Szeregowanie zadań w Linux Kernel 2.6

Michał Gryglicki
Łukasz Kowalski
Piotr Domański

Plan

- Szeregowanie
- Cele szeregowania
- Scheduler 2.6
- Interaktywności
- Wieloprocesorowość
- Tuning schedulera
- 2.6 vs 2.4

Ogólnie szeregowanie

- Wyłączny dostęp do niektórych zasobów
- Sprawiedliwość szeregowania
  - zapobieganie zagłodzeniu procesów
- Kosztowność szeregowania
  - czas potrzebny na wybranie kolejnego procesu do pracy
  - czas potrzebny na przyznanie priorytetów
- Dokonywanie wyboru
  - rozmiar kwantu czasu
  - kolejność wyboru procesów

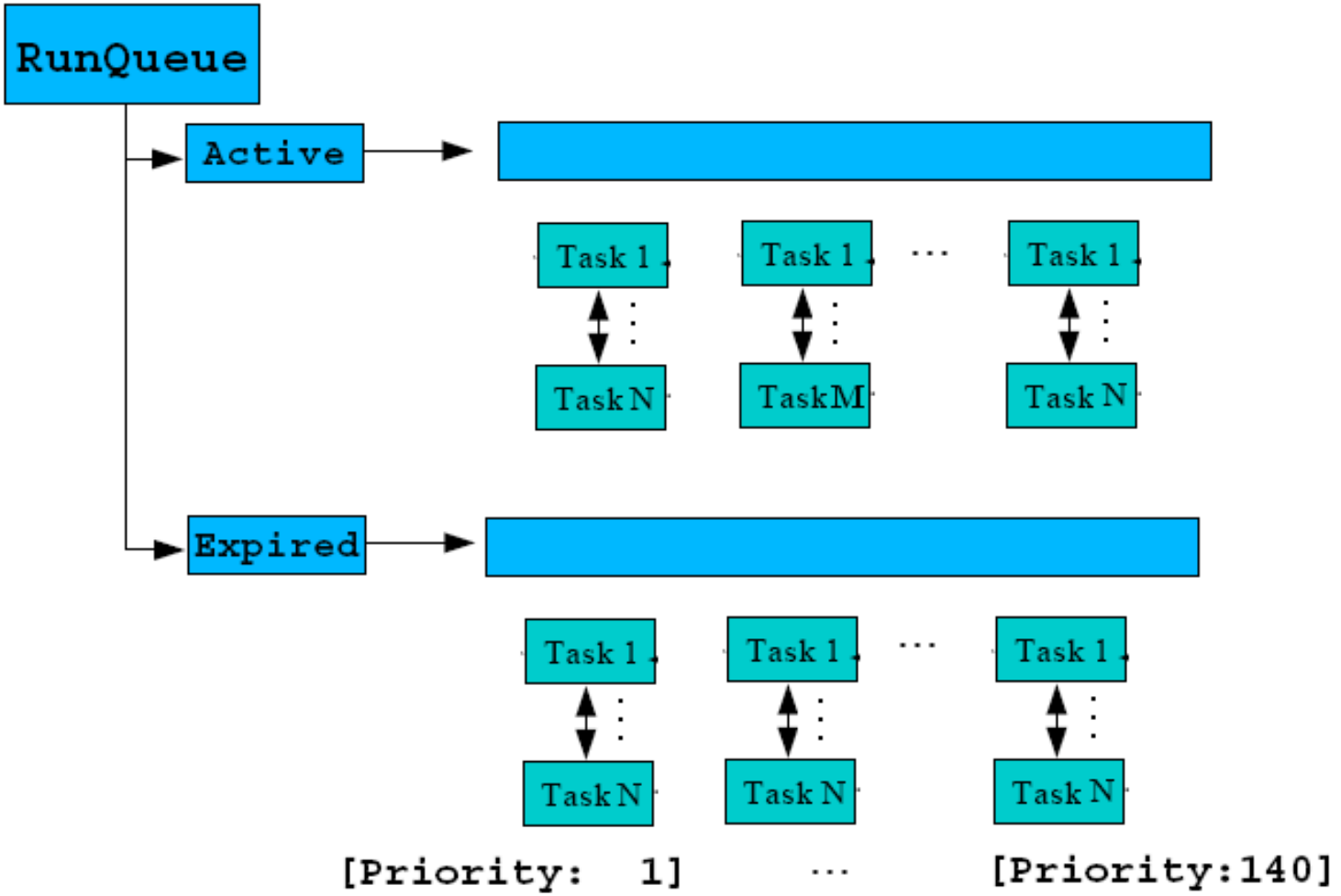
Scheduler – Czego oczekiwać?

