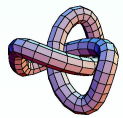
CHiC als CLiC-Nachfolger
– Erste Erfahrungen –

Matthias Pester
pester@mathematik.tu-chemnitz.de

Fakultät für Mathematik
Technische Universität Chemnitz

Symposium Wissenschaftlich-technisches Hochleistungsrechnen
23. März 2007

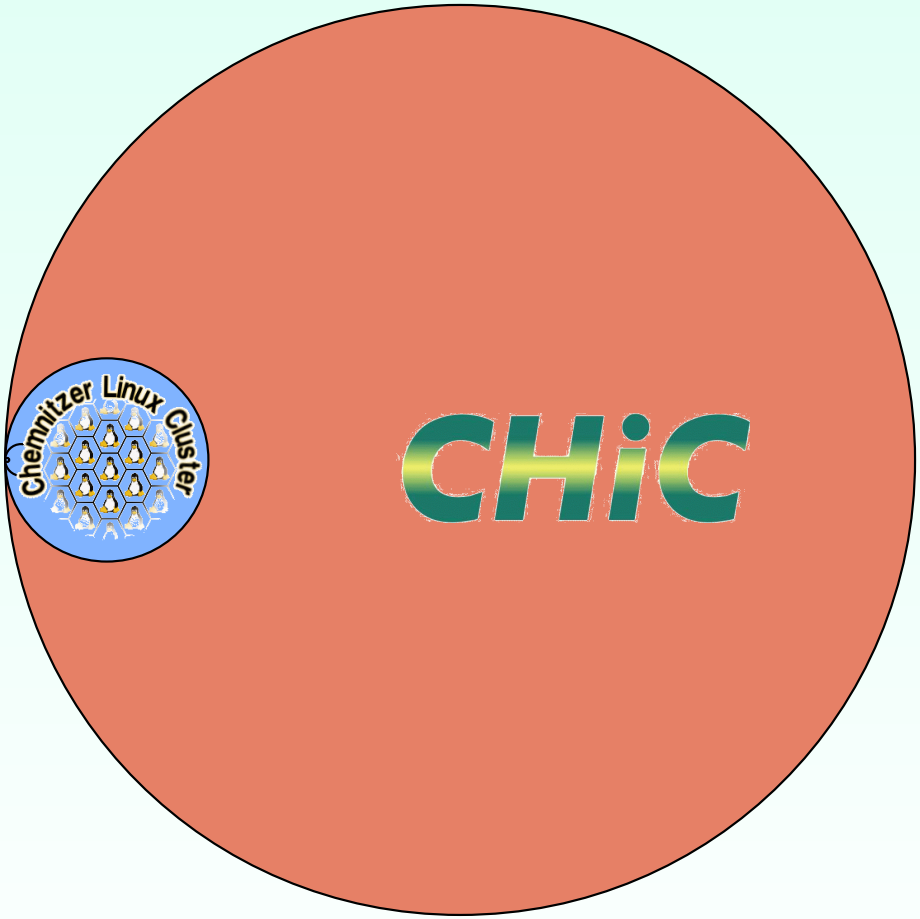


Inhalt

Meilensteine Chemnitzer Parallelrechner

Inbetriebnahme:		2007
2007: CHiC	538×	Opteron 4×2,6 GHz
2000: CLiC	528×	PIII-800 MHz,
1994: GC/PP	128×	PPC 601-80,
1992: Multicluster	32×	T800-20,

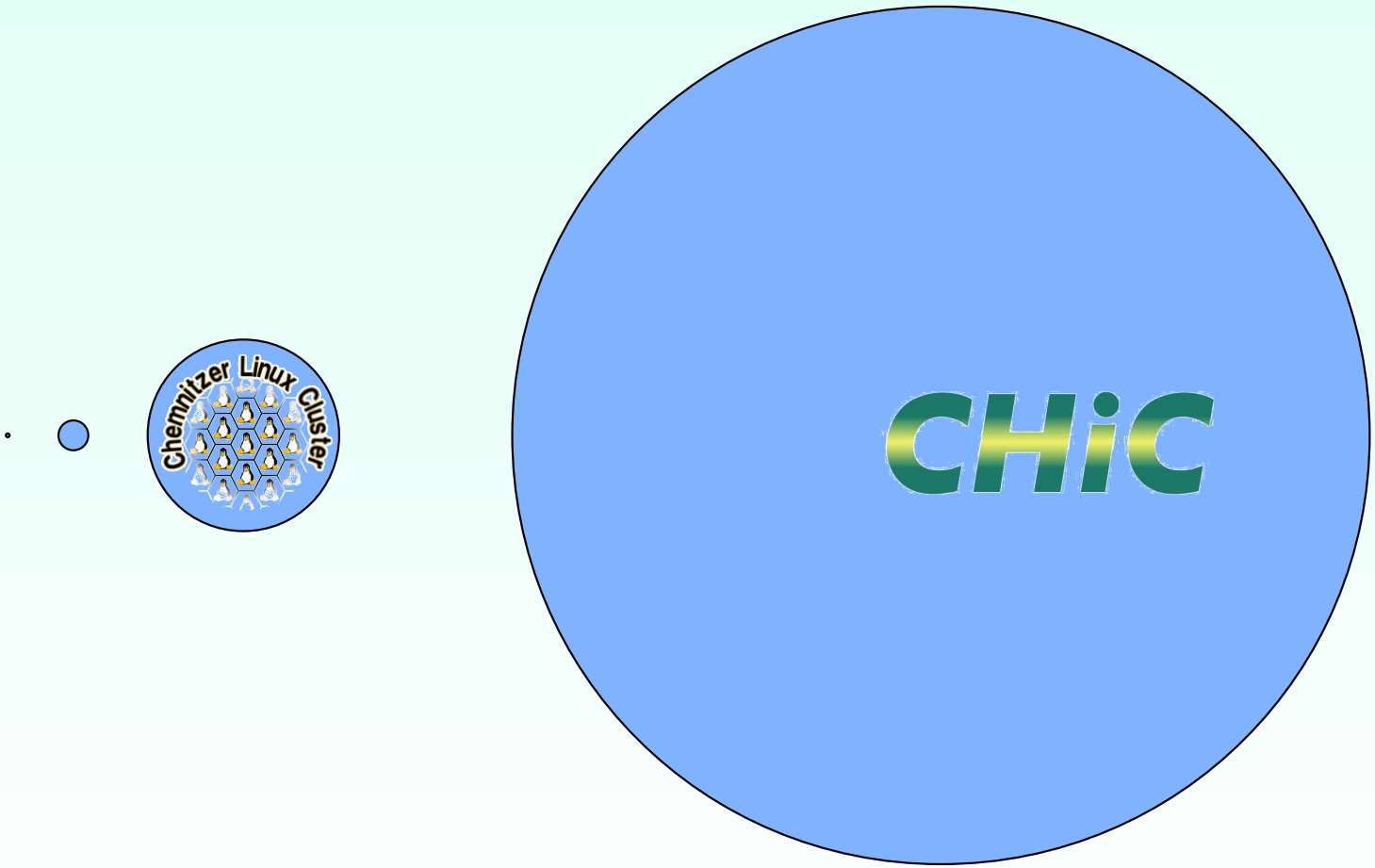
Generationsvergleich: Peak-Leistung ...



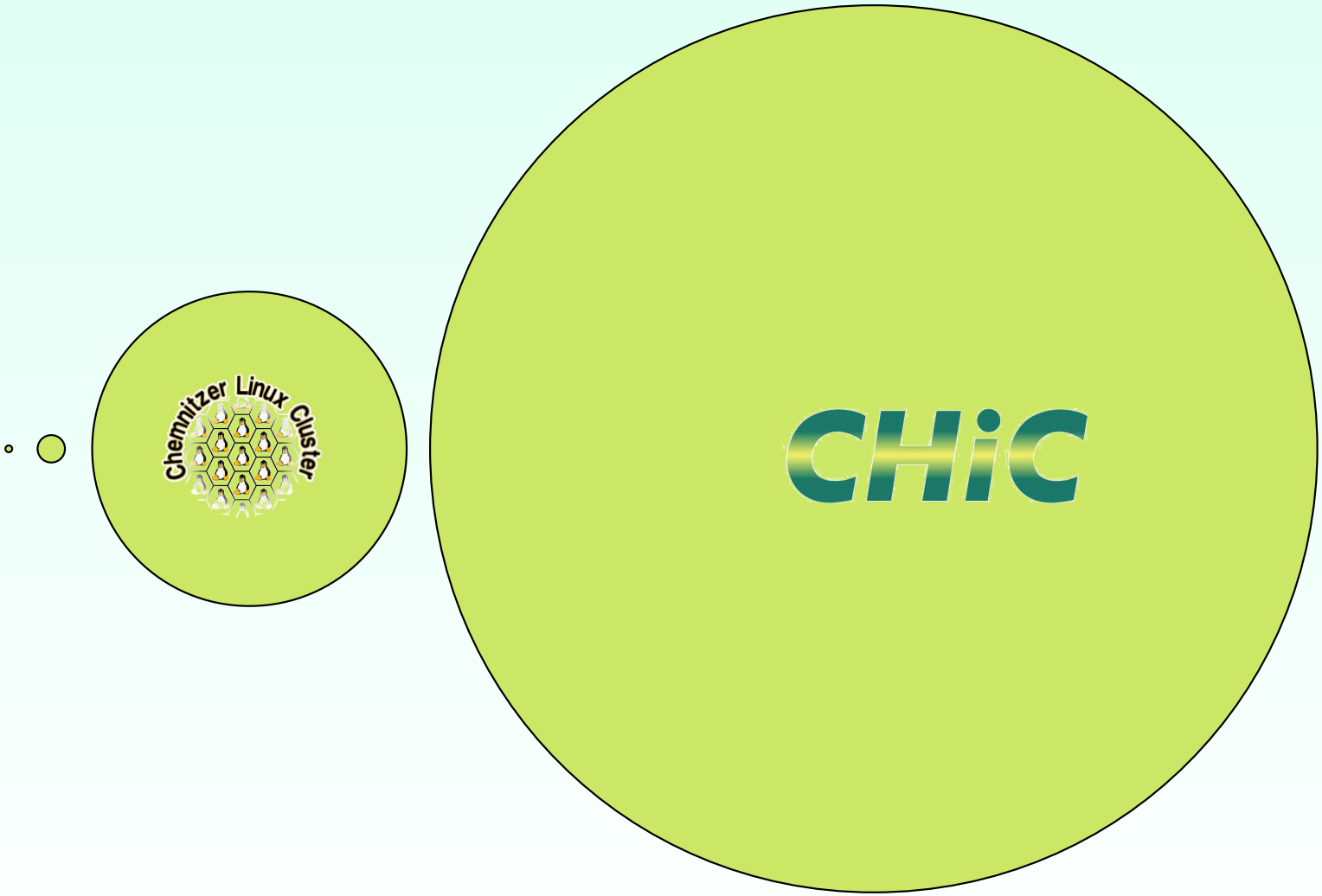
CPUs

- MC-32: 160 Mflops
- GC/PP: 10 Gflops
- CLiC: 422 Gflops
- CHiC: 8 Tflops

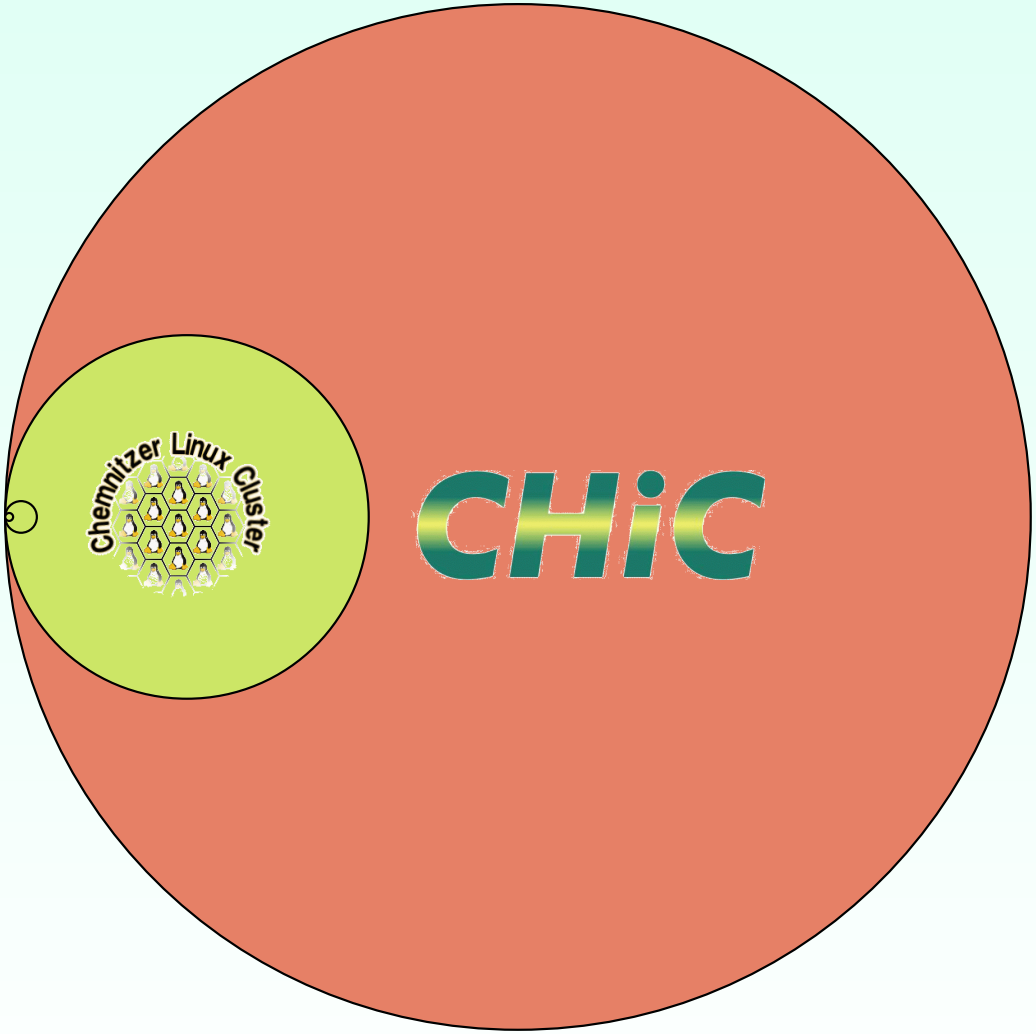
Generationsvergleich: Peak-Leistung . . .



