Parallelisierung mit MPI und Threads auf dem Compute-Cluster dirac

Simulationsrechnungen auf Cluster-Rechnern können die Leistung der vielen Prozessoren nur nach einer Parallelisierung des Algorithmus nutzen. Das Programm wird dazu in unabhängige Teile zerlegt.
Bei einer wichtigen Klasse von Problemen (Finite Elemente, Partielle Differentialgleichungen) werden Modelle auf Datenpunkten in Zeitreihen gerechnet, wobei die Rechnung auf zwei Datenpunkten unabhängig und damit parallelisierbar ist, aber zwischen aufeinander folgenden Zeitschritten keine Unabhängigkeit besteht und Parallelisierung nicht möglich ist.

Datenzerlegung

Die Idee ist, die Datenpunkte des Modells verschiedenen Prozessen zuzuordnen. Eine Matrix kann man in Streifen oder Rechtecke aufteilen, einen Quader in Teilquader. Die Datenpunkte eines Teils gehören dann zu einem Prozess, der die Berechnung auf seinen Punkten vornimmt. Interessant sind die Randpunkte: Die muß jeder Prozess von den Nachbarprozessen abfragen, damit der Zusammenhang des Problems nicht verloren geht.

MPI

Eine Form der Kommunikation ist „Message Passing“, das dem „Message Passing Interface“ MPI seinen Namen gegeben hat. Prinzipiell können beliebig viele Prozesse mit MPI zusammenarbeiten. Die Daten werden abhängig von der Topologie des Clusters durch Kopie im Speicher oder Netzkommunikation übertragen. Bei dirac sind die Rechenknoten durch Gigabit-Ethernet verbunden.

Threads

Eine andere Form ist die Parallelisierung mit Threads, bei der alle Prozesse die Daten im Hauptspeicher sehen, aber mehrere Rechenkerne parallel arbeiten. Auf dirac arbeiten maximal 8 Threads physikalisch gleichzeitig auf 8 Rechenkernen (4 Dualcore CPUs) eines Knoten. Die Kommunikation zwischen den Threads geschieht hier implizit durch Hardware, dauert aber länger als im CPU-Cache.

Gitter

Wir betrachten zunächst feste Gitter. Hier ist eine Aufteilung günstig, bei der zu einer gegebenen Zahl von Prozessen möglichst wenige Randpunkte zu berücksichtigen sind, denn nach jedem Zeitschritt müssen die Werte der Randpunkte kommuniziert werden. Quader zerlegt man also am besten in Quader, Rechtecke in Rechtecke.
(Ein Würfel kann mit drei Flächen in 8 Teilwürfel, aber nur 4 Scheiben zerlegt werden.)

Erwartete Beschleunigung

Gegenüber der seriellen Berechnung kann man beim Einsatz von n Prozessen keine n-fache Beschleunigung der Berechnung erwarten, denn die Kommunikation kostet viel Zeit. Mit einer Verdoppelung der Prozessorzahl halbiert sich zwar die Rechenzeit für lokale Daten, aber die Anzahl der Randpunkte und die Zahl der Nachbarn wird größer. Alle Prozesse müssen ihren Teil berechnet haben, bevor der nächste Zeitschritt gerechnet werden kann – der langsamste Prozess bestimmt das Tempo. Selbst wenn man Flaschenhälse und Verklemmungen bei der Kommunikation der Prozesse

vermeidet, wird die Minimalzeit einer Datenübertragung und der Synchronisation schnell ein Problem.
Bei Message Passing zwischen Knoten über Ethernet spielt die Latenzzeit eine große Rolle. Für ein Modell mit zweidimensionaler Matrix auf n Prozessoren braucht man 8*n synchronisierte Datenpakete mit Randpunktwerten je Zeitschritt. Die Latenzzeit begrenzt hier die möglichen Zeitschritte je realer Sekunde auf grob geschätzt 100. Programme mit aufwendiger Berechnung je Zeitschritt profitieren also eher von MPI.