- Duża wydajność
  - Algorytm wyboru procesu
  - najlepiej niezależny od liczby procesów: O(1)
- Duża inteaktywność podczas obciążenia systemu
  - Pierwszeństwo dla procesów interakcyjnych
- Sprawiedliwość i zapobieganie zagłodzeniom
- System priorytetów
  - podział procesów według ważności ułatwiający wybór kolejnego do działania
- Wspieranie wieloprocesorowości (SMP, SMT, NUMA)
  - współbierzny wybór kolejnego procesu
- Obsługa procesów czasu rzeczywistego
- - różne strategie szeregowania

Scheduler 2.6

- Algorytm O(1) dzięki strukturom *runqueue* i *priority_array*
- System wspierania interaktywności wykrywanej dzięki *sleep_avg* (ile średnio proceś śpi) i *interactive_credit*
- 140 poziomów priorytetu
  - mniejsza wartość – większy priorytet
  - 0-100 priorytety procesów czasu rzeczywistego (bezwzględne pierwszeństwo przed zwykłymi)
  - 101-139 priorytety procesów zwykłych
- Wspieranie wieloprocesorowości dzięki oddzielnym kolejkom dla każdego procesora i *load_balance()*

Rzut oka na tablice priorytetów i kolejki procesów jednego procesora



Dlaczego wybór działa w czasie O(1)?

- Każda *runqueue* zawiera 2 tablice *priority_array* (indeksowane priorytetami, których wartościami są kolejki czekających procesów o danym priorytecie):
  - tablica procesów aktywnych (*active*)
  - tablica procesów zużytych (*expired*)
- oraz mapę bitową pomagającą znaleźć proces o najwyższym priorytecie (1 w mapie znajduje się na i-tym bicie, jeśli i-ta kolejka aktywnej tablicy priorytetów (*active*) jest niepusta
- Znalezienie pierwszego bitu = 1 w mapie jest bardzo szybkie
- Wybieramy zawsze proces z kolejki aktywnych o najwyższym priorytecie

Zatem algorytm O(1) w punktach:

1. Znajdujemy w mapie bitowej kolejki aktywnych pierwszy bit = 1 (jest to najwyższy priorytet spośród posiadanych przez procesy czekające
2. Bierzemy kolejke procesów znajdującą się w *active*[nr_bitu] i zdejmujemy pierwszy proces z tej kolejki
3. Wybieramy ten proces do pracy

Ważne: Algorytm niezależny od liczby procesów działających w systemie

(1) Strategie szeregowania w Scheduler 2.6

- 101-139 priorytety zwykłych zadań
- SCHED_NORMAL
  - priorytet statyczny + priorytet dynamiczny
  - przydział kwantów czasu

(cyklicznie wszystkie procesy w kolejności od najwyższych priorytetów)

(2) Strategie szeregowani w Schduler 2.6

- 0-100 priorytety procesów czasu rzeczywistego
- SCHED_FIFO (FIRST IN FIRST OUT)
  - priorytet statyczny
  - proces kończy działanie kiedy dobrowolnie odda procesor
- SCHED_RR (ROUND-ROBIN)
  - priorytet statyczny
  - przydział kwantów czasu
    (cyklicznie wszystkie procesy w obrębie najwyższego priorytetu, następnie tak samo procesy o priorytecie niższym itd.)

!! Obie strategie mogą powodować zagłodzenia !!

Priorytety procesów użytkownika

- Priorytet statyczny (*nice*)
  -20 <= *nice* <= 19
  MAX_RT_PRIO = 100
  PRIO = *MAX_RT_PRIO* + *nice* + 20

- Priorytet dynamiczny (-5 ... +5)
  (przyznawany w zależności od interakcyjności)

✔ procesy ograniczone przez I/O (*I/O-bound*)
(korzystające głównie z urządzeń I/O – interakcyjne są nagradzane przez system)

✔ procesy ograniczone przez procesor
(CPU-bound)
(korzystające głównie z procesora – nieinterakcyjne – są karane przez system)