. . . und Arbeitsspeicher



. . . und Arbeitsspeicher



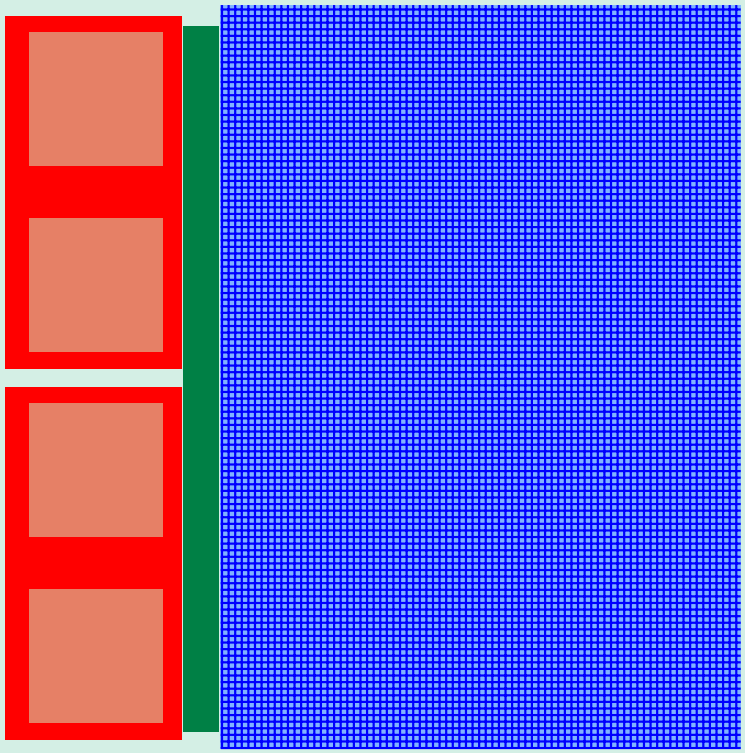
RAM

- MC-32: 128 MB
- GC/PP: 2 GB
- CLiC: 270 GB
- CHiC: 2 TB

Testumgebung

Die Prozessorknoten

2×AMD Opteron Dual Core



2 × 1 MB Cache 2 × 2 GB RAM

Compiler & MPI

- `mpif77`, `mpif90`, `mpicc`, ...
- MVAPICH → gfortran / gcc
- Open MPI → g77 / gcc
- Pathscale EKOPath Compiler Suite

Probleme (in der Startphase)

... richtige (beste) Compilerflags finden
... MPI-Versionen heißen 'gcc',
aber GNU-Fortran unterschiedlich,
... Fortran ist pedantischer geworden
... Shared Libraries auf Knoten, X11
... nicht-exklusive Knoten

Diverse Testsituationen

- Einzelprozessor = ein Knoten, nur eine CPU (von „4“)
- `mpirun -np 64 ...`
  - 64 Knoten, je 1 CPU (bis 4 GByte)
  - 32 Knoten, je 2 CPU's (bis 2 GByte)
  - 16 Knoten, je 4 „CPU-Kerne“ (bis 1 GByte)
- je 2 verschiedene MPI-Versionen (MVAPICH, Open MPI)
- je 2 verschiedene Kommunikationsbibliotheken
  - `MPIcom` - globale Kommunikation über `MPI_Allreduce` usw.
  - `MPIcubecom` - Hypercube-Routinen mit `MPI_sendrecv`

- Zeitmessungen werden erschwert durch laufende oder hängende Prozesse (eigene oder fremde)

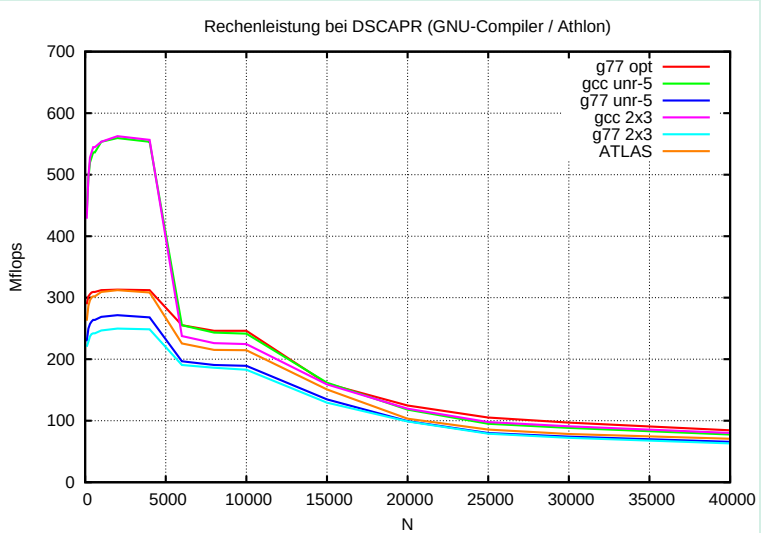
Rechenleistung auf einem Prozessorknoten

(1) Skalarproduktberechnung

- Berechne (k_N-mal) $s = \sum_{i=1}^{N} x_i y_i$
- für verschiedene Vektorlängen $N = 100, \ldots, 100\,000, \ldots,$ und $k_N \cdot N \approx const$.
- verschiedene Programmversionen (einfach, unrolled loops; C, Fortran)
- Bestimme Mflops aus der Rechenzeit und zeige
- Abhängigkeit von Speicherzugriffen (für kleine N fast nur Cache)

Zum Vergleich:

Athlon-500 (GNU-Compiler)



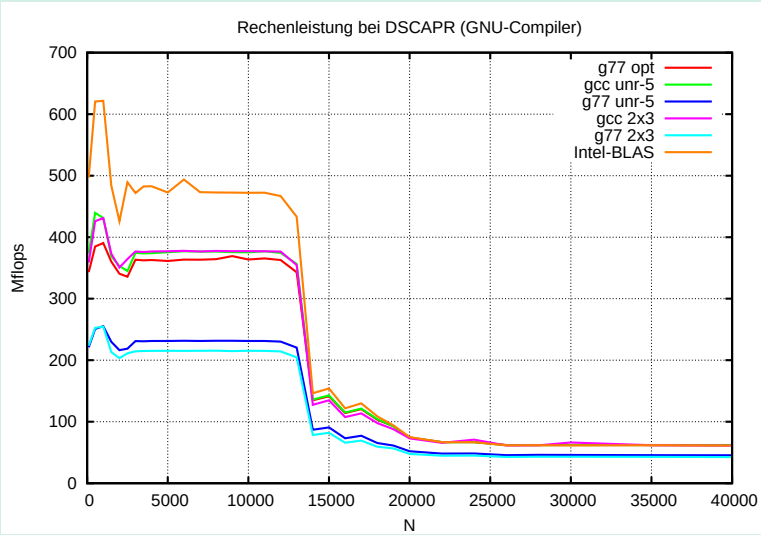
Rechenleistung auf einem Prozessorknoten

(1) Skalarproduktberechnung

- Berechne (k_N-mal) $s = \sum_{i=1}^{N} x_i y_i$
- für verschiedene Vektorlängen $N = 100, \ldots, 100\,000, \ldots,$ und $k_N \cdot N \approx const$.
- verschiedene Programmversionen (einfach, unrolled loops; C, Fortran)
- Bestimme Mflops aus der Rechenzeit und zeige
- Abhängigkeit von Speicherzugriffen (für kleine N fast nur Cache)

Zum Vergleich:

P III-800 (GNU-Compiler)



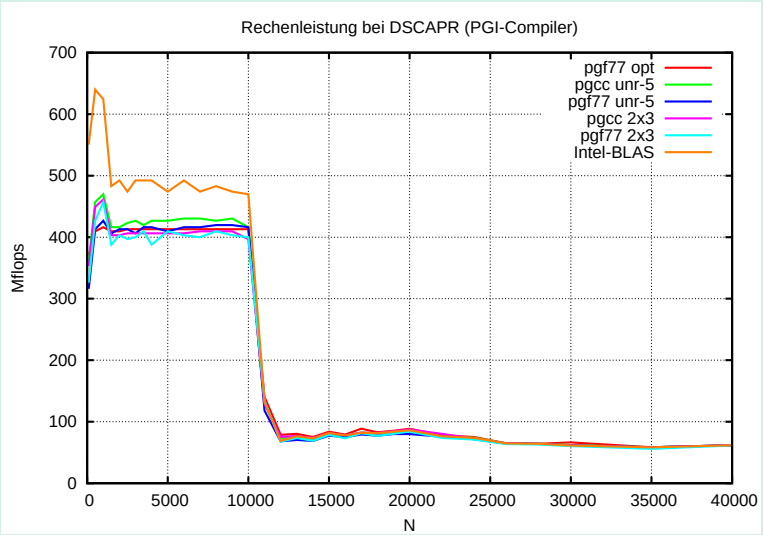
Rechenleistung auf einem Prozessorknoten

(1) Skalarproduktberechnung

- Berechne (k_N-mal) $s = \sum_{i=1}^{N} x_i y_i$
- für verschiedene Vektorlängen $N = 100, \ldots, 100\,000, \ldots,$ und $k_N \cdot N \approx const$.
- verschiedene Programmversionen (einfach, unrolled loops; C, Fortran)
- Bestimme Mflops aus der Rechenzeit und zeige
- Abhängigkeit von Speicherzugriffen (für kleine N fast nur Cache)

Zum Vergleich:

P III-800 (PGI-Compiler-Suite)



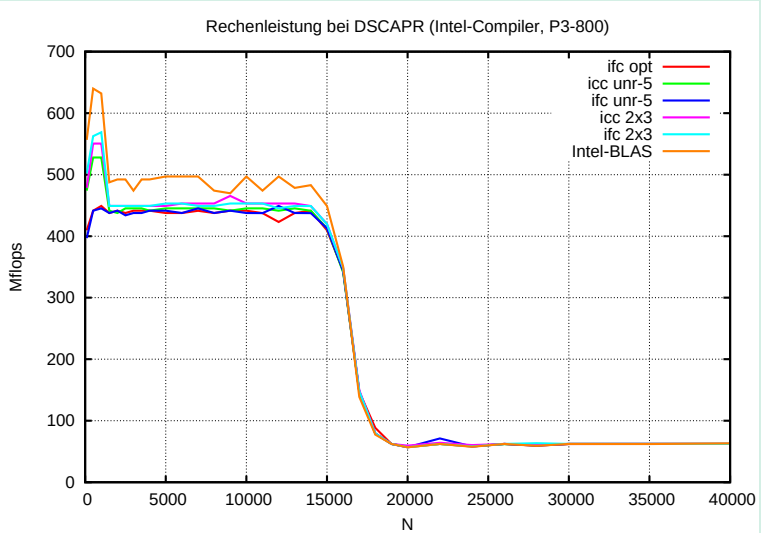
Rechenleistung auf einem Prozessorknoten

(1) Skalarproduktberechnung

- Berechne (k_N-mal) $s = \sum_{i=1}^{N} x_i y_i$
- für verschiedene Vektorlängen $N = 100, \ldots, 100\,000, \ldots,$ und $k_N \cdot N \approx const$.
- verschiedene Programmversionen (einfach, unrolled loops; C, Fortran)
- Bestimme Mflops aus der Rechenzeit und zeige
- Abhängigkeit von Speicherzugriffen (für kleine N fast nur Cache)

Zum Vergleich:

P III-800 (Intel-Compiler-Suite)



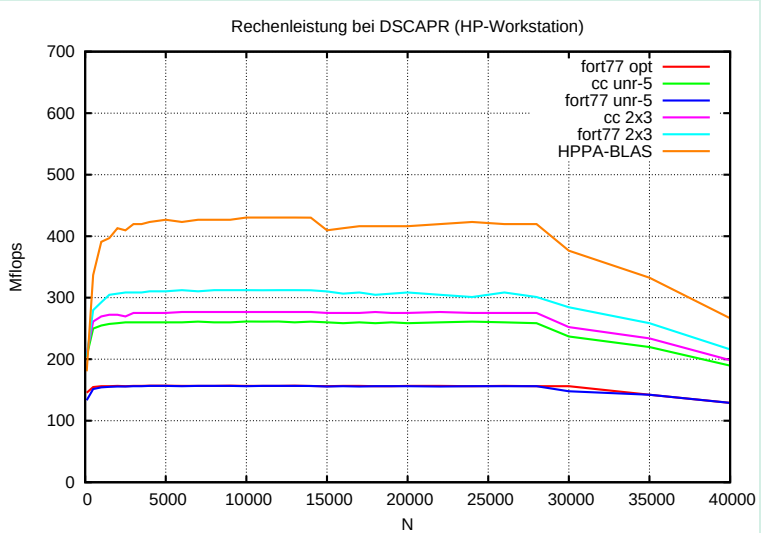
Rechenleistung auf einem Prozessorknoten

(1) Skalarproduktberechnung

- Berechne (k_N-mal) $s = \sum_{i=1}^{N} x_i y_i$
- für verschiedene Vektorlängen $N = 100, \ldots, 100\,000, \ldots,$ und $k_N \cdot N \approx const$.
- verschiedene Programmversionen (einfach, unrolled loops; C, Fortran)
- Bestimme Mflops aus der Rechenzeit und zeige
- Abhängigkeit von Speicherzugriffen (für kleine N fast nur Cache)

Zum Vergleich:

HP Workstation



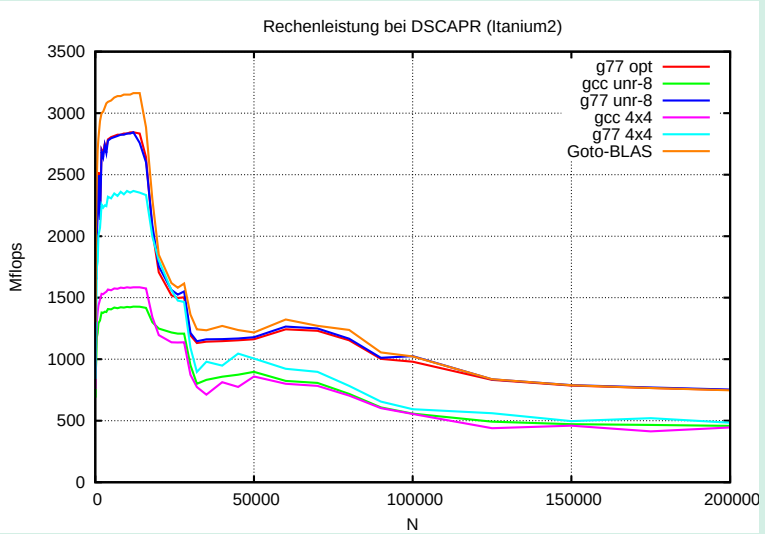
Rechenleistung auf einem Prozessorknoten

(1) Skalarproduktberechnung

- Berechne (k_N-mal) $s = \sum_{i=1}^{N} x_i y_i$
- für verschiedene Vektorlängen $N = 100, \ldots, 100\,000, \ldots,$ und $k_N \cdot N \approx const$.
- verschiedene Programmversionen (einfach, unrolled loops; C, Fortran)
- Bestimme Mflops aus der Rechenzeit und zeige
- Abhängigkeit von Speicherzugriffen (für kleine N fast nur Cache)

Zum Vergleich:

Itanium-2 (GNU und Goto-BLAS)



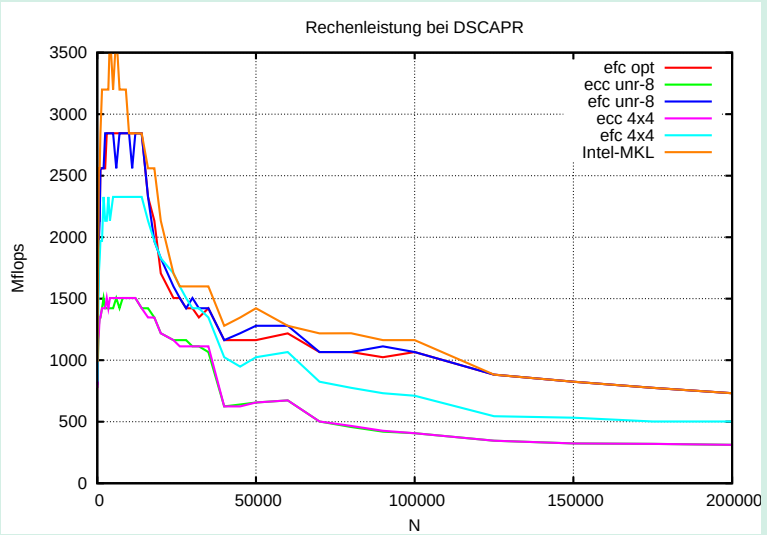
Rechenleistung auf einem Prozessorknoten

(1) Skalarproduktberechnung

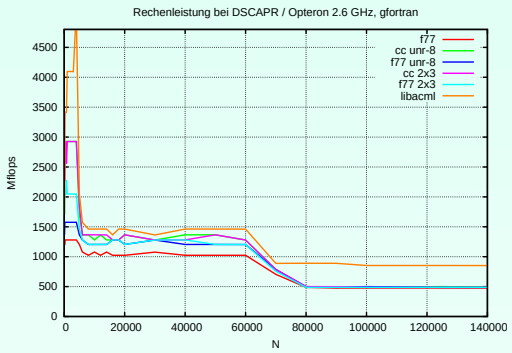
- Berechne (k_N-mal) $s = \sum_{i=1}^{N} x_i y_i$
- für verschiedene Vektorlängen $N = 100, \ldots, 100\,000, \ldots,$ und $k_N \cdot N \approx const$.
- verschiedene Programmversionen (einfach, unrolled loops; C, Fortran)
- Bestimme Mflops aus der Rechenzeit und zeige
- Abhängigkeit von Speicherzugriffen (für kleine N fast nur Cache)

