausführung zwei Ranks ignoriert werden würden.

Der Scheduler wiederholt die Suche so lange, wie noch neue Tests gefunden werden, die sich parallel ausführen lassen. Wenn dies nicht mehr der Fall ist, verpackt er die Informationen über Tests, sowie dafür vorgesehenen Ranks in eine Nachricht und verschickt diese an die Tester.

Anschließend macht er sich sofort wieder an die Arbeit um die nächste Gruppe von gleichzeitig ausführbaren Tests zu finden, damit die Tester möglichst keinen Augenblick ohne Arbeit verbringen

Tester

Die Tester laufen in einer Endlossschleiffe, in Erwartung von Nachrichten. Die jeweils ersten Ranks auf einem Node haben die besondere Rolle der Resender. Die zugrunde liegende Idee, hinter den Resendern, ist es die Menge an Daten, die übers Netzwerk verschickt werden, sowie die dafür benötigte Zeit zu verringern, indem die Resender die die Nachrichten an die auf dem gleichen Node laufenden Ranks, hoffentlich auf lokalem Weg, weiterleiten. Nachdem alle Tester mit Informationen versorgt sind, werten sie die Nachricht aus.Zunächst einmal findet eine Überprüfung des Nachrichtentyps statt. Nachrichten vom Typ MNB_QUIT_MESSAGE erlauben es dem Tester die Enschlossschleife zu verlassen und nach einer Aufräumphase beendet zu werden. Nachrichten vom Type MNB_EXECUTE_MESSAGE zwingen den Tester zu weiteren Aktionen. Zunächst einmal müssen alle Tester die für die Ausführung benötigten MPI_Comm's bzw, die dafür benötigten MPI_Group's erstellen. Dabei findet auch eine Überprüfung statt ob die Tester Teil eines der Kommunakatoren sind. Sollte dies der Fall sein, wissen sie automatisch auch, dass sie an der Ausführung der ihm zugeordneten Tests beteiligt sind.

Anschließend kann mit der eigentlichen Test Ausführung begonnen werden. Dazu wird zunächst einmal der der Testname benutzt um einen Zeiger auf die Funktion in welcher der Test und die Zeitmessung passiert zu erhalten. Für jede in der Nachricht enthaltene Größe wird daraufhin die Funktion mit der Mitgeschickten Mindestanzahl von Wiederholungen aufgerufen. Dabei werden die längste, minimale und durchschnittliche Dauer festgehalten. Nach jedem Test werden die Messungen aller beteiligten Ranks beim Rank 0 des Testkommunikators zusammengefasst und von ihm ausgegeben.

Die Testfunktion

```
typedef double (*testFunk)
   (long size, MPI_Comm comm);
```

Text 5: Typ der Testfunktion

Funktionen welche als Test eingesetzt werden, müssen einen Parameter für die Datenmenge, mit welcher der Test durchgeführt werden soll, sowie den alle am Test beteiligten Ranks enthaltenden Kommunikator akzeptieren, sowie die für die Ausführung benutzte Zeit zurückgeben. Die mitgelieferten Testfunktionen entsprechen den von MPI bereitgestellten Kommunikationsfunktionen.

Die Nachrichten

Die Kommunikation wird realisiert durch Nachrichten, welche als Abfolgen von elementaren Datentypen(int und char) durch MPI_Send/MPI_Recv verschickt werden. Diese Nachrichten enthalten insbesondere den bereits erwähnten Typ, sowie für jeden auszuführenden Test einen Block bestehend aus Header, welcher Informationen enthält, die dass strukturierte Senden und Empfangen erleichtern, sowie einem Body, welcher die eigentlichen Informationen, wie Name des Tests oder die Datenmengen mit denen er ausgeführt werden soll enthalten.

Die Ergebnisse

Die folgende Beispiel zeigt die vom Programm erzeugten Ausgaben für eine Konfigurationsdatei in der die Ausführung alle Tests mit einem Mebibyte an zu übertragenen Daten auf jeweils zehn Ranks, verteilt auf zwei Nodes. Für die Ausführung wurden 30Ranks verteilt auf 6 Nodes von MPI bereitgestellt. Die Ausführung fand unter Einsatz von MPICH2 auf dem cluster-neu des Arbeitsbereich Scientific Computing//Wissenschaftliches Rechnen statt.