MPI Beispiel Poisson-Gleichung

Dem Buch Using MPI[1] ist das hier vorgeführte und angepasste Beispiel entnommen.
Eine partielle Differentialgleichung, die zweidimensionale Poisson-Gleichung, wird iterativ gelöst. Dabei soll die Summe der zweiten Ableitungen der gesuchten Funktion u einer Zielfunktion f gleichen. Die Ableitungen werden durch Differenzenquotienten angenähert, zu deren Berechnung man neben der Gitterschrittweite h zu einem Punkt u(i,j) die vier Nachbarn benötigt. Die Näherungsgleichung für die Matrizen u(m,n), f(m,n) mit m Spalten und n Zeilen:

$$1/h^2 * (u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1) – 4*u(i,j)) = f(i,j)$$

Die Randwerte für i=1,m und j=1,n sind vorgegeben. Die Lösung wird durch die Jacobi-Iteration gesucht. Von beliebigen Startwerten für die inneren Punkte ausgehend wird für k=1,2,...
gerechnet

$$u_{k+1}(i,j) = ¼ * (u_k(i-1,j) + u_k(i+1,j) + u_k(i,j-1) + u_k(i,j+1) – 4*u_k(i,j)\ h^2*f(i,j))$$

Das Näherungsverfahren wird beendet, wenn der Fehler

$$delta_{k+1} = \sum (u_{k+1}(i,j) – u_k(i,j))^2$$

kleiner als eine vorgegebene Schranke wird.

Parallelisierung bedeutet hier die Aufteilung der Matrix u in Teilmatrizen, die dann einzelnen Prozessen zugeordnet werden. MPI unterstützt die Programmierung in FORTRAN und C / C++.
Alle Prozesse führen dazu das gleiche Programm aus. Nach dem Programmstart erfragt jeder Prozess seine Prozessnummer und bestimmt mit MPI-Routinen seine Teilmatrix und die Kommunikationspartner, mit denen er die Werte der Randpunkte, hier auch „ghost points“ genannt, nach jedem Rechenschritt austauscht.
In der Hauptschleife des Fortran-Beispielprogramms werden die „ghost points“ asynchron während der Berechnung der inneren Punkte ausgetauscht und danach kollektiv die Fehlersumme bestimmt, um die Berechnung am Ziel zu beenden:

1. Empfangen der Randpunkte von vier Nachbarn beginnen
2. Versenden der Randpunkte an Nachbaren beginnen
3. innere Punkte berechnen
4. Empfangsende abwarten
5. eigene Randpunkte berechnen
6. Fehlersumme synchronisiert berechnen
7. Schleife abbrechen, wenn Fehlersumme kleiner als Schranke

Ein paar Tricks kommen dazu: Die Iteration wird mit zwei Matrizen a und b gerechnet, damit Daten für die Kommunikation nicht extra in Zwischenpuffer kopiert werden müssen; Zunächst wird b aus a berechnet, dann a aus b. Die Fehlersumme wird nur alle 64 Iterationen berechnet, denn das Verfahren konvergiert langsam und die synchronisierte Summenbildung ist zeitaufwändig.

MPI hat für alle Teilaufgaben passende Lösungen, insbesondere für die Problemzerlegung und die Bestimmung der Kommunikationspartner.

MPI_Init, MPI_Finalize	Anfang und Ende
MPI_Comm_Rank	Wer bin ich?
MPI_Comm_Size	Wie viele Prozesse?
MPI_Dims_Create	
MPI_Cart_Create, _Shift, _Get	Problemzerlegung
MPI_IRecv, MPI_ISend	asynchron empfangen und senden
MPI_Waitall	Ende asynchnoner Kommunikation abwarten
MPI_Barrier	alle Prozesse synchronisieren
MPI_WTIME	Zeitmessung

Mehrere Kommunkationspartner werden in Kommunikatoren bzw. ID-Argumenten zusammengefasst. Mit einem Funktionsaufruf werden Daten an möglicherweise sehr viele Partner berschickt. Randpunkte liegen im Speicher nicht immer sequentiell, und mit MPI-Routinen kann man leicht komplexere Datenstrukturen beschreiben, z.B. Ränder von Rechtecken oder Randflächen von Quadern, die dann von MPI serialisiert übertragen werden.

Fortran, C / C++

Mit Fortran können mehrdimensionale Martrizenrechnungen leichter formuliert werden, weshalb viele Numerikalgorithmen immer noch in Fortran formuliert werden. MPI bietet deshalb neben gleichwertigen Schnittstellen für C und C++ ein Fortran-API. Die Fortran-Aufrufe haben im Zweifel mindestens einen Parameter ierr mehr.

Unterschiedliche Architekturen

Mit MPI können Anwendungen Daten über Architekturgrenzen hinweg austauschen, eine eventuell nötige Konversion übernimmt MPI.

Ablaufsteuerung

Alle Prozesse sind prinzipiell gleichberechtigt, und ein Programm kann mit einem Prozess das gleiche Ergebnis wie mit n Prozessen erreichen. Wenn man einen speziellen Koordinator braucht, nimmt man meist den Prozess mit der Nummer 0, für spezielle Aufgaben z.B. den mit der höchsten verfügbaren Nummer, und führt entsprechende Programmteile in IF THEN ELSE Blöcken aus. Ein Kombination aus Datenaustausch und Berechnung bietet MPI_Allreduce; Alle Prozesse steuern Werte zum Ergebnis bei, das alle synchron erhalten.