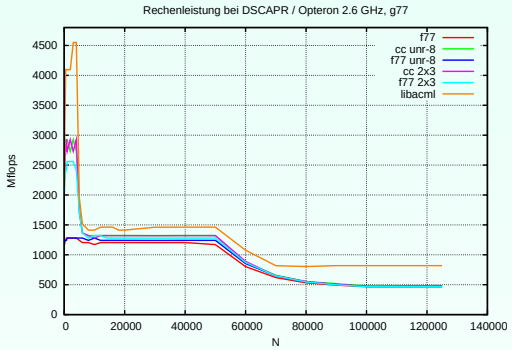
Zum Vergleich:

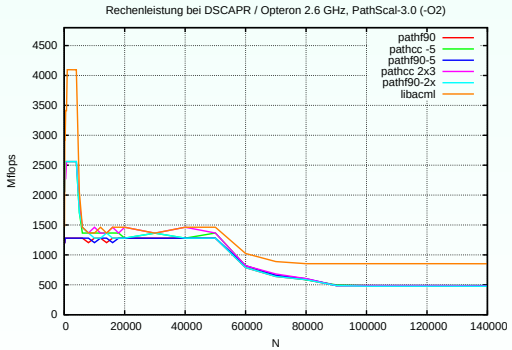
Itanium-2 (Intel-Comp. und MKL)



Einzelknoten auf CHiC

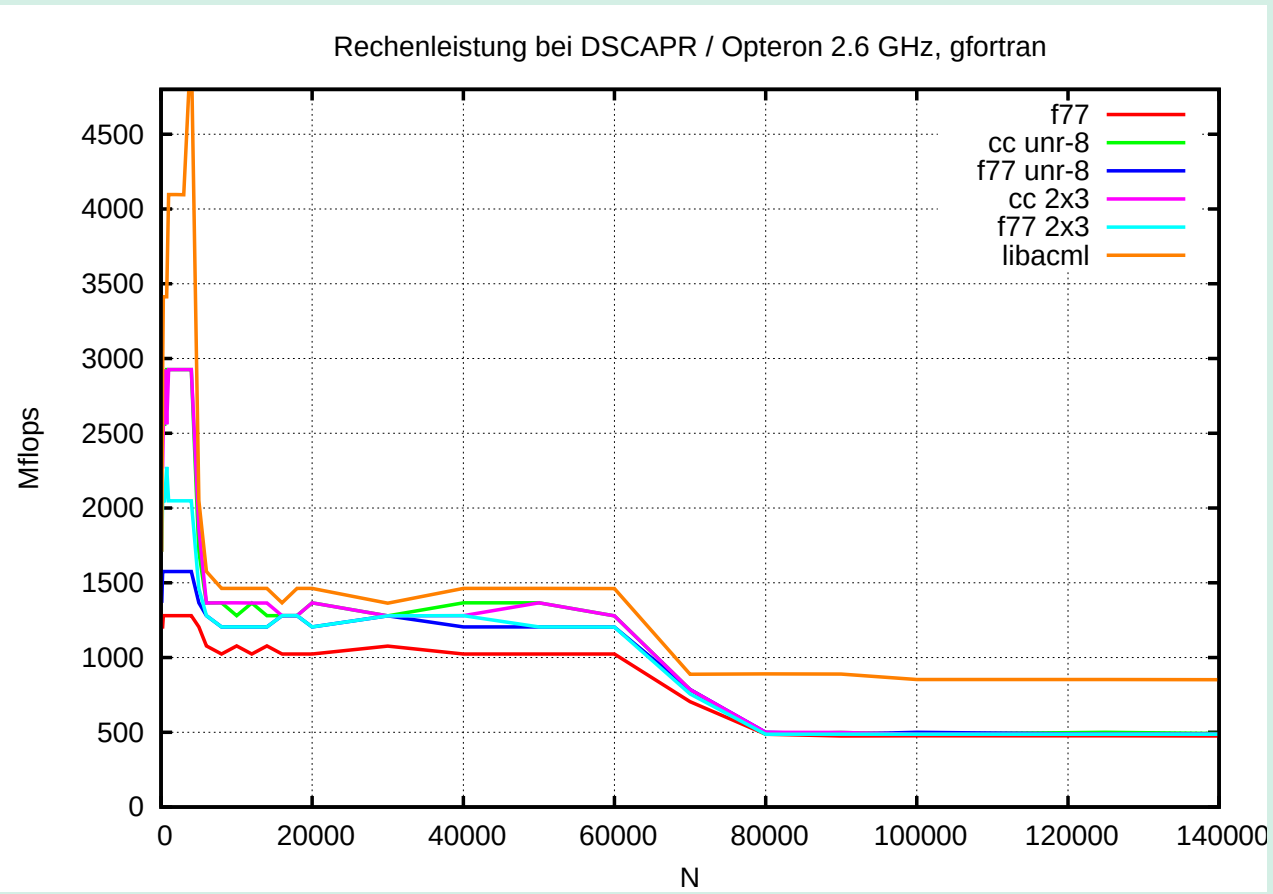




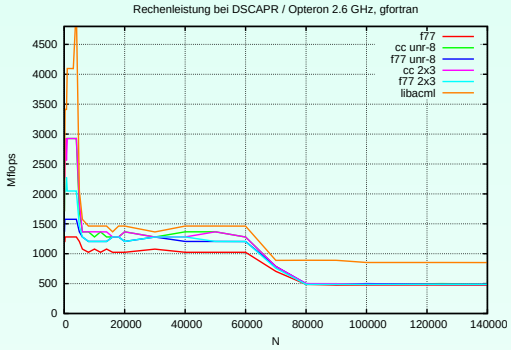


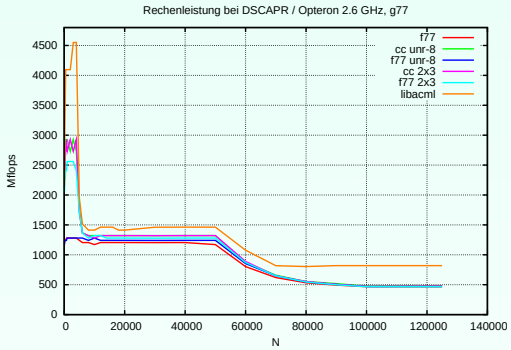
BLAS-Bibliothek: ACML = AMD Core Math Library

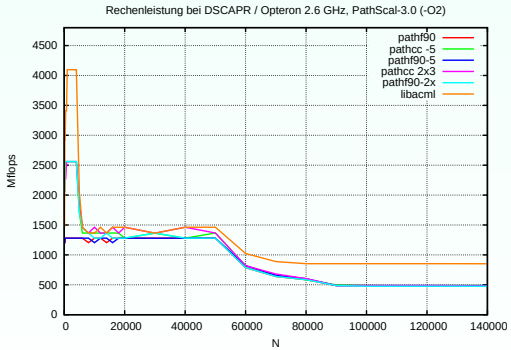
CHiC – gfortran/gcc



Einzelknoten auf CHiC

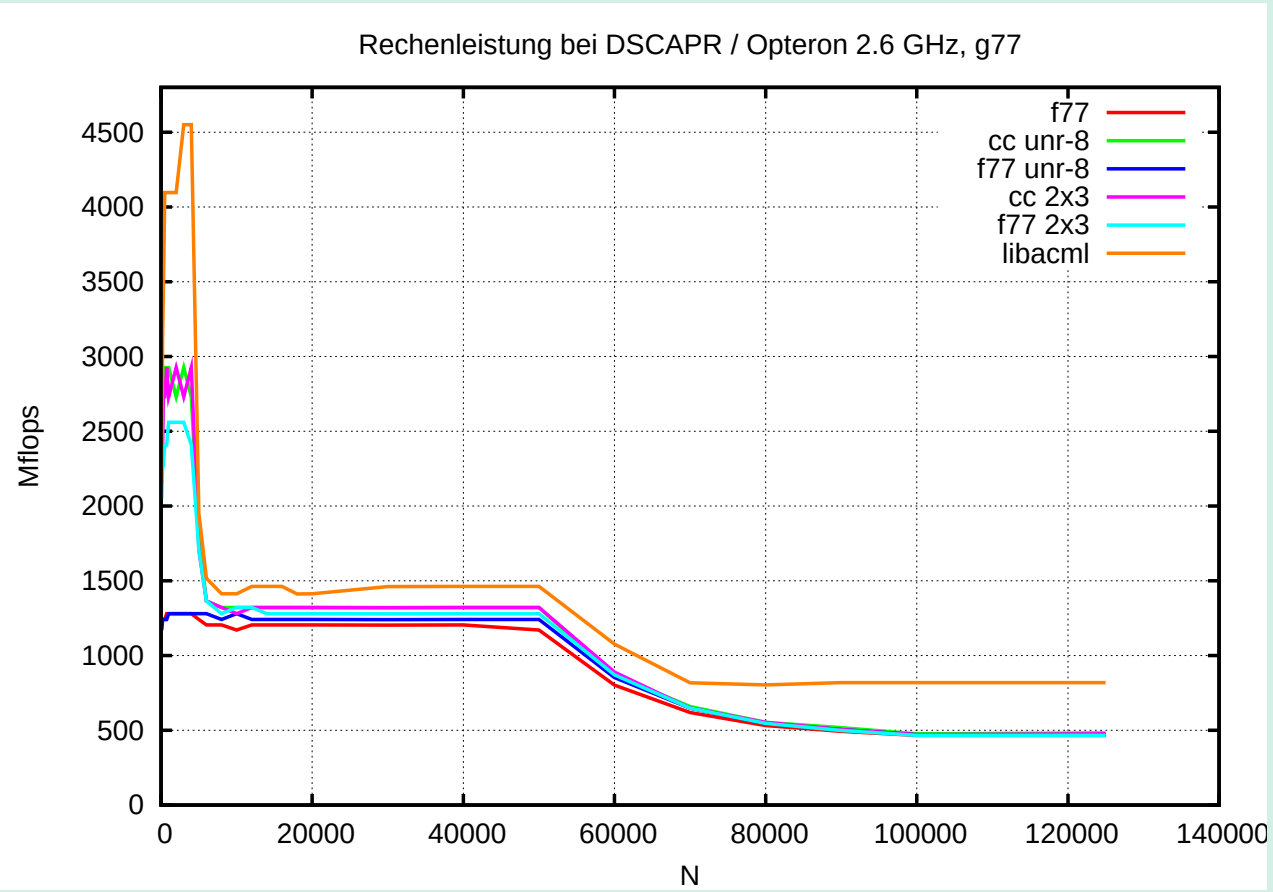




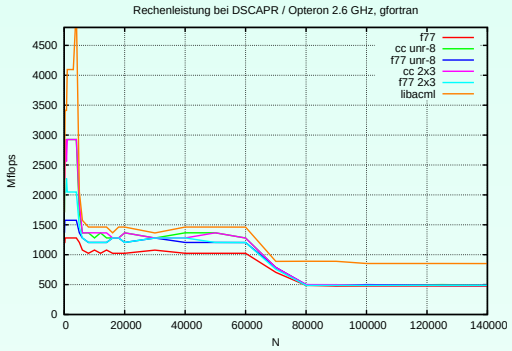


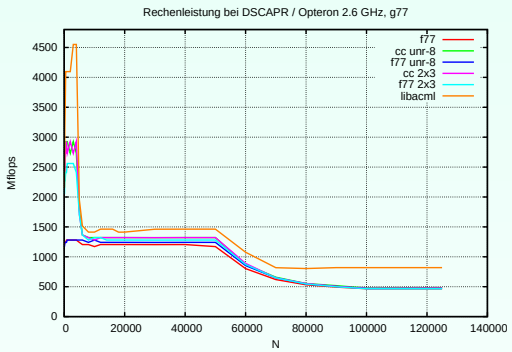
BLAS-Bibliothek: ACML = AMD Core Math Library

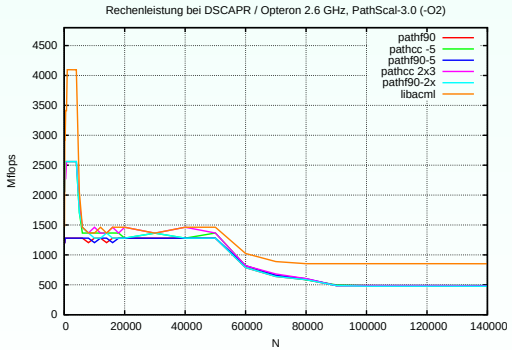
CHiC – g77/gcc



Einzelknoten auf CHiC

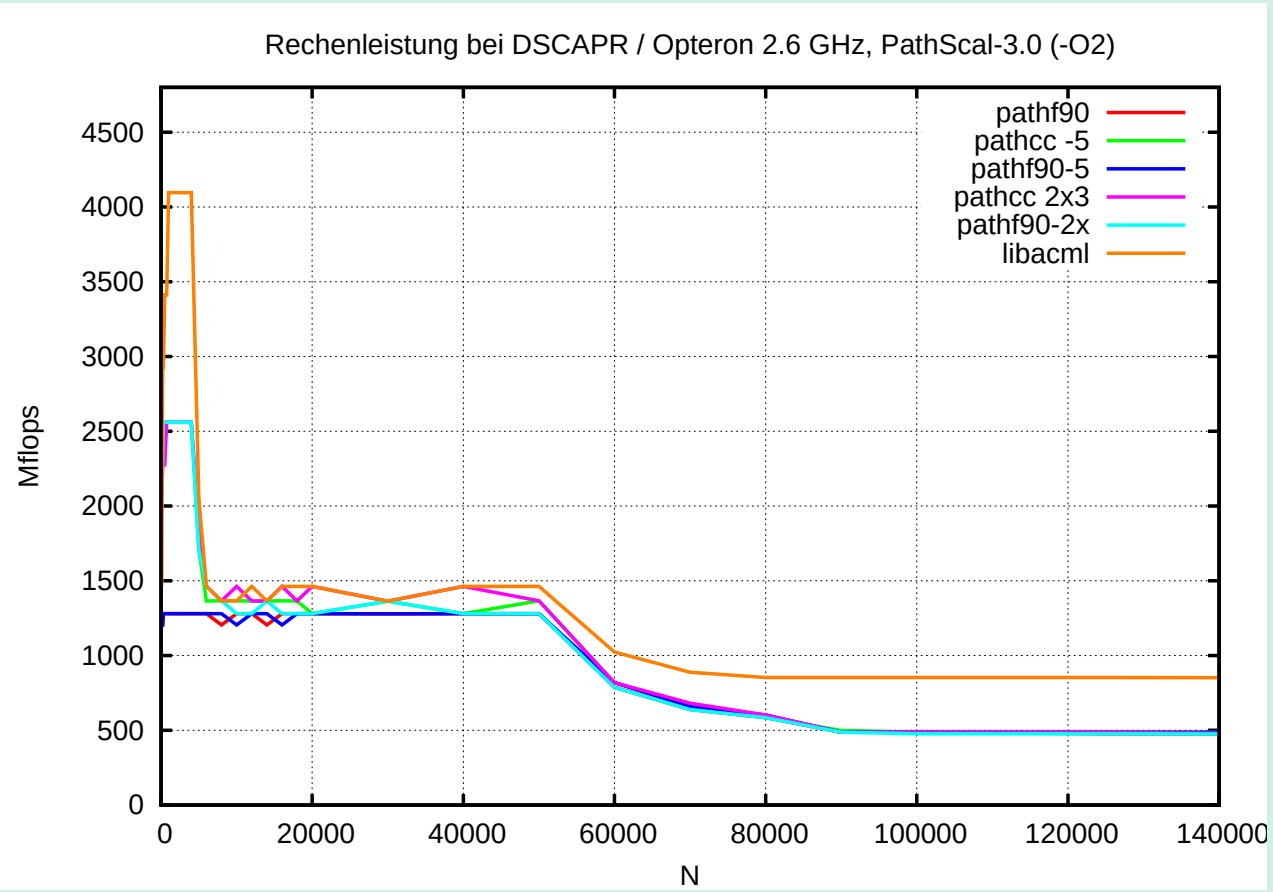




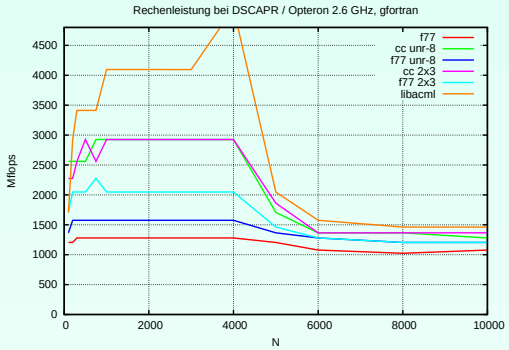


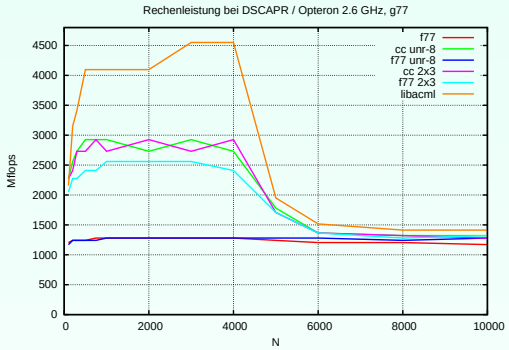
BLAS-Bibliothek: ACML = AMD Core Math Library