Sposób przydzielania procesów do interakcyjnych

- Na podstawie heurystyk, biorących pod uwagę jak długo proces śpi (przypuszczalnie na I/O)
- Interakcyjność jest skalowana, gdyż informacja binarna nie oddałaby zachowania procesu (większość procesów w systemie jest w środku tej skali)
- *sleep_avg*
  – zmienna dla każdego procesu pamiętająca ile proces ostatnio spał i działał
  – *sleep_avg* = ile_spał – ile_działał;
  – im wyższa wartość *sleep_avg* tym proces dostanie większą nagrodę w postaci < 0 priorytetu dynamicznego

Sprawiedliwość przy fork()

Kiedy tworzymy nowy proces przy pomocy funkci *fork()*, sleep_avg zarówno ojca jak i syna są zmniejszane

- Zapobiega to tworzeniu przez wysoko interakcyjne procesy, nowych wysoko interakcyjnych procesów, co mogłoby zostać niewłaściwie wykożystane
  – np. proces interakcyjny mógłby utworzyć synów (którzy odziedziczyliby duży bonus do priorytetu) po to aby wykonali dla niego zadanie wymagające dużego zaangażowania procesora (natomiast gdyby wykonał je sam straciłby bonusy za interakcyjność)

Priorytet efektywny

Funkcja effective_prio() wylicza:

- ***prio*** = *proces->static_prio – bonus;*
- ***bonus*** = *CURRENT_BONUS(p) – MAX_BONUS* / 2;

```
#define CURRENT_BONUS(p) \
NS_TO_JIFFIES((p)->sleep_avg) * MAX_BONUS /
  MAX_SLEEP_AVG)
```

- *CURRENT_BONUS* zamienia *sleep_avg* na zakres 0 – *MAX_BONUS* (= 10)
- Zatem bonus jest w zakresie [-5 .. +5]
- Na końcu sprawdza czy może przyznać procesowi taki bonus nie wyjść poza zakres priorytetów,
  (np. proces o priorytecie statycznym -19 nie może dostać bonusu równego -5 !!)

Wyliczanie kwantów czasu

- Kiedy proces kończy swój kwant czasu (kiedy jest dodawany do kolejki procesów zużytych)
- Zależy tylko od *static_priority* (im w wyższy tym większy kwant czasu proces dostanie)
- Funkcja task_timeslice() wylicza:

```
#define BASE_TIMESLICE(p)  (MIN_TIMESLICE + \
((MAX_TIMESLICE – MIN_TIMESLICE) * \
(MAX_PRIO – 1 – (p)->static_prio) / (MAX_USER_PRIO – 1)))
```

  - skaluje priorytet statyczny na wartość: MIN_TIMESLICE - MAX_TIMESLICE

- TIMESLICE_GRANUALITY (maksymalny czas po którym jest sprawdzane, czy inny proces o takim samym priorytecie nie czeka => jeśli tak to RR wykonuje te procesy na zmianę)

Ponowne umieszczanie w kolejce procesów aktywnych

- Procesy interakcyjne po wykorzystaniu swojego kwantu czasu są ponownie wrzucane do kolejki procesów aktywnych z nowym kwantem czasu

- Dzieje się tak, tak długo, dopóki żadne procesy w kolejce procesów zużytych nie są zagładzane
  - tzn. czekają dłużej dłużej niż zmienna STARVATION_LIMIT
- Nie pozwala to interakcyjnym procesorm czekać w kolejce procesów zużytych, gdy nieinterakcyjne procesy zużywają swoje kwanty czasu (a to ogranicza czas reakcji systemu na I/O)

(1) Poziom interakcyjności - więcej

- Przechowywany w zmiennej *interactive_credit*
- Rosnącej kiedy proces śpi i zmniejszającej się kiedy używa CPU
- > 100 => procesy wysoce interakcyjne
- < -100 => procesy mało interakcyjne
- Po co to?
  - rodzaj “reputacji” procesu w systemie
  - pozwala odróżnić procesy interakcyjne, które czasem potrzebują dłużej użyć procesora i nieinterakcyjne, które czasem dłużej pośpią i nie dawać im zbyt dużych kar / nagród.

