

MPI

Vortrag über MPI (Message Passing Interface) am Beispiel einer Waldbrandsimulation

im Seminar Parallele Algorithmen gehalten von Stanislav Velychko und
Daniela Tovar

Message Passing Paradigma

- In MPP werden Probleme auf Prozesse aufgeteilt, die getrennt voneinander an Teillösungen arbeiten
- Prozesse kommunizieren miteinander, da sie getrennten Speicherraum haben
- Andere Arten: gemeinsamer Speicher (Thread), SIMD, SPMD
 - Stellen wir aber nicht vor, kommen später in anderen Vorträgen (OpenMP, Cuda)

MPP – Vorteile

- Vorteile:
 - Einfaches konzeptionelles Modell für verteiltes Rechnen
 - Programmierer sind gezwungen, sich Gedanken über effiziente Algorithmen und Datenverteilung zu machen
 - Sonst zu hoher Kommunikationsaufwand
 - Hohe Datenlokalität
 - Oft gut skalierbare Programme

MPP - Nachteile

- Erhöhter Aufwand (Kommunikation zwischen Prozessen)
- Oft schwierig zu programmieren
 - Mehr Aufwand für den Entwurf des Algorithmus nötig

MPI

- MPI prominentester Vertreter des MPP
 - Schnittstellen existieren für C/C++ und Fortran
 - Inoffizielle für Python, Java u.a.
- MPI ist eine Bibliothek
- Verschiedene Implementierungen:
 - MPICH und MPICH2
 - OpenMPI
 - LAM/MPI (auf unseren Clustern)
 - Herstellerabhängige Implementierungen, z.B. HP-MPI

Mini-Referenz

- Programme kompilieren und ausführen:
 - `mpicc test.c -o testprogramm`
 - `mpirun -np 4 ./testprogramm`
- `MPI_Init` / `MPI_Finalize`
 - Initialisiert/beendet das MPI-Laufzeitsystem
 - MPI-Befehle dürfen nur dazwischen aufgerufen werden

Mini-Referenz

- `MPI_Comm_size`
 - Definiert einen Kommunikator
 - Kommunikatoren sind Prozessgruppen
 - Jeder Prozess im Kommunikator `MPI_COMM_WORLD`
- `MPI_Comm_rank`
 - Gibt Prozessnummer eines Prozesses in einem Kommunikator zurück
 - Achtung! Prozessnummern in unterschiedlichen Kommunikatoren sind nicht disjunkt, aber innerhalb eines jedem Kommunikator eindeutig

Mini-Referenz

- **MPI_Send**
 - Sendet Nachrichten an einen bestimmten Prozess
 - Nachrichtentyp und -Länge muss bestimmt sein
 - Nachrichten können auch an mehrere Prozesse geschickt werden
- **MPI_Recv**
 - Empfängt Nachrichten von einem bestimmten Prozess
 - Nachricht muss spezifiziert sein

MPI Datentypen

- Können mit MPI_Send und MPI_Recv verschickt werden
- Nur primitive Datentypen, wie in C/C++
- C-Datentyp wird mit Präfix MPI_CDATENTYP bezeichnet
 - z.B.: MPI_CHAR, MPI_INT, MPI_UNSIGNED_LONG
 - Achtung! kein String in den Standarddatentypen

Nutzerdefinierte Datentypen

- Eigene Datentypdefinition möglich
- `MPI_Type_struct`
 - Definiert neuen Datentyp
 - Wird mit `MPI_Type_commit` an das Laufzeitsystem übergeben
 - Wird mit `MPI_Type_free` wieder gelöscht

Blockierende Kommunikation

- Deadlocks
 - Jeder Prozess wartet auf Daten, keiner kann etwas empfangen (Bsp. An der Tafel)
- Überholen von Nachrichten über Dritte
 - Reihenfolge des Empfangs nicht garantiert
- Fairness
 - Die Arbeitslast ist nicht immer optimal verteilt

Kommunikationsarten

- Synchroner Kommunikation
 - Sendeoperation wird erst beendet, wenn korrespondierende Empfangsoperation gestartet wurde
 - MPI_Ssend
- Gepufferte Kommunikation
 - Nachricht wird in Buffer gespeichert, statt dass der Sender auf den Empfänger wartet
 - MPI_Bsend / MPI_Buffer_attach / MPI_Buffer_detach
 - Achtung! Buffer muss Nachricht fassen können

Kommunikationsarten

- “Bereit”-Modus
 - Sender weiss, dass Empfänger schon MPI_Recv aufgerufen hat
 - Vorteil: effiziente Implementierung möglich
 - Nachteile: Sender muss Information haben, dass Empfänger bereits MPI_Recv aufgerufen hat
 - Geringe praktische Bedeutung
 - MPI_Rsend

Nichtblockierende Kommunikation

- Gleichzeitig Senden/Empfangen mit Berechnungen
- Weder Sender noch Empfänger muss (im Idealfall) warten
- Nachrichten werden in Buffer zwischengespeichert
- Nachrichten werden mit einem Handle identifiziert
- Nichtblockierende Kommunikation muss immer beendet werden