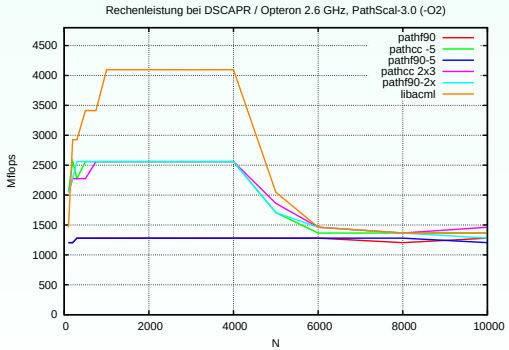
CHiC – PathScale-3.0 (-O2)



Einzelknoten auf CHiC

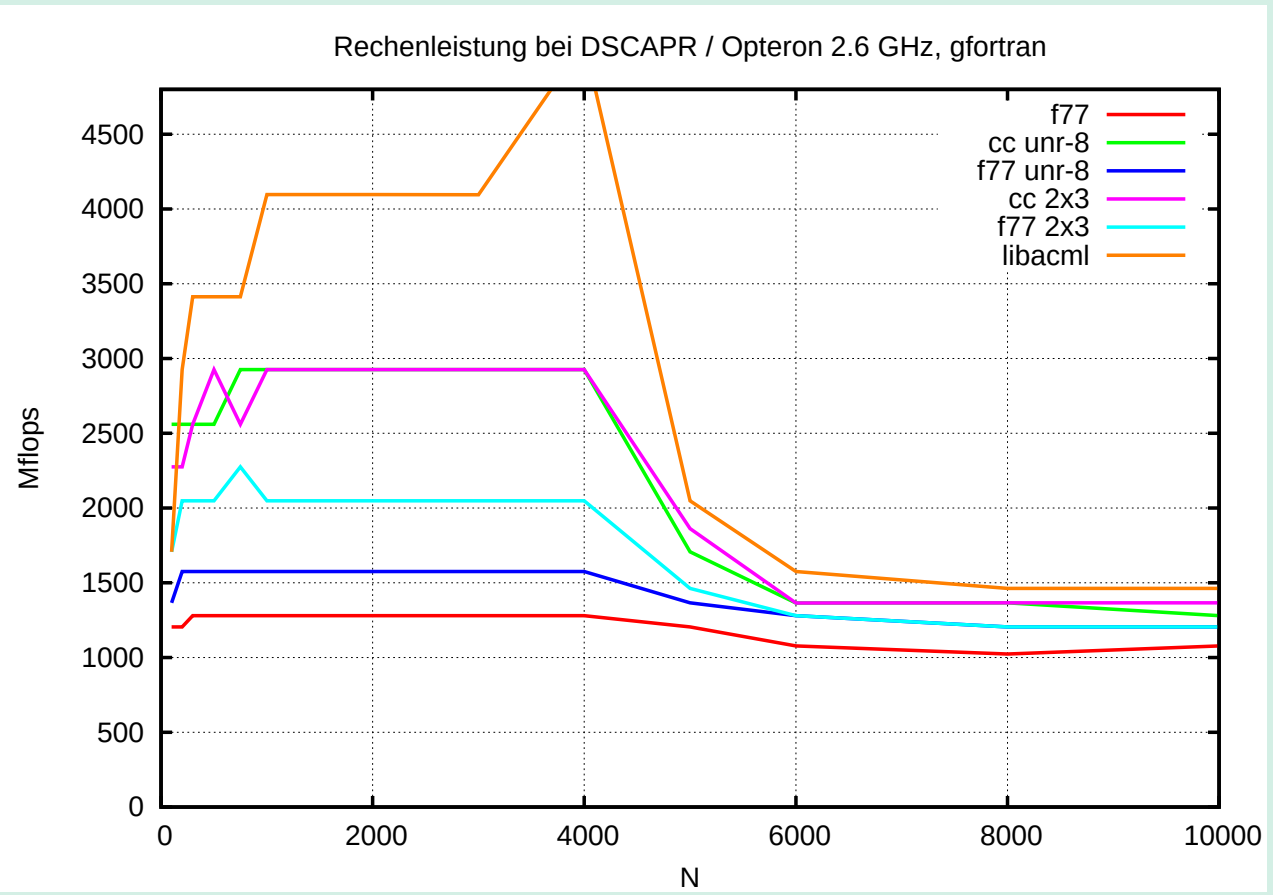




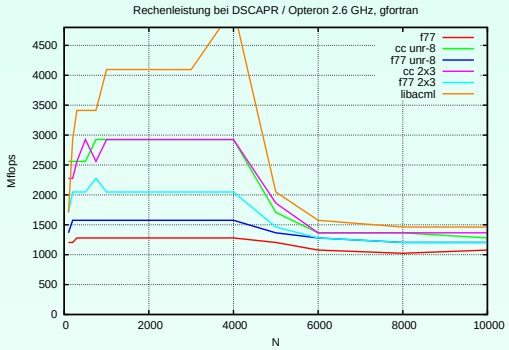


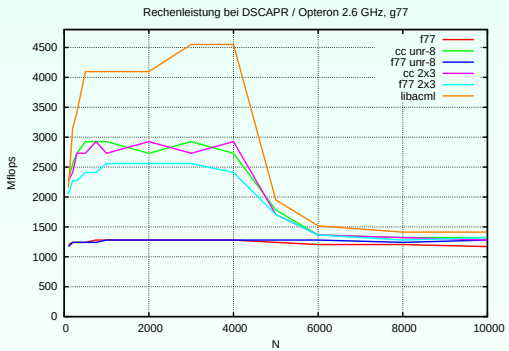
BLAS-Bibliothek: ACML = AMD Core Math Library

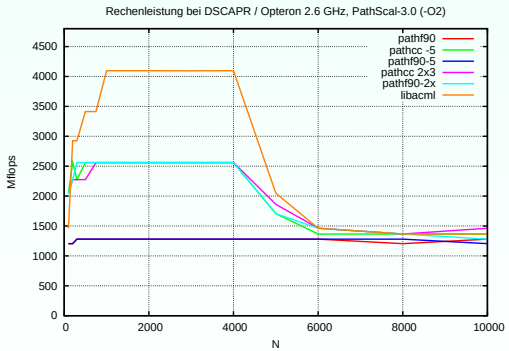
CHiC – gfortran/gcc



Einzelknoten auf CHiC

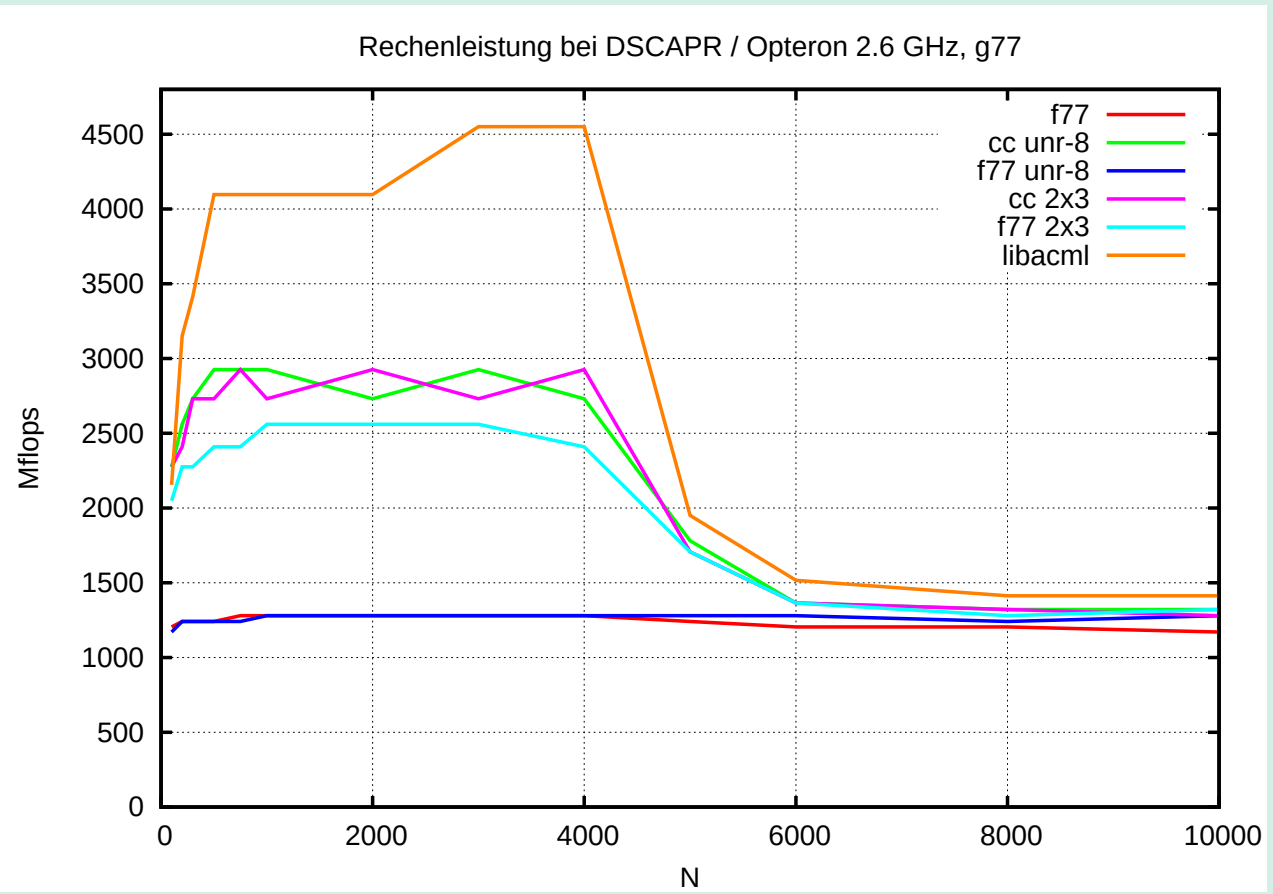




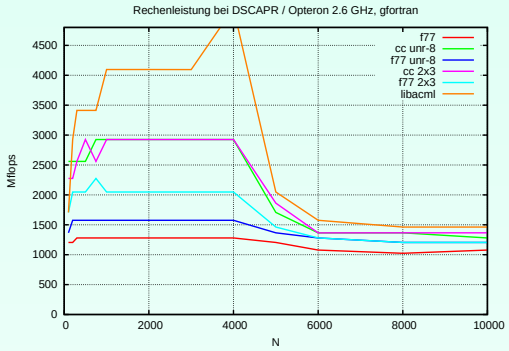


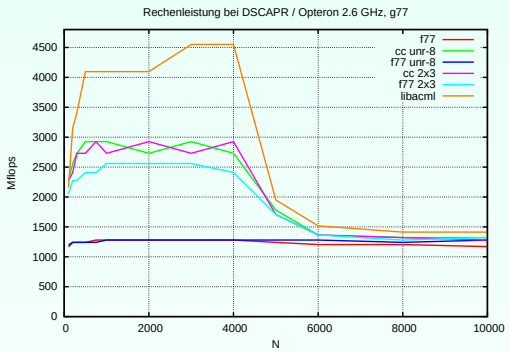
BLAS-Bibliothek: ACML = AMD Core Math Library