Beispielprogramm

Sie finden das vollständige Beispiel auf dirac im Verzeichnis /home/dmf/SRC/MPI/Jacobi/
Die Problemgröße nx der Zeilen- und Spaltenzahl stellen Sie in der Datei param.f ein. Das serielle Programm ist `simple`, die parallele MPI-Version heißt `twod`.
`twod` braucht auf dirac minimal ein siebtel der Zeit von `simple` bei ungefähr 48 Prozessen. Mehr Prozesse durch höhere Kommunikationslast langsamer! Die Problemgröße nx spielt eine wichtige Rolle: Je größer die Matrix, desto mehr Prozesse können sinnvoll eingesetzt werden. Sinnvoll sind 16 oder 32 Prozesse, die dann 2 oder 4 Knoten belegen.

Übersetzung und Aufruf von MPI-Programmen auf dirac

MPI-Programme müssen mit speziellen Optionen übersetzt und gegen Hilfsbibliotheken gebunden werden, zur Laufzeit sorgt eine spezielle Umgebung für die parallele Ausführung.

Auf dirac ist LAM/MPI[2] installiert. Die Compiler mpicc, mpif77 und mpic++ (/opt/lam-7.1.2/bin/) sind sogenannte Wrapper für gcc, gf77 und g++ und definieren die nötigen Include-Pfade und zusätzlichen MPI-Bibliotheken. Parallele Programme werden grundsätzlich nur mit mpirun aufgerufen. Aufrufparameter von mpirun bestimmen, wie viele Prozesse gestartet werden. Das Kommando mpirun startet die parallelen Prozesse und wartet selber auf deren Ende. Mit mpirun kann man Programme nur starten, wenn zuvor die Laufzeitumgebung mit lamboot gestartet wurde.
Aber: Verschiedene Nutzer könnten mit mpirun beliebig viele Prozesse starten und sich ins Gehege kommen – und dann hätte niemand etwas von der Parallelisierung. Deshalb sollen parallele Programme wie alle anderen Dauerläufer hier mit der SUN Grid Engine[3] gestartet werden. Dabei wird eine Jobdatei mit dem Kommando qsub in die Warteschlange gestellt. Optionen der Jobdatei fordern eine bestimmte Zahl von Prozessen oder Knoten an. Die Grid Engine überwacht die Jobs und Last auf den Knoten. Der Job wird nur dann gestartet, wenn die geforderten Ressourcen vorhanden sind und reserviert werden können. Das Programm selbst wird dann in der Jobdatei mit mpirun gestartet. Beispieldatei mpijob

```
#! /bin/bash
# Grid Engine Parameter
#$ -S /bin/bash
#$ -cwd
#$ -pe lam 1-16

/opt/lam-7.1.2/bin/mpirun -np 16 /home/dmf/SRC/Jacobi/twod
```

Die Datei ist eine ausführbare Bash-Kommandodatei (chmod +x mpijob). Die mit #$ beginnenden Zeilen geben Parameter für die Grid Engine, hier u.a. -pe lam 1-16 damit 16 Prozessoren reserviert werden. Im Kommando mpirun selbst werden absolute Dateipfade verwendet (es gelten nicht die Standardpfade nach Login) und mit -np 16 werden 16 Prozesse gestartet.

Man stellt den Job mit

```
qsub mpijob
```

in die Queue. Wegen -pe lam wird das LAM/MPI Laufzeitsystem vor der Ausführung von mpirun gestartet. Nach Jobende findet man im Startverzeichnis (wegen -cwd, current working directory) die Dateien mpijob.o<nr> (Programmausgabe), mpijob.e<nr> (Fehlerausgabe), mpijob.po<nr> (Ausgabe des MPI Laufzeitsystems) und mpijob.pe<nr> (Fehler), <nr> ist eine fortlaufende Nummer, die der Job bekommen hat.
Die Auslastung im Cluster und die Jobs überwacht man mit dem grafischen Werkzeug `qmon`.
Gesundheit und Auslastung von dirac sieht man HMI-intern mit http://dirac im Browser.

