10 Lastverteilung

Teilproblem: Eine Berechnung lasse sich parallel abarbeitbare *Teilprobleme* aufteilen

Lastverteilung: Was tun wenn Teilprobleme nicht gleich groß sind?

Klassifikation: Schwierig weil nur Teilaspekt der Parallelisierung

Statisch: Problemunabhängige a priori Zuteilung von Teilproblemen an PEs. Beispiel: Gausselimination zyklische statische Lastverteilung

Problemabhängig: Teilproblemgrößen hängen von Eingabe ab. Beispiel: Quicksort

Dynamisch: Verteilung zur Laufzeit. Adaptiv.

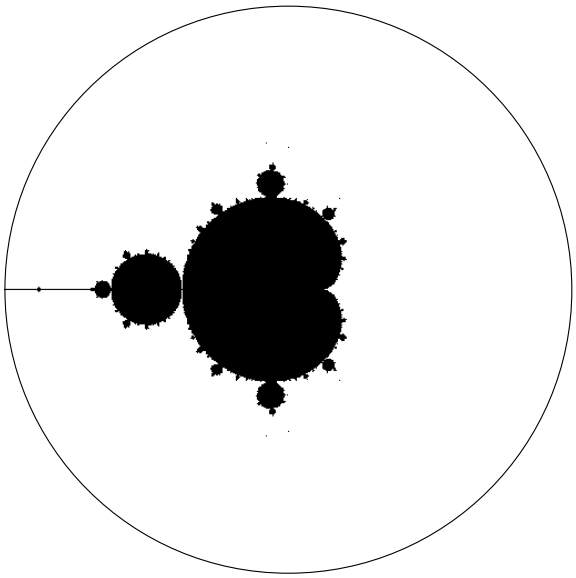
- Bei nicht vorhersagbaren Teilproblemgrößen
- Bei Clustern mit externer Last

10.1 Beispiel Mandelbrotmenge

$$z_c(m) : \mathbb{N} \to \mathbb{C}$$

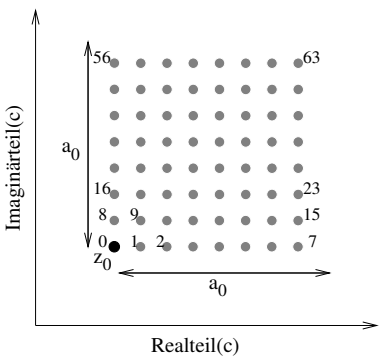
$$z_c(0) := 0, \quad z_c(m+1) := z_c(m)^2 + c$$

$$M := \{c \in \mathbb{C} \,|\, z_c(m) \text{ ist beschränkt}\} \ .$$



10.1.1 Angenäherte Berechnung

- Berechnung nur für quadratischen Ausschnitt der komplexen Zahlenebene
- Berechnung nur für diskretes Gitter von Punkten
- z_c unbeschränkt falls $|z_c(k)| \geq 2$
- Abbruch nach $m_{\max}$ Iterationen



Wo liegt das Lastverteilungsproblem?

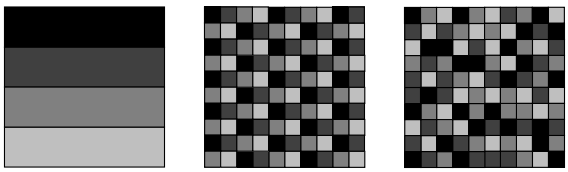
10.1.2 Code

```
int iterate(int pos, int resolution, double step
{
  int iter;
  complex c =
    z0 + complex((double)(pos % resolution)
                 * step,
                 (double)(pos / resolution)
                 * step);
  complex z = c;
  for (iter = 1;
       iter < maxiter && abs(z) <= LARGE;
       iter++)
  {
    z = z*z + c;
  }
  return iter;
}
```

10.1.3 Statische Äpfelverteilung

Da kaum Kommunikation stattfindet sind wir sehr flexibel

- Streifenweise Zerlegung
  - Warum attraktiv?
  - Warum besser nicht?
- zyklisch. Gut aber können wir das beweisen?
- Zufällig



10.2 Zufällige statische Lastverteilung

- n Teilprobleme
- Größen $l_1, \ldots, l_n$
- Maximale Größe $l_{\max} := \max_{i \le n} l_i$
- Gesamtlast $L := \sum_{i \le n} l_i$
- Zufällige Zuordnung
- $L_i := \sum_j l_j [\text{PE } i \text{ zuständig für } j]$

Satz 2.

$$\mathbf{Pr}[\max_i L_i \le (1+\epsilon)\frac{L}{P}] \ge 1 - P^{-\beta}$$

falls

$$\frac{L}{P l_{\max}} \ge 2(\beta+1)\frac{\ln P}{\epsilon^2 + \mathrm{O}(\epsilon^3)}$$