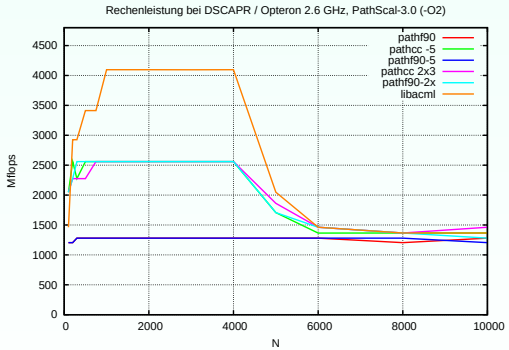
CHiC – g77/gcc



Einzelknoten auf CHiC

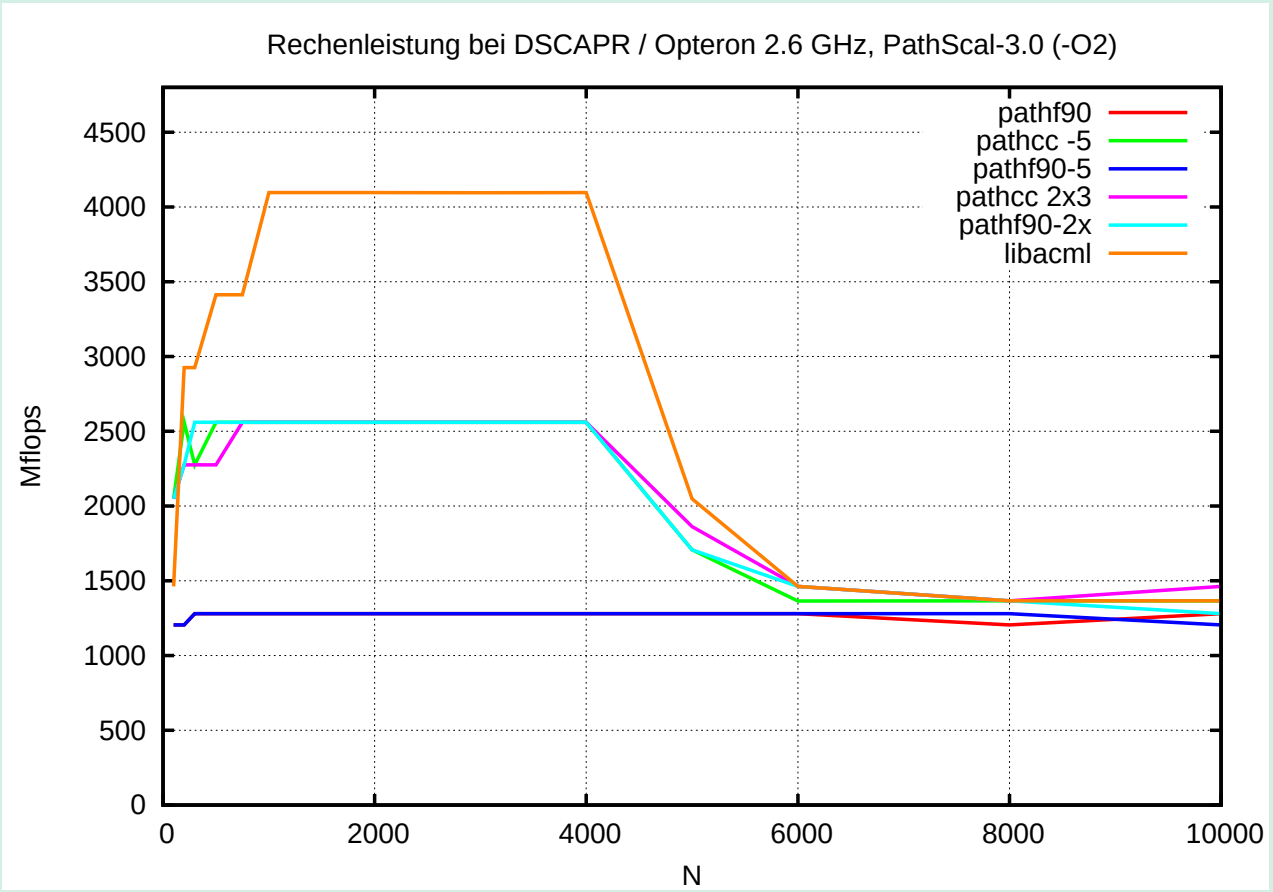




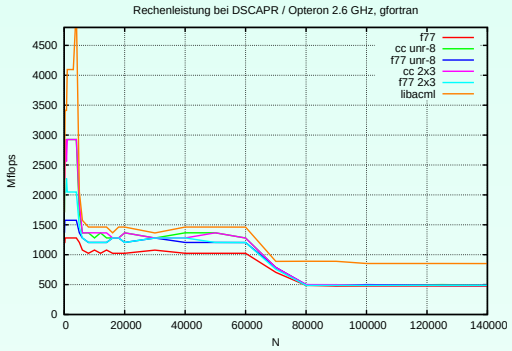


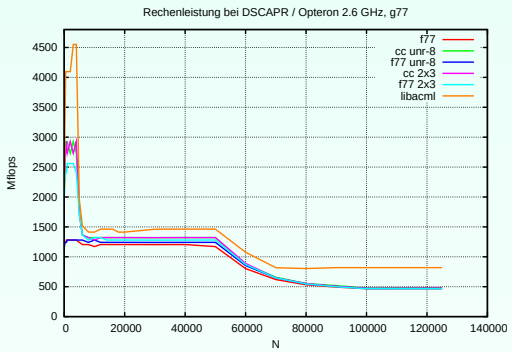
BLAS-Bibliothek: ACML = AMD Core Math Library

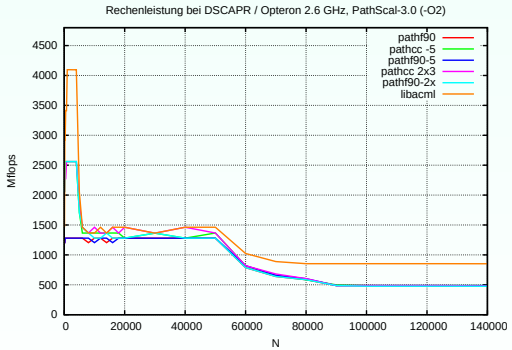
CHiC – PathScale-3.0 (-O2)



Einzelknoten auf CHiC

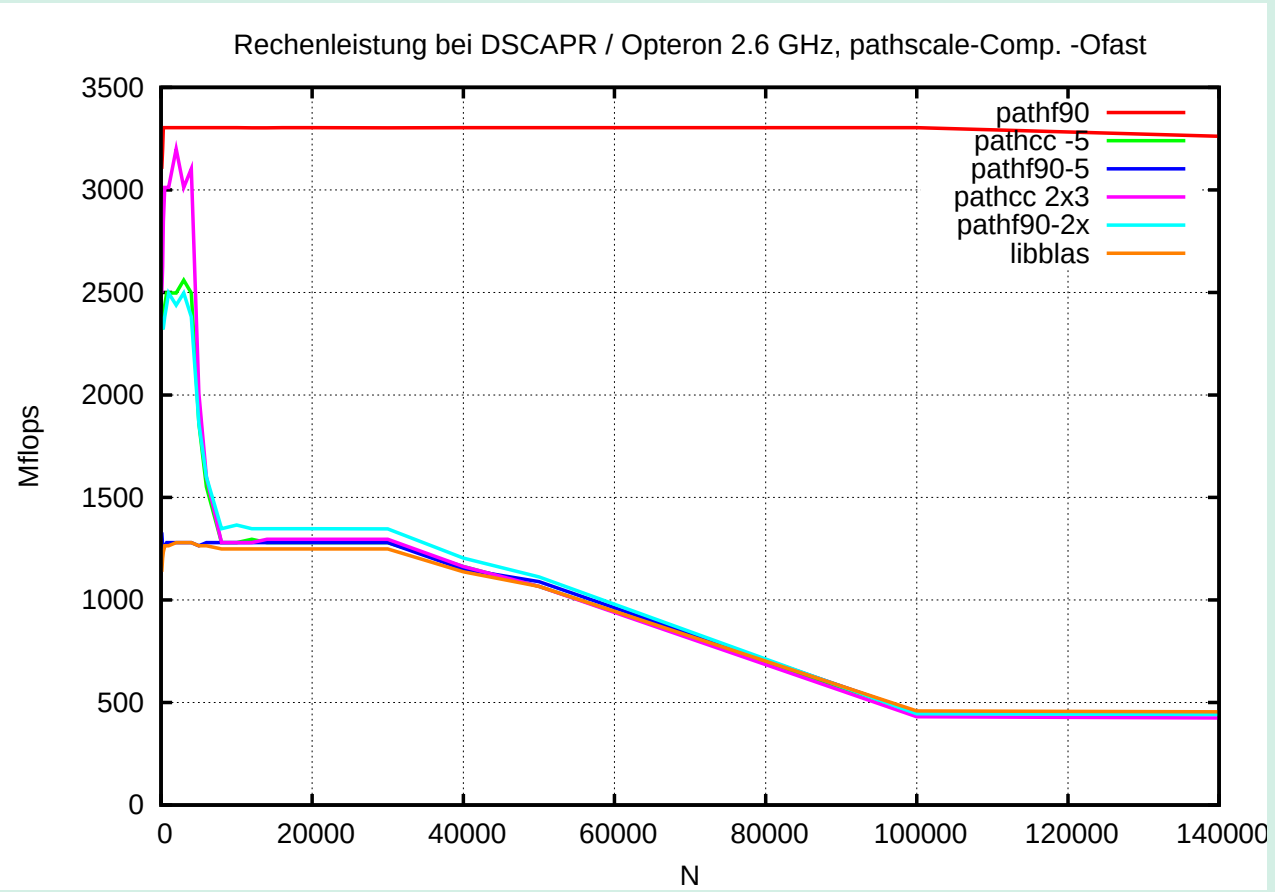




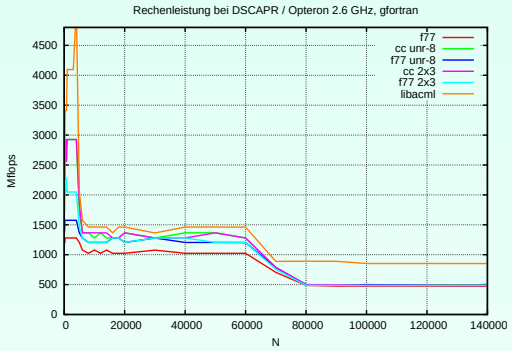


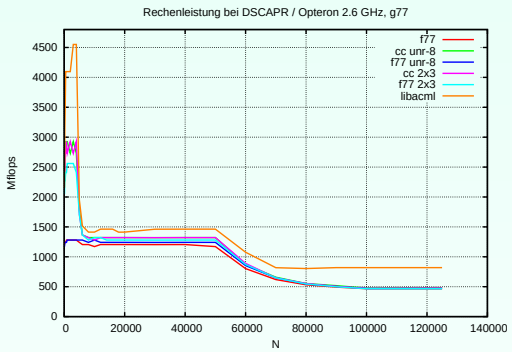
„surprising features“

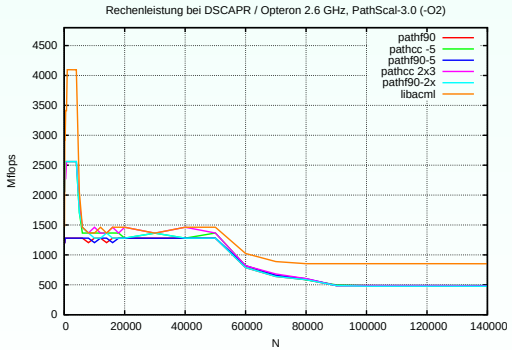
CHiC – PathScale-2.4 (-Ofast)



Einzelknoten auf CHiC

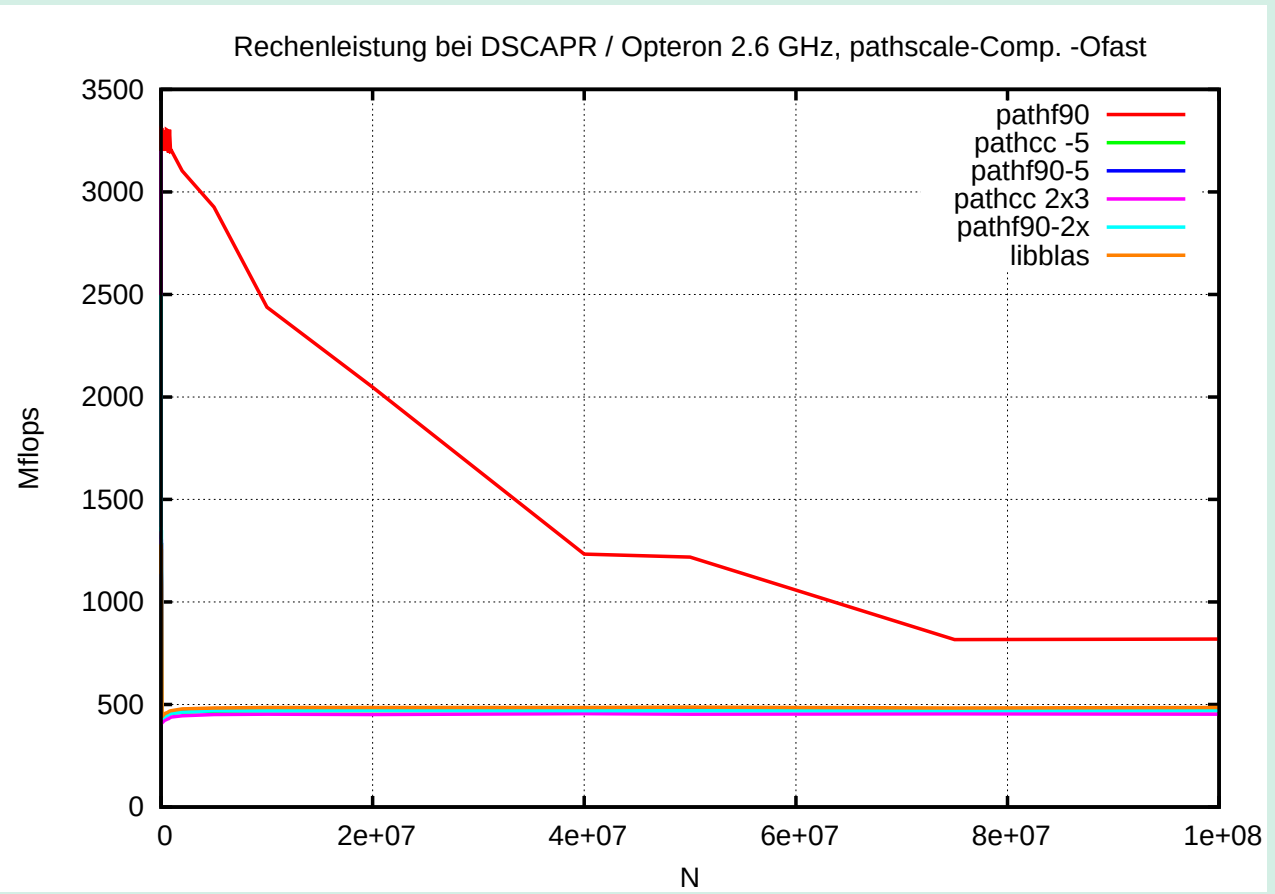




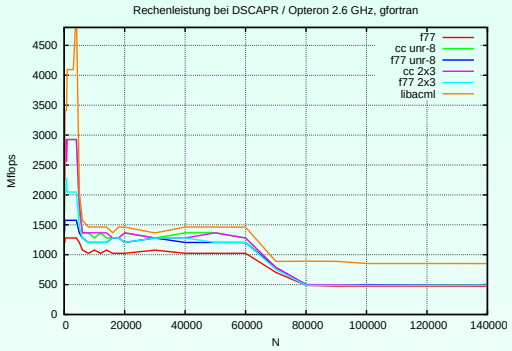


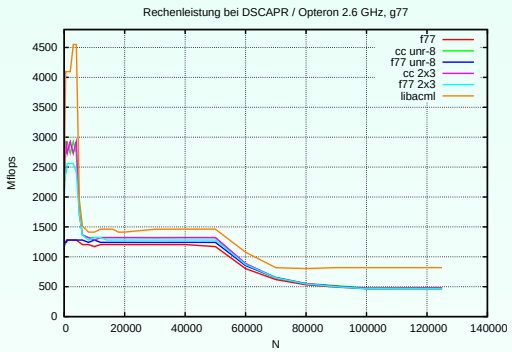
„surprising features“

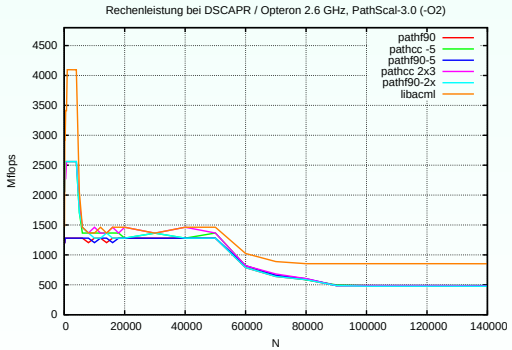
CHiC – PathScale-2.4 (-Ofast)