(2) Poziom interakcyjności - więcej

- Procesy o niskiej wartości budzone z uninterruptible sleep mają zmniejszony wzrost sleep_avg, gdyż są najprawdopodobniej procesami CPU-bound czekającymi na I/O tylko czasami
- Procesy o wysokiej wartości mają o mniejszą wartość zmiejszane sleep_avg, żeby nie traciły statusu interakcyjności zbyt szybko
- Procesy o niskiej wartości mogą dostać tylko raz bonus do priorytetu dynamicznego od sleep_avg (jeśli jest < 0), gdyż pewnie nie mogli spać zbyt długo ostatnio skoro mają niską interakcyjność

(1) Struktura *runqueue*

- *spinlock_t lock*
  - tylko jeden proces może modyfikować kolejkę w danym czasie
- *unsigned long nr_running*
  - liczba gotowych procesów w kolejce
- *unsigned long long nr_switches*
  - liczba zmian kontekstu od momentu powstania kolejki (nigdzie nie używane)
- *atomic_t nr_iowait*
  - liczba procesów czekających na I/O

(2) Struktura *runqueue*

- *unsigned long nr_uninterruptible*
  - liczba zadań w kolejce, które nie obsługują przerwań
- *unsigned long expired_timestamp*
  - czas ostatniej zmiany epoki
- *int best_expired_prio*
  - najwyższy priorytet z procesów zużytych

(3) Struktura *runqueue*

- *task_t *curr*
  - wskaźnik do aktualnie działającego procesu
- *task_t *idle*
  - wskaźnik do procesu który działa wtedy, gdy nic innego nie działa
- *prio_array_t *active*
- *prio_array_t *expired*
  - wskaźniki do tabel

Struktura *prio_array*

- *unsigned int nr_active*
  - liczba gotowych zadań w kolejce
- *unsigned long bitmap[BITMAP_SIZE]*
  - do szukania najwyższego priorytetu
- *struct list_head quene[MAX_PRIO]*
  - tablica list procesów o danych priorytetach

Wsparcie Scheduler 2.6 dla SMP

- System SMP jest podzielony na węzły, z których każdy jest jakimś procesorem, albo kolejnym systemem podobnym do SMP
- Każdy procesor ma własną kolejkę *runqueue*
- Wyrównywanie obciążeń między procesorami
  - wyrównywanie tylko wewnątrz węzła
- *Runqueue* ma element *cpu_load*, który się uaktualniany przez *rebalance_tick()*
  - *cpu_load_curr* = liczba procesów w kolejce *active*
  - *cpu_load = AVG (cpu_load_curr, cpu_load_ old):*
- Jeśli domena nie była wyrównywana dłużej niż "pewną ilość czasu" *rebalance_tick()* wywołuje *load_balance()*

load_balance()

- Wywoływana co:
  - Co 1ms, gdy procesor jest bezczynny
  - Co 200ms, gdy pracuje
- szuka najbardziej obiążonego procesora
- jeśli takiego nie ma, lub aktualny CPU jest w tej grupie to nic nie robi
- Wpp. Jeśli różnica ilości zadań jest większa niż 25% to przenosi procesy z grupy bardziej obciążonej do kolejki procesów aktywnych aktualnego CPU: korzystając z funkcji *move_tasks()*
- Procesy zawsze są ściągane przez procesor mniej obciążony z procesora bardziej obciążonego, nigdy nie są wysyłane

move_tasks()

- *move_tasks()*:
  - wywoływana przez procesor mniej obciążony, aby wyszukać w systemie procesory bardziej obciążone i odebrać im część pracy
- Sposób wyboru:
  - najpiew próbuje wybierać procesy z tablicy zużytych (*expired*)
    - zaczyna od tych z najniższymi priorytetami
  - sprawdza, czy proces może zostać przeniesiony na aktualny procesor (mapa bitowa preferencji i zależności proces-procesor)
  - powtarzane dopóki nie są zrównoważone (w sensie tych 25%)
  - *pull_task()* przenosi proces z jednej kolejki do drugiej

Tuning Schedulera

- Do wprowadzenia jakichkolwiek zmian wymagana jest rekompilacja jądra linuxa

- *MIN_TIMESLICE* / *MAX_TIMESLICE*
  – przedział wielkości kwanu czasu
  – zwiększając, zwiększa się ogólna wydajność systemu, kosztem czasu reakcji

Tuning Schedulera

- *PRIO_BONUS_RATIO*
  – zakres bonusu do priorytetu dynamicznego (w % promienia rozpiętości nice())
  - im większa, tym mniejsze znaczenie priorytetu statycznego
  (domyślnie 25% ==> 25%*20 = 5)

Tuning Schedulera

- *MAX_SLEEP_AVG*
  – ograniczenie na parametr sleep_avg
  – zwiększenie, zwiększa rozkład procesora pomiędzy procesy interakcyjne i nieinterakcyjne
  - im wyższa wartość tym trudniej uzyskać maksymalne bonusy

Tuning Schedulera

- *STARVATION_LIMIT*
  – jeśli któryś proces w tablicy zużytych czeka dłużej od tego czasu, to procesy interakcyjne nie mogą być ponownie wkładane do kolejki aktywnych!
  (idą do kolejki zużytych)

Co można usprawnić?