Consider an arbitrary but fixed PE j. It suffices to show that PE j receives load exceeding $(1+\epsilon)L/P$ with probability at most $P^{-(\beta+1)}$. Define the 0-1 random variable X_i to be one if and only if subproblem i is assigned to PE j. Let $Y := \sum_{i \le n} l_i X_i / l_{\max}$ denote the "normalized load" received by PE j. Since the X_i are independent and the weights $l_i/l_{\max}$ are in $(0,1]$ we can apply the weighted Chernoff bound from [**?**, Theorem 1] to see that

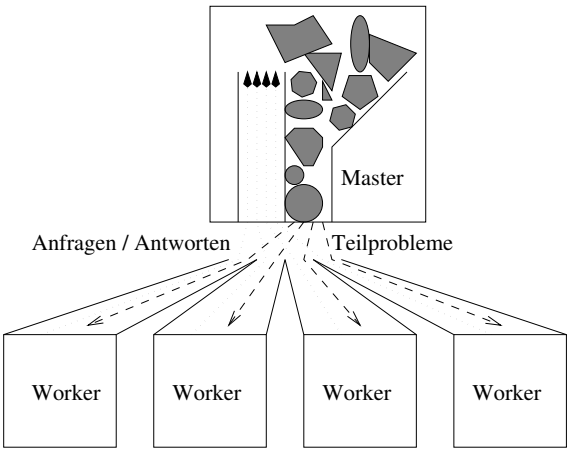
$$P_{\text{fail}} := \mathbf{Pr}[Y > (1+\epsilon)\mathbf{E}Y] \le \left(\frac{e^\epsilon}{(1+\epsilon)^{(1+\epsilon)}}\right)^{\mathbf{E}Y}.$$

10.2.1 Parallelisierung der Zuordnungsphase

- Wenn die Teilprobleme irgendwie auf die PEs verteilt sind: Zufallspermutation via all-to-all. (Siehe auch sample sort)
- Implizites Erzeugen der Einzelteile
  - Teilproblem läßt sich allein aus seiner Nummer $1 \ldots n$ erzeugen.
  - Problem: Parallele Berechnung einer (Pseudo)Zufallspermutation

Zufallsgeneratoren auf der Basis von linearen Kongruenzen lassen sich "vorspulen".

10.3 Das Master-Worker-Schema



10.3.1 Diskussion

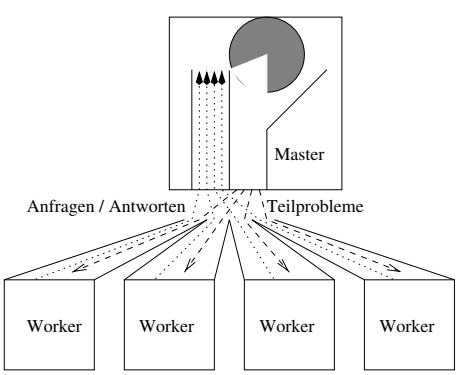
+ Einfach
+ Natürliches Ein- Ausgabeschema (aber u.U. gesonderter Plattensklave)
+ Leicht zu debuggen
– Kommunikationsengpaß
– Wie soll aufgespalten werden?
– Multilevelschemata sind kompliziert und nur begrenzt hilfreich

10.3.2 Größe der Teilprobleme

- Möglichst grosse Probleme abgeben solange Lastverteilung nicht gefährdet. Warum?
- Konservatives Kriterium: Eine *obere* Schranke für die Größe des abgegebenen Teilproblems sollte $1/P$-tel einer *unteren* Schranke für die sonst im System befindlichen Last nicht übersteigen.

– Woher Grössenabschätzung nehmen

- Aggressivere Verfahren ggf. sinnvoll



10.3.3 Worker Code

```
do {
  MPI_Receive(.....);
  task = .....;
  workerWork(task);
  MPI_Send(....); /* result and request */
} while (.....); /* not finished */
```

10.3.4 Einfacher Master Code ($P = 2$)

```
newWorkForWorker(0);
newWorkForWorker(1);
while (.....) {
  MPI_Recv(....., MPI_ANY_SOURCE, .....);
  .....
  /* process results and send new work */
}
```


10.3.5 Fairer (aber langsamer) Master Code

```
newWorkForWorker(0);
newWorkForWorker(1);
MPI_Irecv(....., &requests[0]);
MPI_Irecv(....., &requests[1]);
while (.....) {
   MPI_Waitsome(2, requests, &outcount,
                indices, statuses);
   for (i=0;  i < outcount;  i++) {
      /* process results and
         send new work for
         worker indices[i] */
      .....
      /* post new receive operation */
      MPI_Irecv(.....,
                &requests[indices[i]]);
   }
}
```