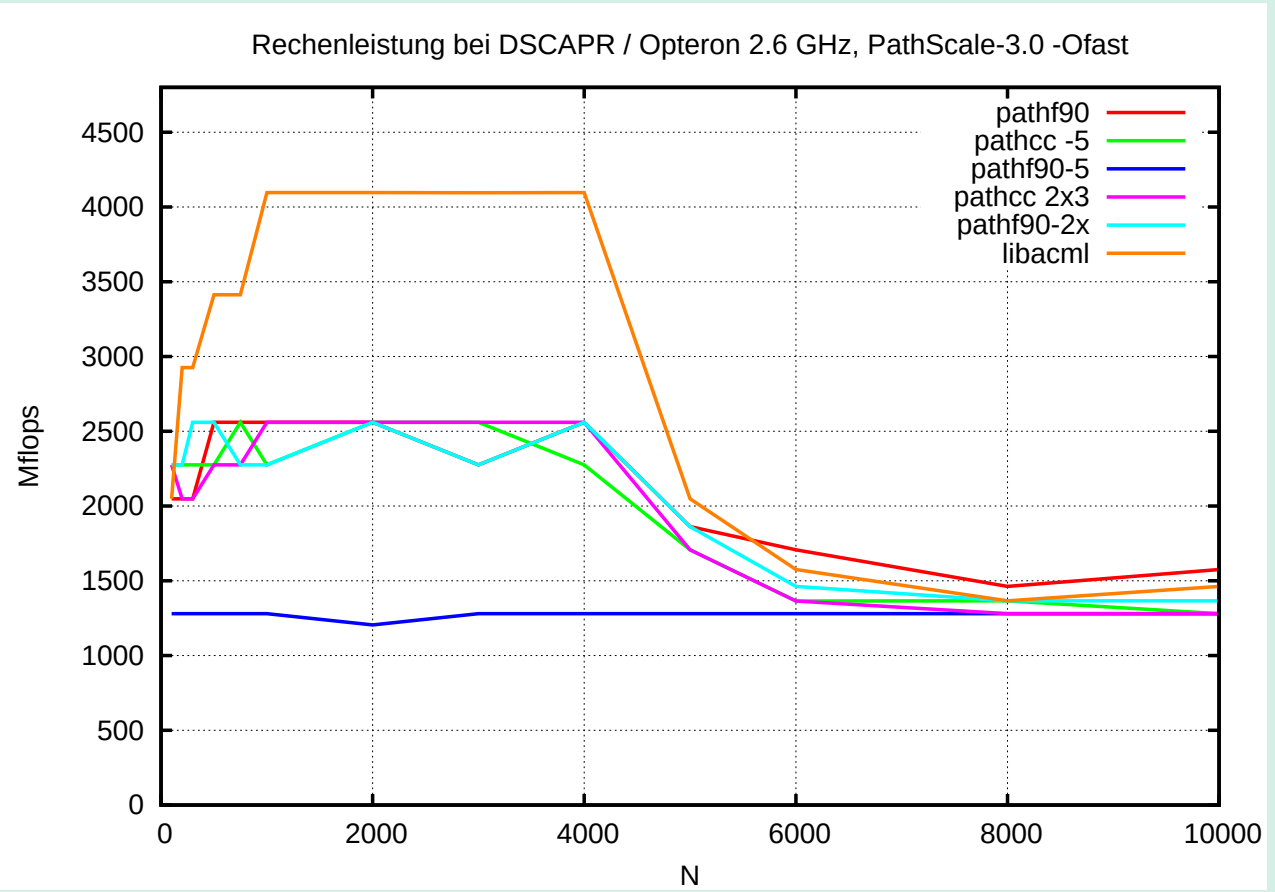
Einzelknoten auf CHiC



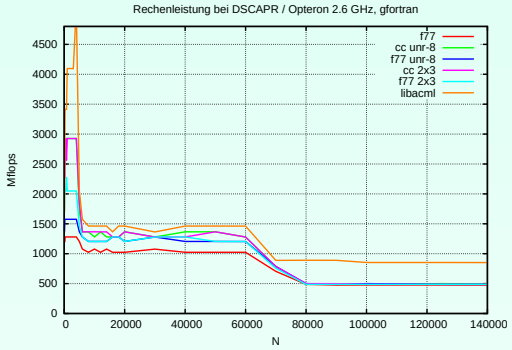


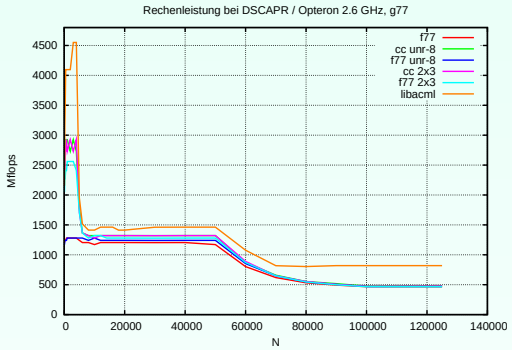


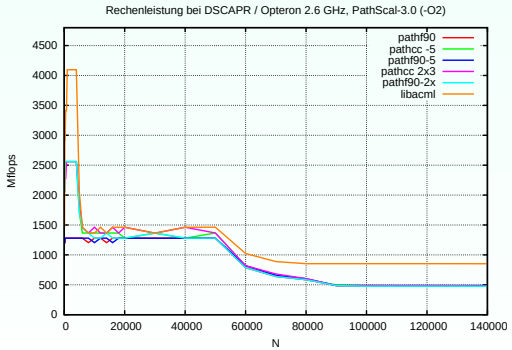
CHiC – PathScale-3.0 (-Ofast)



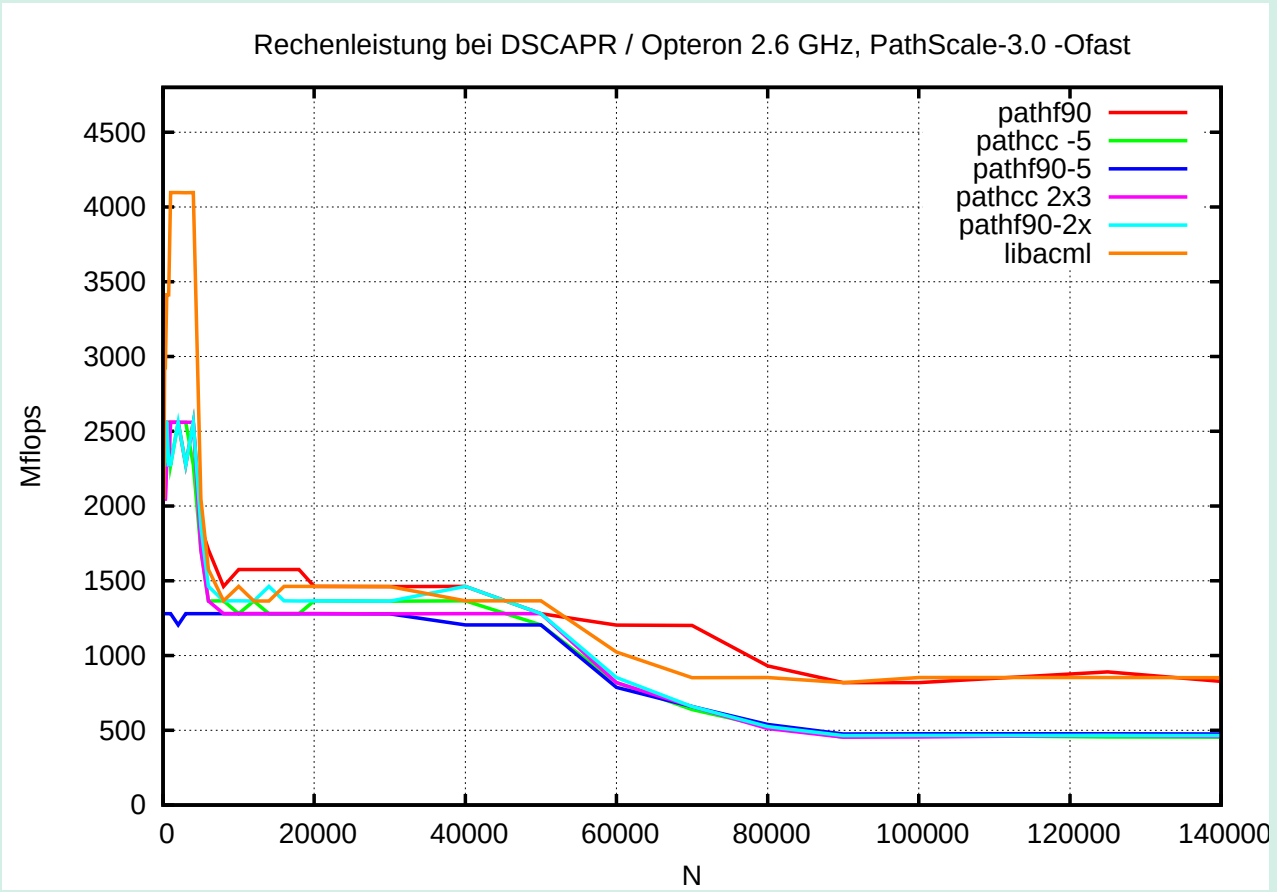
Einzelknoten auf CHiC



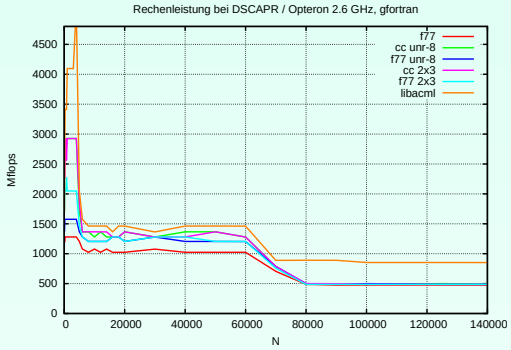


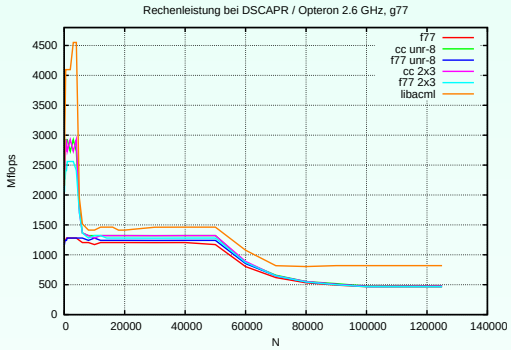


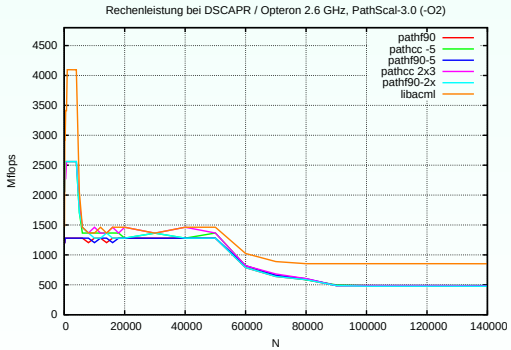
CHiC – PathScale-3.0 (-Ofast)



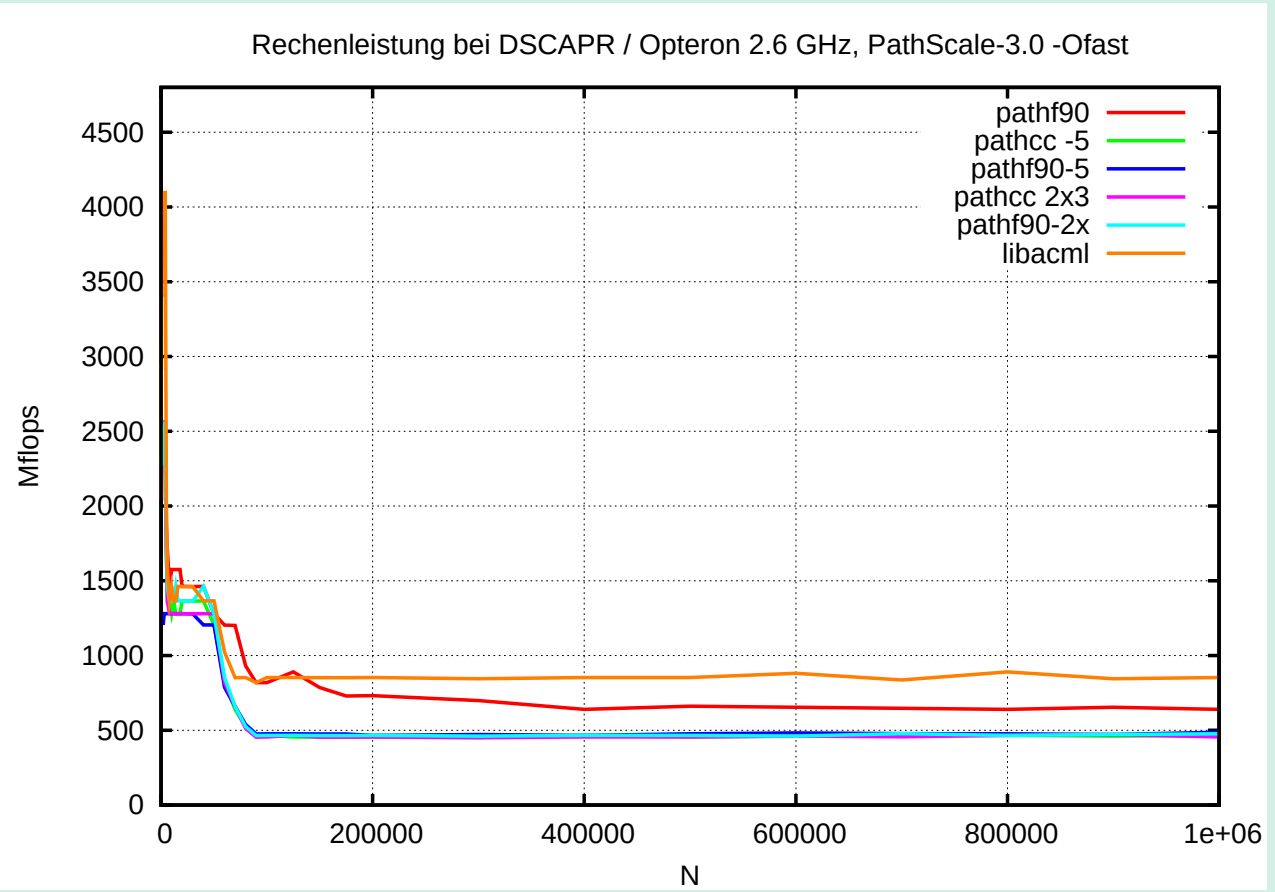
Einzelknoten auf CHiC







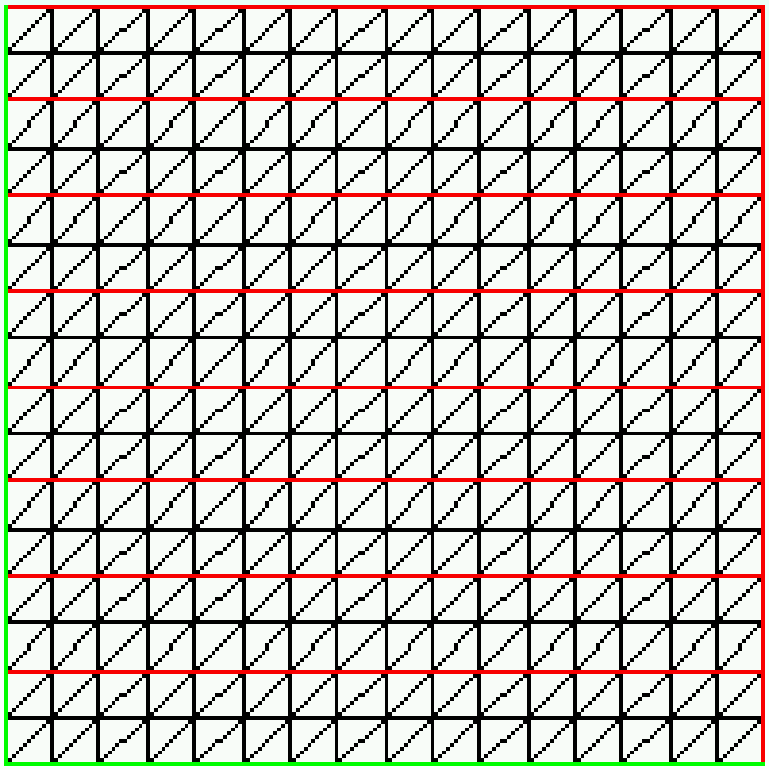
CHiC – PathScale-3.0 (-Ofast)

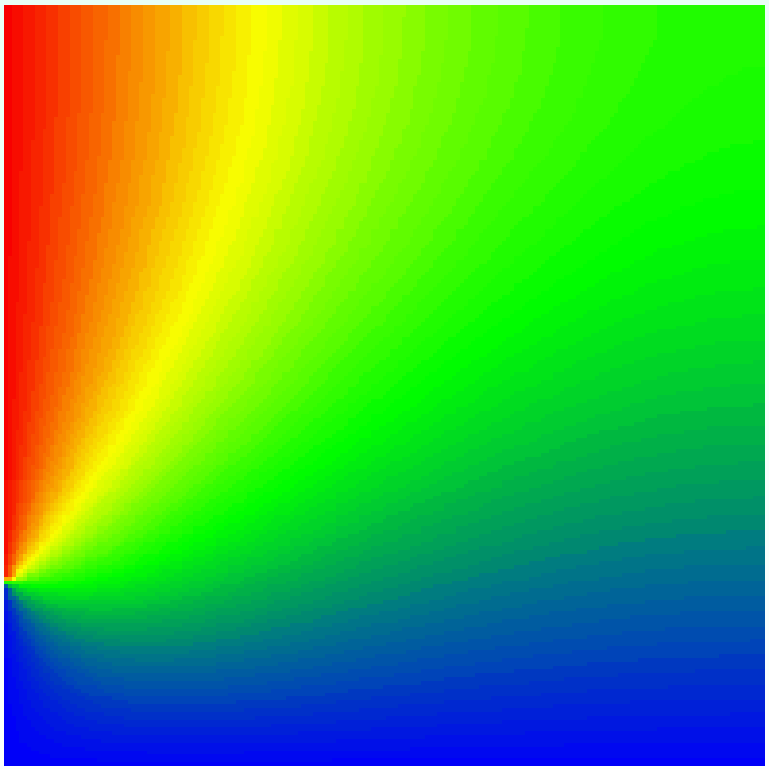


Rechenleistung auf einem Prozessorknoten

(2) FEM-2D-Referenzbeispiel

Das aus 128 Dreieckselementen bestehende Quadrat wurde bisher auf sehr vielen Architekturen als Referenzbeispiel verwendet.



