Alexander Schnapp
Max Menges

Intro HPC: Blatt 2

04.11.1014

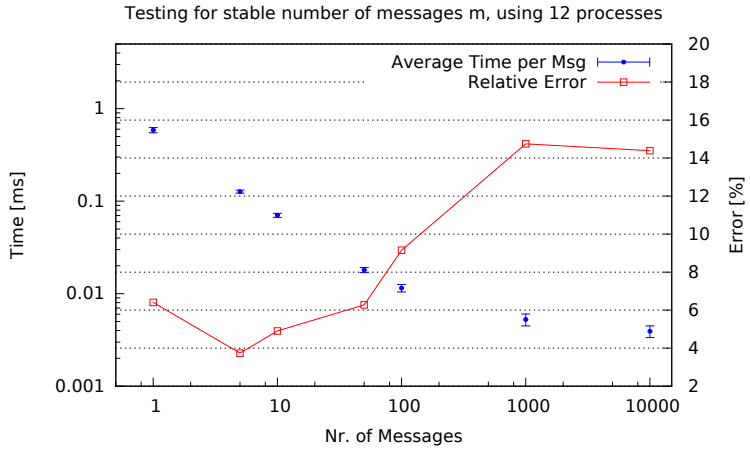
2.1 MPI Ring Communication

Der Quelltext zur aufgabe liegt unter `../2/2_1/2_1.cpp`. In dem Ordner liegt auch ein `Makefile` zum Compilieren und Ausführen. Benutzung:

`make`	Compile
`make clean`	Bin und `*.o` löschen
`make run proc=$p msg=$m v=$v`	Ausfrühren mit `$p` Prozessen, `$m` Nachrichten
`make run_opt proc=$p msg=$m v=$v`	Siehe `run`, aber mit optimiertem Mapping

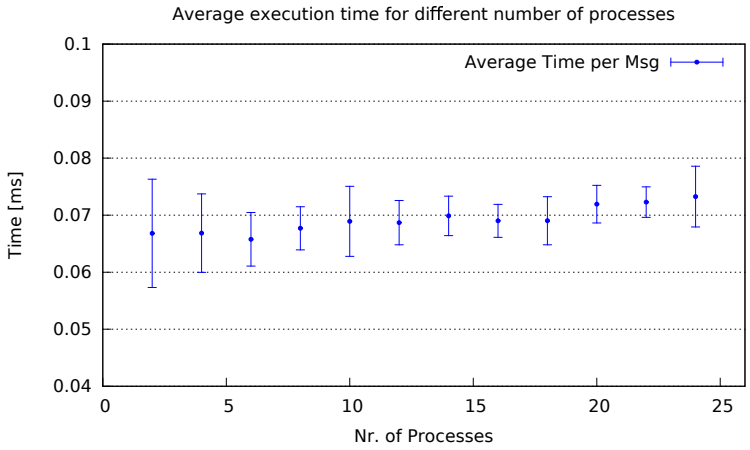
Parameter `$v` für Ausgabe der Zeit, `1` für nett formatiert, `0` nur gesamte Zeit und pro Nachricht.

Plot unten zeigt durchschnittliche Zeit pro Nachricht bei 12 Prozessen. Jeder run wurde 20 mal gemessen um den Fehler in der Laufzeit zu bestimmen (stabilste message size m). Obwohl 5 Nachrichten eine kleine Abweichung haben, haben wir für die Messung 10 Nachrichten genommen um ein wenig bessere Statistik machen zu können.



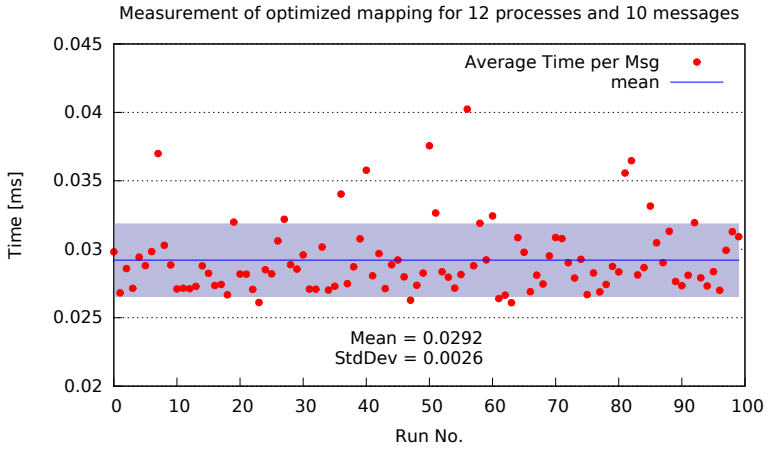
Messung der Ausführungsdauer bei 24 Prozessen, 10 Nachrichten pro Prozess und Mapping `creek01,..,creek08`. Das Programm wurde auch wieder für jede Prozessanzahl 20 mal ausgeführt. Die Ergebnisse sind unten geplottet. Die Zeit pro Nachricht steigt mit der Anzahl Prozesse leicht an und liegt bei $\approx 0.07\,\text{ms}$.