Parallelisierung mit Threads

Zum Vergleich wurde der Algorithmus in C implementiert und die Parallelisierung durch Threads untersucht.
Die Idee ist hier eine Aufteilung in Master und Worker. Beim Start des Programms werden n Worker-Threads und ein Master-Thread gestartet. Mit einem Aufruf der C-Routine

doParallel(1,NX,f_sweep)

wird der Master mit einem Auftrag belegt, der Auftrag selbst steckt in der Funktionsreferenz f_sweep und den beiden Zusatzparametern 1 und NX. Der Master teilt den Indexbereich 1..NX in n Teilbereiche $1=na_1..ne_1$, $na_2..ne_2$, $na_n..ne_n=NX$ auf und beauftragt die Worker; Worker j ruft f_sweep(na_j,ne_j). Der Master wartet dann die erfolgreiche Ausführung in den Threads ab, und damit ist dieser Aufruf von doParallel beendet.
Im Beispiel werden die aktuellen Parameter von f_sweep durch globale Variablen ergänzt, die vor dem Aufruf von doParallel gesetzt werden, und dann natürlich von allen Threads gelesen werden können.
Die Problemzerlegung ist hier wesentlich einfacher als beim Message Passing. Es reicht bei Zeitschrittverfahren, die äußerste Schleife innerhalb der Berechnung eines Zeitschritts zu finden, und den Programmcode dieser Schleife in eine Funktion wie f_sweep zu kleiden.
Die Beschleunigung der Ausführung ist hier weniger von der Problemgöße NX abhängig. Als Daumenregel sollten so viele Worker, wie CPUs zur Verfügung stehen, angegeben werden, eventuell einer mehr, auf dirac also 4 (kleines NX) oder 5.

Durch Parallelisierung wird die Ausführung beschleunigt, wenn ein Job einen Knoten für sich hat. In der Jobdatei sollte also eine Option -l cpu=5 genügend CPUs anfordern. Wenn man das nicht tut, weil das Cluster schon sehr belegt ist, und auf dem Knoten kein anderes Thread-paralleles Programm läuft, kriegt man trotzdem mehr Leistung als von einer CPU ;-)

Implementierung

Das vollständige Beispiel findet man im Verzeichnis /home/dmf/SRC/Thread/

Makefile	
doParallel.h	
doParallel.c	Thread-Toolroutinen
twod.c	Poisson-Problem
job	Jobdatei für qsub

doParallel.c basiert auf POSIX-Threads und ist damit portabel. Unter Linux muss nichts besonders installiert werden.
twod.c benutzt die gleichen Tricks wie twod.f, die 2D Adressen werden allerdings in C selbst berechnet, und bestimmte Feldwerte in automatischen Variablen gehalten.
Übersetzt mit -DSERIAL wird das Programm „roh“ ohne Threads ausgeführt.

Die Anpassungen zur Parallelisierung in twod.c hervorgehoben:

```
#include "doParallel.c"

#define MAXITER 5000
#define NX 512
#define NB (NX+2)
#define NSIZE (NB*NB)
static const double cdelta = 1.0e-5;
static double a[NSIZE], b[NSIZE], pfun[NSIZE], darr[NB];

struct {
  double *a;
  double *b;
  int testit;
} s_sweep;

static inline double Sqr(double v) { return v*v;}

static void f_sweep(int j, int k) {
  const double h = 1.0 / (NX*NX);
  while (j <= k) {
    double dsum = 0;
    int i;
    double *ap = &s_sweep.a[j*NB];
    double *bp = &s_sweep.b[j*NB];
    double *fp = &pfun[j*NB];
    double lv, v = ap[0], rv = ap[1];
    for (i=1; i<=NX; i++) {
      ap++;
      lv = v;
      v = rv;
      rv = ap[1];
      bp[i] = 0.25 *(lv + rv + ap[-NB] + ap[NB]) - h * fp[i];
      if (s_sweep.testit)
        dsum += Sqr(bp[i] - v);
    }
    if (s_sweep.testit) darr[j] = dsum;
    j++;
  }
}

static void sweep (double *v, double *w, int testit) {
  s_sweep.a = v;
  s_sweep.b = w;
  s_sweep.testit = testit;
  doParallel(1,NX,f_sweep);
}

int main (int argc, char ** argv) {
..
  i = initParallel(nump);
..
  for (i=0; i<MAXITER; i++) {
    sweep(a,b,0);
    sweep(b,a,1);
    double d = darr[1];
    for (j=2; j<=NX; j++)
      d += darr[j];
    if (d < cdelta)
      break;
  }
..
  finishParallel();
..
}
```

Referenzen und weiterführende Links:

1. Using MPI, Portable Parallel Programming with the Message-Passing Interface second edition, William Gropp, Ewing Lusk, Anthony Skjellum, MIT Press
2. http://www.lam-mpi.org/
3. http://www.mpi-forum.org/
4. http://en.wikipedia.org/wiki/Message_Passing_Interface
5. http://en.wikipedia.org/wiki/POSIX_Threads
6. http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html (API Referenz)
7. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html