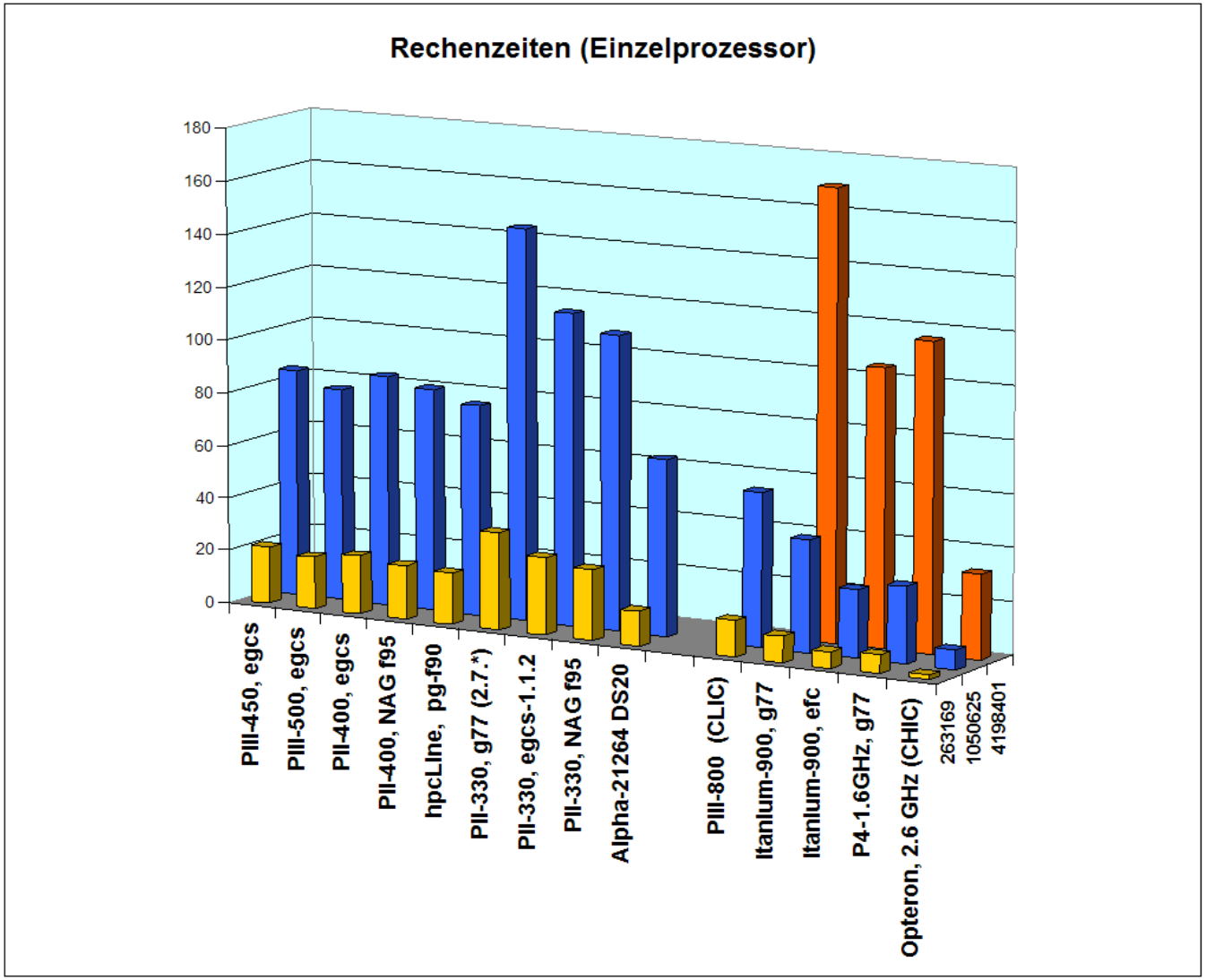


(2) FEM-2D-Referenzbeispiel (1 Prozessor)

Eine Auswahl getesteter Einzelprozessoren (überwiegend vor der Beschaffung von CLiC), jeweils 5-, 6-, 7-mal verfeinertes Netz.

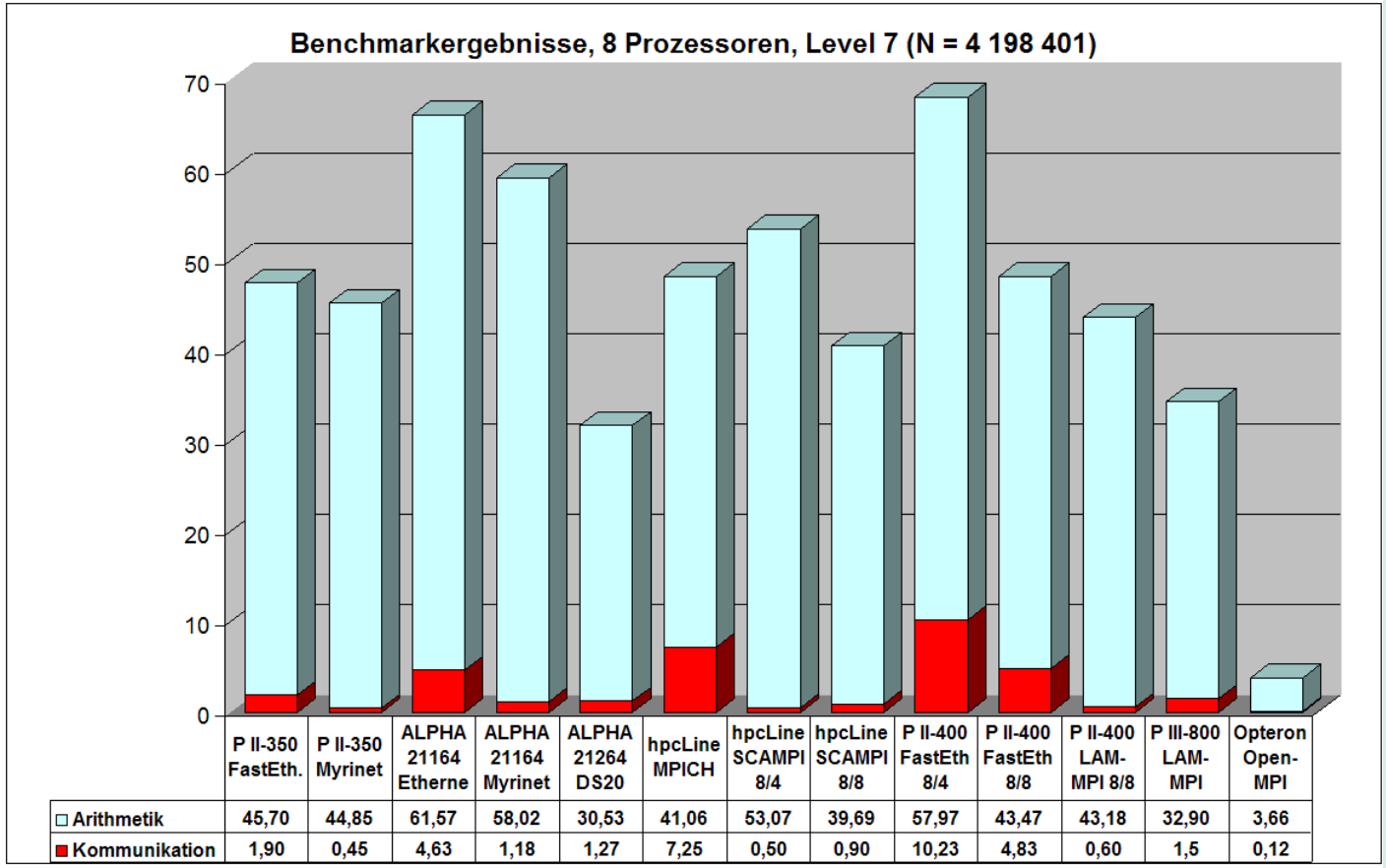
Level	5	6	7
Unbekannte	263 169	1 050 625	4 198 401
Iterationen	44	45	45
	Rechenzeiten [s]		
hpcLine	19,3	79,3	—
Alpha 21264 DS20	13,0	66,2	—
PIII-800 (CLiC)	13,7	57,8	—
Itanium-900	6,1	25,5	104,4
P4 - 1.6 GHz	7,1	28,7	116,1
Opteron-2.6 GHz	1,7	7,3	31,8

(2) FEM-2D-Referenzbeispiel (1 Prozessor)



Parallel auf 8-Prozessor-Cluster

Beispiel mit 4 198 401 Unbekannten (7-mal verfeinertes Netz).
Vor der Beschaffung von CLiC getestete Cluster, vgl. mit CHiC

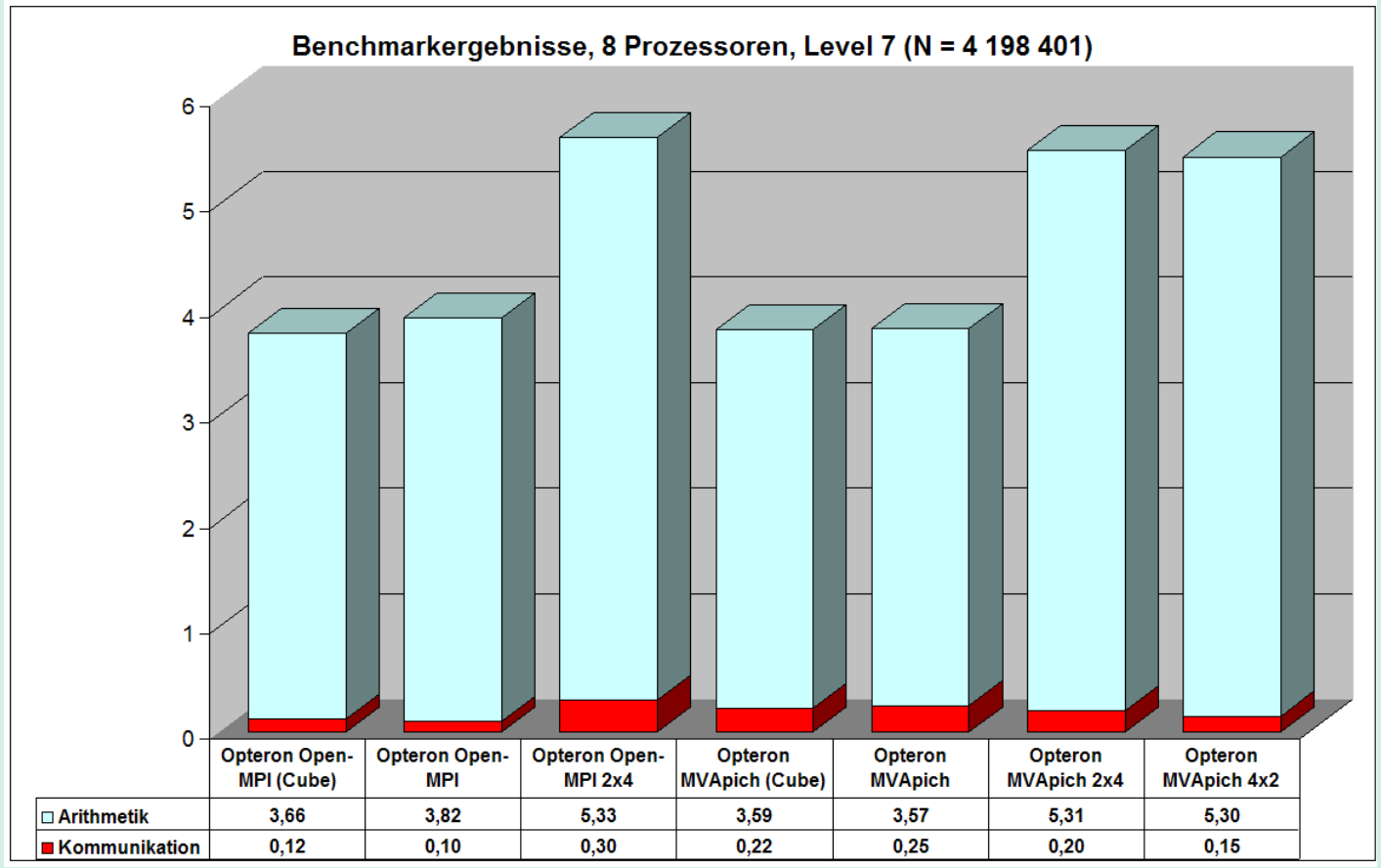


	P II-350 FastEth.	P II-350 Myrinet	ALPHA 21164 Etherne	ALPHA 21164 Myrinet	ALPHA 21264 DS20	hpcLine MPICH	hpcLine SCAMPI 8/4	hpcLine SCAMPI 8/8	P II-400 FastEth 8/4	P II-400 FastEth 8/8	P II-400 LAM-MPI 8/8	P III-800 LAM-MPI	Opteron Open-MPI
Arithmetik	45,70	44,85	61,57	58,02	30,53	41,06	53,07	39,69	57,97	43,47	43,18	32,90	3,66
Kommunikation	1,90	0,45	4,63	1,18	1,27	7,25	0,50	0,90	10,23	4,83	0,60	1,5	0,12

Parallel auf 8-Prozessor-Cluster

Beispiel mit 4 198 401 Unbekannten (7-mal verfeinertes Netz).