Für den letzen Teil der Aufgabe wurde das Mapping für bessere Performance geändert. Jeder Prozess sendet an seinen Nachfolger, da dieser durch das "normale"Mapping



jeweils auf einem anderen Rechner liegt, muss jede Nachricht einmal durchs Netzwerk, was Zeit kostet. Das Mapping wurde so angepasst, dass die ersten 8 Prozesse (`creek04` hat laut `nproc` 8 Cores) auf `creek04` laufen und die nächsten 4 auf `creek05`. Dadurch sind nur zwei Nachrichten über das Netzwerk nötig, alle anderen laufen nur von Core zu Core in der selben Maschine.

Dadurch verringert sich die Zeit pro Nachricht auf $0.0292 \pm 0.0026\,\text{ms}$ (siehe Plot). Das optimierte Mapping ist damit ≈ 2.35 fach schneller.



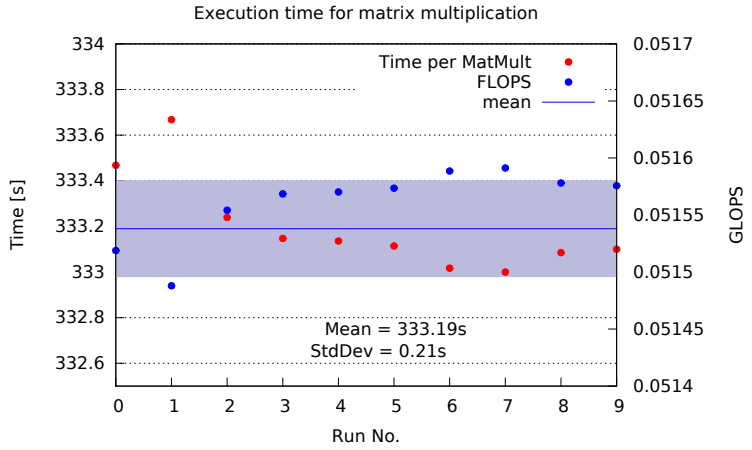
2.2 Barrier Synchronization

2.3 Matrix multiply – sequential version

Der Quelltext zur aufgabe liegt unter `../2/2_3/2_3.cpp`. In dem Ordner liegt auch ein `Makefile` zum Compilieren und Ausführen. Benutzung:

`make`	Compile
`make run n=$n runs=$x`	Matrix Multiplation mit $\$n * \$n$-Matritzen, `$x` mal ausgeführt.

Ausgeführt auf `creek04` braucht die nicht optimierte Matrix Multiplikation im Schnitt 333.19 s. Die Matrix Multiplikation besteht aus 3 ineinander verschachtelten Schleifen und bei jedem Durchlauf der inneren Schleife werden zwei floating point Operationen (1 `add`, 1 `mul`) ausgeführt, insgesamt also $2 \cdot n^3$ floating point Operationen pro Matrix Multiplikation. Bei $n = 2048$ ergibt dies $\approx 0.05155 GFLOPS$.



Intel gibt für den *Xeon E5-1620*[1] theoretische 115.2 GFLOPS bei 4 Cores an. Mit einem Core sollten somit noch immer noch 28.8 GFLOPS zu erreichen sein.

Der performance gap kommt von der größe der Matrix und der Zugriffszeit auf den Hauptspeicher bei einem Cache Miss. Da die Matrix nicht mehr komplett in den Cache passt (32 MB bei $n = 2048$ double-precision), muss ständig auf den Hauptspeicher zugegriffen werden. Da außerdem die B Matrix (bei $C = A \times B$) zeilenweise durchlaufen wird, werden – je nach Matrix und Cache-Line größe – nur ein oder wenige Einträge pro Cache-Line verwendet.

[1] `http://download.intel.com/support/processors/xeon/sb/xeon_E5-1600.pdf`