Nichtblockierende Kommunikation

- MPI_Isend
 - Bekommt zusätzlich ein MPI_Request als Handle übergeben
 - Sonst wie MPI_Send
- MPI_Test
 - Prüft, ob der entsprechende MPI_Request abgeschlossen ist

Nichtblockierende Kommunikation

- `MPI_Wait`
 - Wird erst abgeschlossen, sobald der Sendebuffer wieder verwendet werden kann
- `MPI_Irecv`
 - analog zu `MPI_Isend`
- Analog zur blockierenden Kommunikation
 - `MPI_Issend`, `MPI_Ibsend`, `MPI_Irsend`

Nichtblockierende Kommunikation

- `MPI_Cancel`
 - Beendet Request
 - Request wird nicht mehr übertragen
- `MPI_Waitany` / `MPI_Waitall` / `MPI_Waitsome`
 - Wie `MPI_Wait` nur statt auf einen Request zu warten wird auf irgendeinen/alle/eine Gruppe gewartet

Kollektive Kommunikation

- Andere Form der datenparallelen Programmierung
 - Daten werden anders auf die Prozesse verteilt
- Alle Prozesse sind beteiligt
- Jeder Prozess ruft gleiche Funktion auf
- Vorteile:
 - Übersichtlichere Programme
 - Effizientere Implementierung möglich

Kollektive Kommunikation

- Nachteile
 - Kollektive und paarweise Kommunikation darf nicht gemischt werden
 - Es existiert weder Puffer- noch Synchron- noch “Bereit”-Modus
 - Keine nichtblockierende Kommunikation
 - Keine Unterstützung für Tags

Kollektive Kommunikation

- MPI_Barrier
 - Wartet, bis alle Prozesse MPI_Barrier aufgerufen haben
- MPI_Bcast
 - Versenden identischer Nachricht an alle Prozesse (vom root-Prozess)
- MPI_Scatter
 - Versenden unterschiedlicher Nachrichten an jedes Gruppenmitglied (vom root-Prozess)

Kollektive Kommunikation

- MPI_Reduce
 - Prozesse schicken Daten, die der Hauptprozess zu einem Gesamtergebnis kombiniert
- MPI_Gather
 - Prozesse schicken unterschiedliche Daten an den Hauptprozess
- MPI_Allgather
 - Prozesse schicken unterschiedliche Daten und erhalten von allen Prozessen Daten
 - Kein Hauptprozess!

Zeitmessung

- `MPI_Wtime`
 - Gibt an, wieviel Zeit (in Sekunden) seit einem (willkürlich festgelegten) Zeitpunkt vergangen ist.
- `MPI_Wtick`
 - Gibt Auflösung der Zeit zurück (von `MPI_Wtime`)
 - Je nachdem wie die Systemuhr hardwaremäßig implementiert ist

MPI-2

- Erweiterung von MPI-1 (1997)
- Thread-ähnliche Kommunikation (mit Locks)
 - 2 Prozesse kommunizieren über ein Fenster, das ein dritter Prozess freigibt (zeitkritische Abläufe)
- Dynamische Prozesserweiterung
 - Zur Laufzeit können neue Prozesse angelegt bzw. vorhandene gelöscht werden
- weitere Kommunikationsarten:
 - Einseitige Kommunikation (RMA = Remote Memory Access)

Zelluläre Automaten - Geschichte

- 1940 von Stanislaw Ulam erfunden
- Erweiterung durch John von Neumann zu einem universellen Berechnungsmodell
- 1970 John Horton Conways “Game of Life”
- Seitdem immer wieder neue interessante zelluläre Automaten (Simulation von Räuber-Beute-Systemem, Evolution, Flüssen, etc.)

Zelluläre Automaten

- Formale Definition:
 - Raum R
 - Oft 2 dimensionales Spielfeld
 - Endliche Nachbarschaft N
 - 8-Nachbarschaft
 - 4-Nachbarschaft
 - Zustandsmenge Q
 - Überföhrungsfunktion $Q^N \rightarrow Q$
 - Wird für alle Felder gleichzeitig ausgeführt

“Game of Life”

- Berühmtester zellulärer Automat
- Zellen haben Zustände “lebendig” und “tot”
- “tote” Zellen leben im nächsten Zustand, falls genau 3 Nachbarn leben
- “lebendige” Zellen sterben, falls weniger als 2 oder mehr als 3 Nachbarn leben

“Waldbrandmodell”-Regeln

- Leere Felder werden mit einer Wahrscheinlichkeit p_{Baum} im nächsten Zug mit einem Baum besetzt, mit Wahrscheinlichkeit $(1 - p_{\text{Baum}})$ bleiben sie leer
- Felder mit einem Brand werden im nächsten Zug als leer deklariert
- Felder mit Baum werden im nächsten Zug mit einem Brand besetzt, falls ein Nachbarfeld einen Brand hat oder sonst mit einer Wahrscheinlichkeit p_{Blitz} . Sonst bleibt ein Baum