Verschiedene Testumgebung auf CHiC



	Opteron Open-MPI (Cube)	Opteron Open-MPI	Opteron Open-MPI 2x4	Opteron MVApich (Cube)	Opteron MVApich	Opteron MVApich 2x4	Opteron MVApich 4x2
Arithmetik	3,66	3,82	5,33	3,59	3,57	5,31	5,30
Kommunikation	0,12	0,10	0,30	0,22	0,25	0,20	0,15

Parallel auf 64 bzw. 128 Knoten

Lev.	Unbekannte	Iter.[1]	64 Proz. (64 × 1) Ass.	PCG	IO	64 Proz. (16 × 4) Ass.	PCG	IO[2]
7	4 198 401	45	0,10	0,36	10%	0,11	0,49	10%
8	16 785 409	45	0,42	1,72	4%	0,44	2,68	5%
9	67 125 249	44	1,67	7,29	4%	1,75	11,34	5%
10	268 468 225	47	6,73	32,59	2%	7,00	49,84	5%

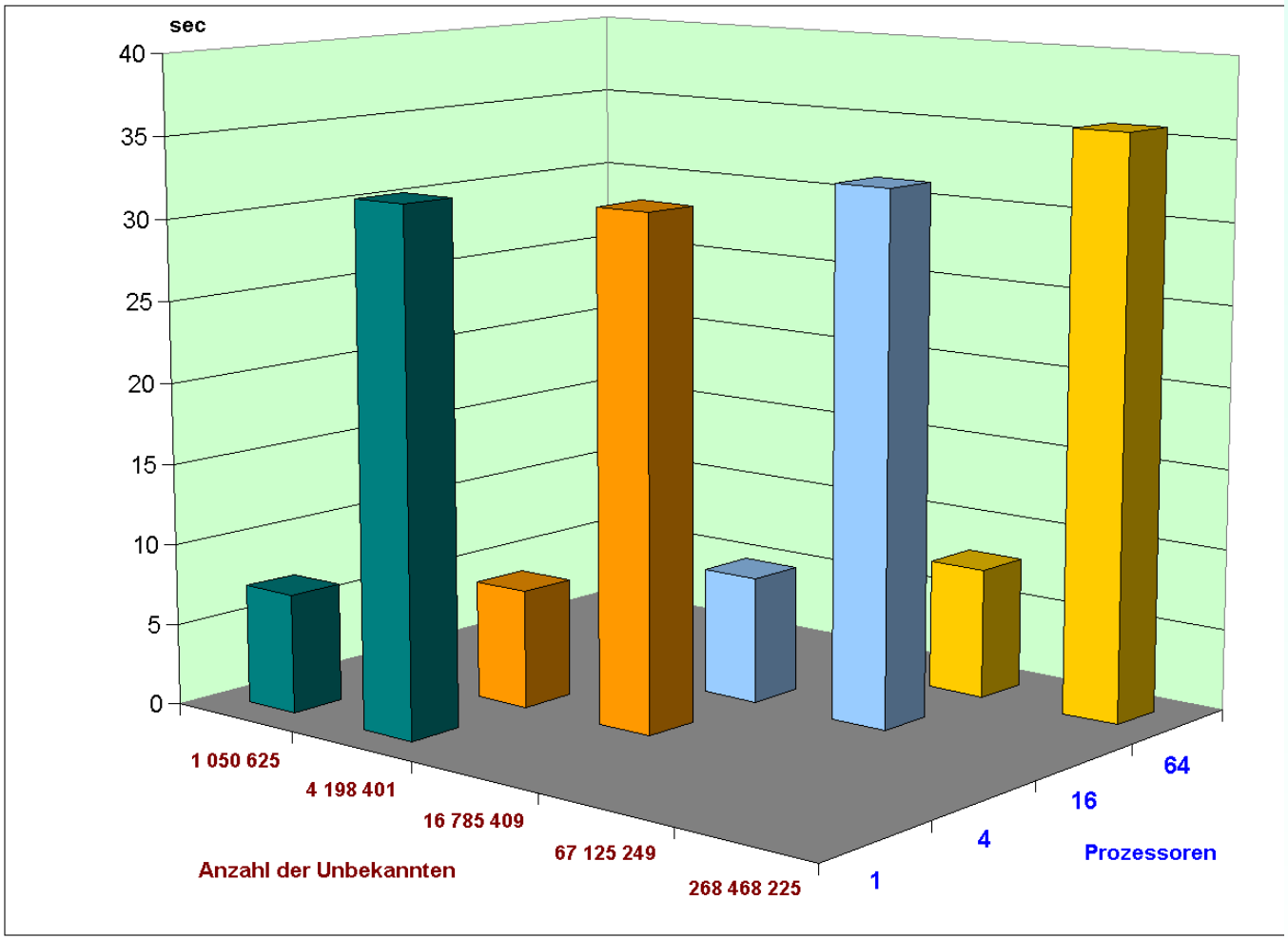
			128 Proz. (128 × 1)			128 Proz. (32 × 4)		
7	4 198 401	45	0,05	0,15	5%	0,06	0,2	30%
8	16 785 409	45	0,21	0,8	3%	0,22	1,3	6%
9	67 125 249	44	0,84	3,7	4%	0,89	5,6	5%
10	268 468 225	47	3,37	13,9	2%	3,52	23,1	5%
11	1 073 807 361	48	13,82	64,1	3%		—	

[1] PCG ohne Grobgitterlöser

[2] Anteil der Kommunikation als grobe Mittelung (schwankt zwischen Proz.)

Skalierung mit der Problemgröße

Pro Verfeinerungsschritt:
4-fache Zahl von Unbekannten mit 4-facher Prozessoranzahl



Globale Kommunikationsleistung

(1) Problembeschreibung:

- Welche Leistung bietet das Kommunikations-Netzwerk dem Nutzer in seiner Entwicklungsumgebung tatsächlich?
  (also ohne spezielles Tuning zwecks Hardware-Benchmark)
- Dazu wird folgende einfache Situation in Fortran realisiert (wie schon nach der Installation des CLiC):
  - Jeder Prozessor besitzt einen Vektor (`double`) der Länge N.
  - Über alle Prozessoren werden die Vektoren aufsummiert.
  - Es werden $p = 2^n$ Prozessoren verwendet.
- Zwei Implementationen:
  - `Cube_DoD` (MPIcubecom)
    Hypercube-Routine auf Basis von `MPI_sendrecv`
  - `MPI_Allreduce` (MPIcom)
    sollte die „beste“ Implementierung durch MPI sein

Globale Kommunikationsleistung

(2) Bemerkungen zur Auswertung:

- In der `Cube_DoD`-Variante kann man die von und zu jedem Prozessor gesendete Datenmenge bestimmen und aus der gemessenen Zeit eine Kommunikationsrate in Mbit/s je Prozessor (oder insgesamt) berechnen.
- Berechnungen für $p = 2^n$ Prozessoren, Vektorlänge N:
  Paketlänge: $L = \frac{8N}{(1024)^2}$ [MByte],
  Gesamtdatenfluss: $G = n \cdot p \cdot L$ bzw. je Prozessor $2 \cdot n \cdot L$,
  Gesamtdurchsatz: G/t [MByte/s],
  Kommunikationsrate pro Knoten: $8 \cdot (2 \cdot n \cdot L)/t$ [Mbit/s].
- In der `MPI_Allreduce`-Variante ist das weniger zuverlässig; Kommunikationswege sind durch MPI nicht vorgeschrieben, also nur ein *fiktives* Ergebnis, incl. CPU-Anteil.
- Für kleinere Paketlängen (≈ 100 KByte) sind gemessene Zeiten (anders als bei CLiC) auch mal gleich Null!

Globale Kommunikationsleistung

(3) Ergebnisse einiger Messungen: Mbit/s je Prozessor

lokale Vektor-länge N	Pakete L [MB]	Datenfluss G [GB]	Open-MPI (Cube)	Open-MPI (Reduce)	MVApich (Cube)	MVApich (Reduce)	CLiC
16 Prozessoren:							
2097152	16	1	9 309	1 862	10 240	10 240	141
8388608	64	4	9 525	1 837	11 703	10 180	142
32 Prozessoren:							
2097152	16	2,5	7 529	1 164	9 014	11 700	141
8388608	64	10	7 420	1 222	6 564	11 398	142
64 Prozessoren:							
2097152	16	6	8 533	753	5 485	3 938	141
8388608	64	24	7 062	752	5 535	3 990	141
128 Prozessoren:							
2097152	16	14	6 288	298	5 600	3 990	141
8388608	64	56	5 973	455	4 876	3 775	141

Gesamtzeiten jeweils $\approx 0.1 \ldots 2$ s (bei CLiC: 5. . . 50 s)

Globale Kommunikationsleistung

(3) Ergebnisse einiger Messungen: Mbit/s je Prozessor

lokale Vektor-länge N	Pakete L [MB]	Datenfluss G [GB]	Open-MPI (Cube)	Open-MPI (Reduce)	MVApich (Cube)	MVApich (Reduce)	CLiC
16 Prozessoren:							
2097152	16	1	9 309	1 862	10 240	10 240	141
8388608	64	4	9 525	1 837	11 703	10 180	142
32 Prozessoren:							
2097152	16	2,5	7 529	1 164	9 014	11 700	141
8388608	64	10	7 420	1 222	6 564	11 398	142
64 Prozessoren (32×2):							
2097152	16	6	4 800	725	3 339	2 560	141
8388608	64	24	4 726	746	3 531	2 560	141
128 Prozessoren:							
2097152	16	14	6 288	298	5 600	3 990	141
8388608	64	56	5 973	455	4 876	3 775	141

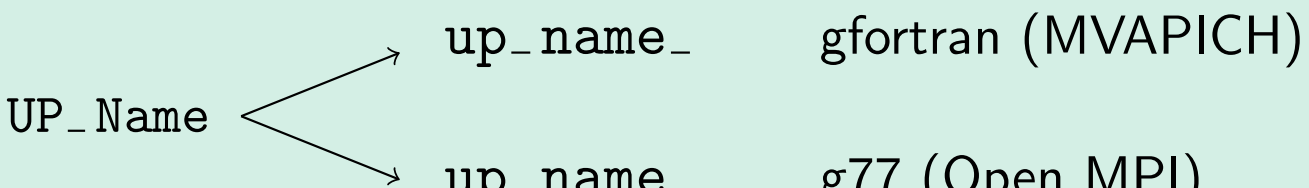
Gesamtzeiten jeweils $\approx 0.1 \ldots 2$ s (bei CLiC: 5. . . 50 s)

Gesammelte Merkwürdigkeiten (1)

- MVAPICH braucht in `hostfile` für jeden von `mpirun` verlangten Prozessor genau einen Eintrag; gleiche Knoten dürfen mehrfach vorkommen; Zuordnung der Prozess-Nummern (`rank`) erfolgt in angegebener Reihenfolge
- Open MPI arbeitet `hostfile` auch mehrfach ab (reihum), um Knoten mit mehreren Prozessen zu füttern.
  Kommt ein Knoten mehrfach in `hostfile` vor, erhält er zunächst entsprechend viele Prozesse, bevor andere Knoten belegt werden (!)
- MPI-Routinen verbrauchen (wie von MPICH bekannt) 100% CPU-Zeit, während sie auf einen Kommunikationspartner warten. $\Rightarrow$ hängende Prozesse stören
  Beachte: MPI beendet u. U. nicht alle Prozesse!
  MVAPICH: Nutzerprogramm auf Knoten > 0
  Open MPI: ein Prozess namens `orted`

Gesammelte Merkwürdigkeiten (2)

- Typisch für 64-Bit-Maschinen: Sorgfalt bei Verwendung älterer C-Programme (`int`, `long int`, ...)
- Mindestens bei `gfortran` (evtl. auch bei g77)
  - In Fortsetzungszeilen von Formatanweisungen ist nur ≤ 1 Tabulator nach dem Fortsetzungszeichen erlaubt.
  - Bisherige g77-Versionen waren tolerant bzgl. Zeilenlänge (> 72 erlaubt, falls in der Zeile ein Tabulator vorkommt); gilt jetzt nicht mehr!
- Underscore-Mimik bei Fortran-C-Schnittstelle verschieden. Passende Compilerflags sind zwar vorhanden, aber Nutzer muss sich nach Vorgabe der MPI-Installation richten.



Gesammelte Merkwürdigkeiten (3)

- Pathscale Compiler (`pathf90`) mit aggressiver Optimierung:
  `-Ofast` bzw. `-O3 -ipa`
  ⇒ Rechnung war schnell, aber falsch (Version 2.4 des Compilers);
  ⇒ also optimal für 

  Mit Compilerversion 3.0 ist das Programm nicht mehr ganz so hyperschnell, rechnet dafür aber richtig.
- Erfahrung zeigt aber:
  Bei falschen Ergebnissen ist am Ende doch immer der Nutzer schuld!

  Compilerentwickler sehen grundsätzlich eine ausreichende und erdrückende Anzahl von Flags vor, um am Ende den Anwender für (fast alle) auftretenden Fehler verantwortlich machen zu können ...

  z.B.: Wahl zwischen *exakter* oder *schneller* Gleitkommaarithmetik

z. B. Ausschnitt aus:

man pathf90

```
 ...
-f[no-]unsafe-math-optimizations
          -funsafe-math-optimizations  improves  FP
          speed by violating  ANSI and IEEE  rules.
          -fno-unsafe-math-optimizations  makes the
          compilation conform to ANSI and IEEE math
          rules at the expense of speed.
 ...
```

Zusammenfassung

- *Gutartige Anwendungen* mit getesteten Compilern und MPI-Installationen gut verträglich, keine nennenswerten Laufzeitunterschiede – nur verschiedene Handhabung beachten
- Kommunikationsnetz erfüllt Erwartungen sehr gut: Trotz Vervielfachung der Rechenleistung bleibt Anteil der Kommunikation im Bereich von wenigen Prozent.
- Dual-Board-Dual-Core-Knoten bringen bei herkömmlichen Anwendungen *Verluste* gegenüber der Verwendung nur je eines Prozessors auf jedem 4er-Knoten
- Bei sehr großen Paketlängen machen sich schlechte Realisierungen der MPI-Routinen für *globale Kommunikation* bemerkbar, besonders auffällig bei *Open MPI*

Nachschlag: Medieninteresse

Medienberichte im Wandel der Zeiten

22.3./25.10.1994

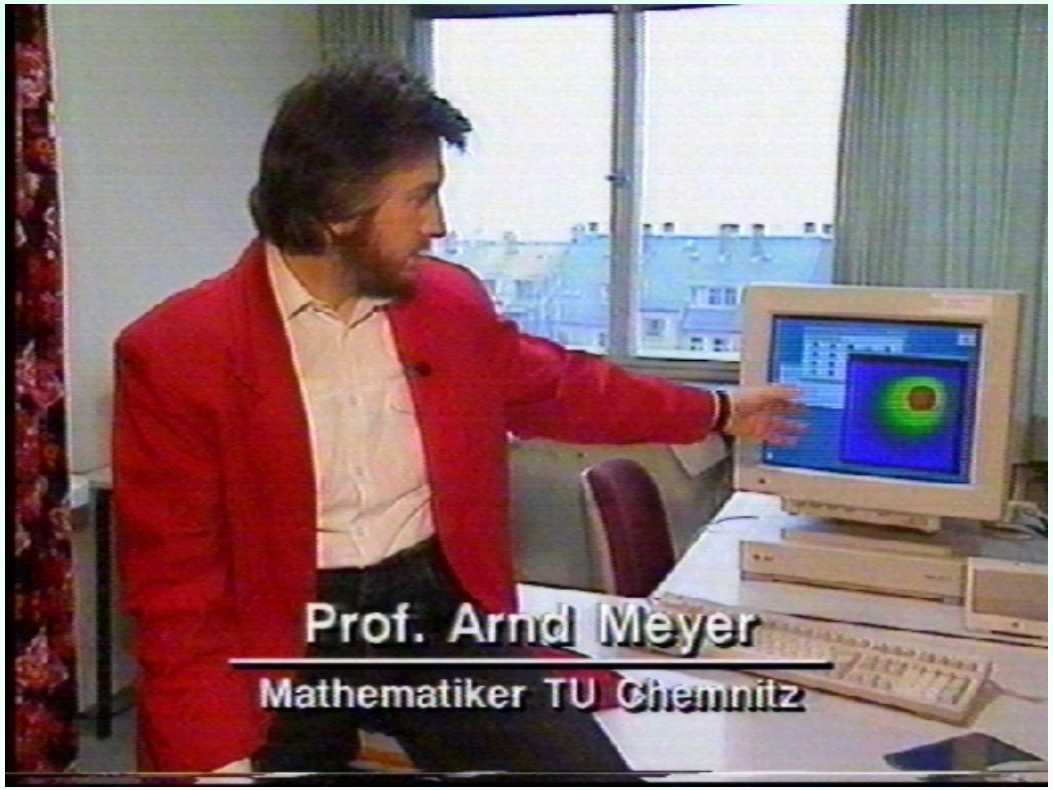
11.10.2000

07.02.2007

GCel-192 / GCPP-128 (1994)

return

GCel-192 / GCPP-128 (1994)



return

CLiC (2000)

return

CHiC (2007)

return