[Clusterparameter einfügen? Clustergröße? 12 CPUs/Kerne 2666.756MHz RAM:9763MiB Netzwerkverbindungen?]

```
LD_LIBRARY_PATH    …  ;export _LMFILES_;
************************************************************
*             Working with 30 ranks on  6 nodes            *
*             0 is schedueler : multi Node                 *
************************************************************
```

```
Test:barrier, Nodes:2 Ranks:10, Size:1048576 byte, minDuration:0.000043s,
maxDuration:0.000075s, avgDuration:0.000059s

Test:reduceTo0, Nodes:2 Ranks:10, Size:1048576 byte, minDuration:0.001159s,
maxDuration:3.446198s, avgDuration:0.306842s

Test:sendToRoot, Nodes:2 Ranks:10, Size:1048576 byte, minDuration:0.000192s,
maxDuration:0.051193s, avgDuration:0.018077s

Test:allreduce, Nodes:2 Ranks:10, Size:1048576 byte, minDuration:0.015272s,
maxDuration:3.291662s, avgDuration:0.316154s

Test:bcast, Nodes:2 Ranks:10, Size:1048576 byte, minDuration:0.010863s,
maxDuration:0.022209s, avgDuration:0.013522s

Test:gather, Nodes:2 Ranks:10, Size:1048576 byte, minDuration:0.000217s,
maxDuration:0.052265s, avgDuration:0.015583s

Test:scatter, Nodes:2 Ranks:10, Size:1048576 byte, minDuration:0.017395s,
maxDuration:0.046083s, avgDuration:0.027125s

Test:allgather, Nodes:2 Ranks:10, Size:1048576 byte, minDuration:0.076220s,
maxDuration:0.115614s, avgDuration:0.092297s

Test:sendRecvRightNeighbour, Nodes:2 Ranks:10, Size:1048576 byte,
minDuration:0.000527s, maxDuration:0.017385s, avgDuration:0.004510s

Test:sendRecvToRoot, Nodes:2 Ranks:10, Size:1048576 byte, minDuration:0.000310s,
maxDuration:0.094294s, avgDuration:0.034711s

Test:sendRecvPaired, Nodes:2 Ranks:10, Size:1048576 byte, minDuration:0.000475s,
maxDuration:0.018697s, avgDuration:0.003020s
```

Entwicklungsprozess

In der langwierigen Entwicklungszeit des Projekts musste die ursprüngliche Vorstellung von der Funktionsweise von MNB immer wieder an angepasst werden um den Anforderungen von MPI gerecht zu werden, so sollten die Nodes nur über die Mitglieder der an ihrem Test beteiligten MPI-Gruppen informiert werden, allerdings war dieses Vorgehen nicht möglich da die Funktion zur Erstellung eines Teilkommunikator von allen Ranks/Prozessen, des übergeordneten Kommunikator aufgerufen werden muss. Dadurch ist es Notwendig jeden Rank über alle benötigten Gruppen zu informieren, wodurch wiederum die fürs Scheduling eingesetzte Menge an Daten, welche übers Netzwerk verschickt werden müssen, enorm gestiegen.

Des weiteren wurde deutlich, dass frühes Testen der Komponenten sehr wichtig ist um die fehlerfreie Funktionsweise zu ermöglichen, da ansonsten die Suche nach Fehlerquellen enorm erschwert wird. Neben der Komplexität, welche auch bei nur aus einem Prozess bestehen, kommt bei MPI erschwerend hinzu, dass zwar alle Prozesse den gleichen Programmcode ausführen sich aber in unterschiedlichen Phasen der Ausführung befinden können, sodass wenn Fehler auftreten, die Menge des Quellcodes, der fehlerursächlich sein kann nochmal sehr stark wächst.

Verbesserungsmöglichkeiten

Leider gibt es noch sehr weitreichende Möglichkeiten MPI Network Bench zu verbessern. Zum einen existiert die bereits angesprochene Problematik der Organisation der in Blöcken/Runden.

Eine Möglichkeit diese zu Umgehen wäre es alle möglichen Kommunikatoren zu Beginn der Ausführung zu erstellen. Allerdings ist dabei zu beachten, dass bei einer einfachen Implementierung, welche Tatsächlich alle Möglichen Kommunikatoren erstellt für n Ranks n! Kommunikatoren notwendig sind. Schon für 20 Ranks wären dies $2{,}432902008 \times 10^{18}$ Kommunikatoren.

Ebenfalls problematisch ist die „Verschwendung“ des Scheduler Rank, sowie aller auf dem Gleichen Host laufenden Prozesse. Da alle Prozesse sowohl über die Informationen über die Ressourcen als auch über die Gewünschten Tests verfügen sollte es möglich sein dass Scheduling von allen Prozessen gleichzeitig übernommen werden kann

Im Nachhinein muss auch die Zwischenschaltung der MNB_Setting Strukturen bei der Verarbeitung der Konfiguration, sowie die Nutzung der libconfig zu hinterfragt werden. .