“Waldbrandmodell” - sequentiell

- Lege Feld der Größe n,m an und fülle es zufällig
- Wiederhole bis abgebrochen wird
 - Iteriere über das Feld und bestimme für jede Position den neuen Wert ($n*m$ Positionsbestimmungen)

“Waldbrandmodell” - sequentiell

```
animate() {
    tmp = copy(feld); // bei der Berechnung wird das Feld benötigt
    for (i = 0; i < sizeX; i++)
        for (j = 0; j < sizeY; j++)
            tmp[i][j] = nextState(i,j);
    feld = tmp;
}
nextState() {
    switch (feld[i][j]) {
    case LEER: // mit pBaum Wahrscheinlichkeit entsteht ein Baum
        if (random < pBaum) return BAUM; else return LEER; break;
    case BRAND: // abgebrannte Bäume verschwinden
        return LEER; break;
    case BAUM: // Bäume können sich spontan entzünden...
        if (random < pBlitz) return BRAND;
        if (neighbourHasFire(i,j)) return BRAND; //und Feuer fangen
        else return BAUM; // sonst bleiben sie bestehen
        break;
    default: // sollte nie erreicht werden
        return ERROR; break;
    }
}
```

Master-Slave-Modell

- Master
 - Verteilt Arbeit auf Slaves
 - Übt Kontrolle aus
 - Berechnet nichts Kompliziertes
- Slaves
 - Führen komplexere Rechnungen durch
 - Warten auf Anweisungen vom Master

“Waldbrandmodell” - parallel Zerlegung der Daten

- Daten sollen so zerlegt werden, dass Prozesse unabhängig voneinander arbeiten können
- Für Berechnung ist aber immer die Nachbarschaft in Betracht zu ziehen
- Zerlegung der Daten wird an Tafel demonstriert

“Waldbrandmodell” - parallel

- init: Teile Matrix auf
- Prozesse:
 - Master:
 - Anzahl Slaves: teile Informationen auf (Formel im Quellcode), damit Last am Besten verteilt ist – Anzahl Blöcke ungleich Anzahl Slaves
 - Problem bei mehr Prozessen als Zeilen
 - Dann erwarte Daten von Slave-Prozessen (erst irgendwelche Kontrollinformationen, dann entsprechende Daten), bis alle Daten da sind
 - Wiederhole

“Waldbrandmodell” - parallel

- Slave:
 - erwarte Message mit Kontrollinformationen (zu bearbeitende Arraygröße und Zeilenposition zum Anfangen)
 - Dann erwarte Message mit Daten
 - Berechne Datensatz (Anzahl aufeinanderfolgender Zeilen) mit Grundlage je einer weiteren Zeile oberhalb/unterhalb (vorher von Master empfangen)
 - Sende Kontrollinformationen an Master
 - Sende neue Daten an Master
 - Wiederhole

Programmausschnitt

```
main():
-Kreiere Ein Feld mit Größe MxN;
-Fülle Feld mit zufallsgenerierten Werten ("1"-Baum, "0"-Leer, "5"-Feuer);

MPI_Init();

Master-Prozess:
- Wieviele Slaves habe ich zur Verfügung?
- Bestimmung der Blockgröße, die an die Slaves gesendet wird;

while (#Slaves > 0){
    - Auslesen der Informationen aus der Matrix und Abspeichern in einem
      1D-Array;
    - MPI_Send(info, 2, MPI_INT, i, tag1, MPI_COMM_WORLD);
    - MPI_Send(blocks_to_send, block_size, MPI_INT, i, tag2,
      MPI_COMM_WORLD);
}
while (#empf. Daten <= Slaves){
    - MPI_Recv(info, 2, MPI_INT, MPI_ANY_SOURCE, tag1, MPI_COMM_WORLD,
      &status);
    - MPI_Recv(blocks_to_recv, info[0], MPI_INT, k, tag2, MPI_COMM_WORLD,
      &status);
    - #empf. Daten++;
}
```

Programmausschnitt

Slaves:

- MPI_Recv(info, 2, MPI_INT, 0, tag1, MPI_COMM_WORLD, &status);
- Anlegen von Speicher für die zu erwartenden Informationen;
- MPI_Recv(blocks_to_send, block_size_recv, MPI_INT, 0, tag2, MPI_COMM_WORLD, &status);

- for(i = 0; i < block_size; i++){
 - Berechne für jedes Feld, was dort im nächsten Schritt sein wird;
- }

- MPI_Send(info, 2, MPI_INT, 0, tag1, MPI_COMM_WORLD);
- MPI_Send(blocks_to_recv, block_size, MPI_INT, 0, tag2, MPI_COMM_WORLD);

- Freigeben von Speicherplatz (erhaltene und gesendete Informationen)

- MPI_Finalize();

main:

- Freigabe des Speichers für die Matrix;

Demonstration

Wer jetzt eingeschlafen ist, kann einen Brand erleben!