- Możliwość dynamiczne zmiany ustawień schedulera (przez roota – komenda)

  np. większe wspieranie interacyjności na stacjach roboczych i odwrotnie większe wspieranie procesów nieinterakcyjnych na serwerze

Główny scheduler rozdzielający czas między użytkowników; każdy użytkownik sam wybiera rodzaj schedulera dla swoich programów

2.6 vs 2.4

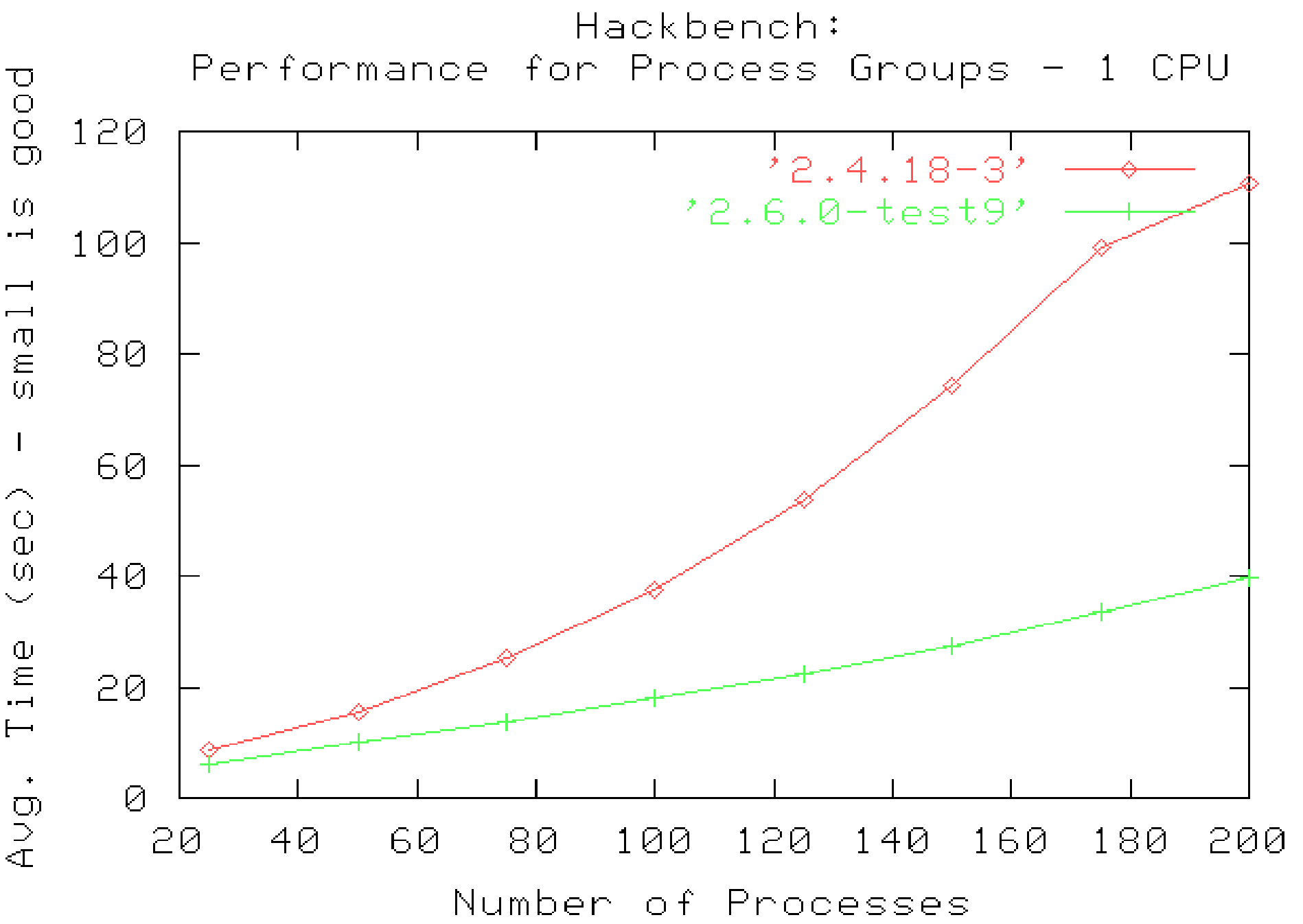
Scheduler w jądrze 2.4:

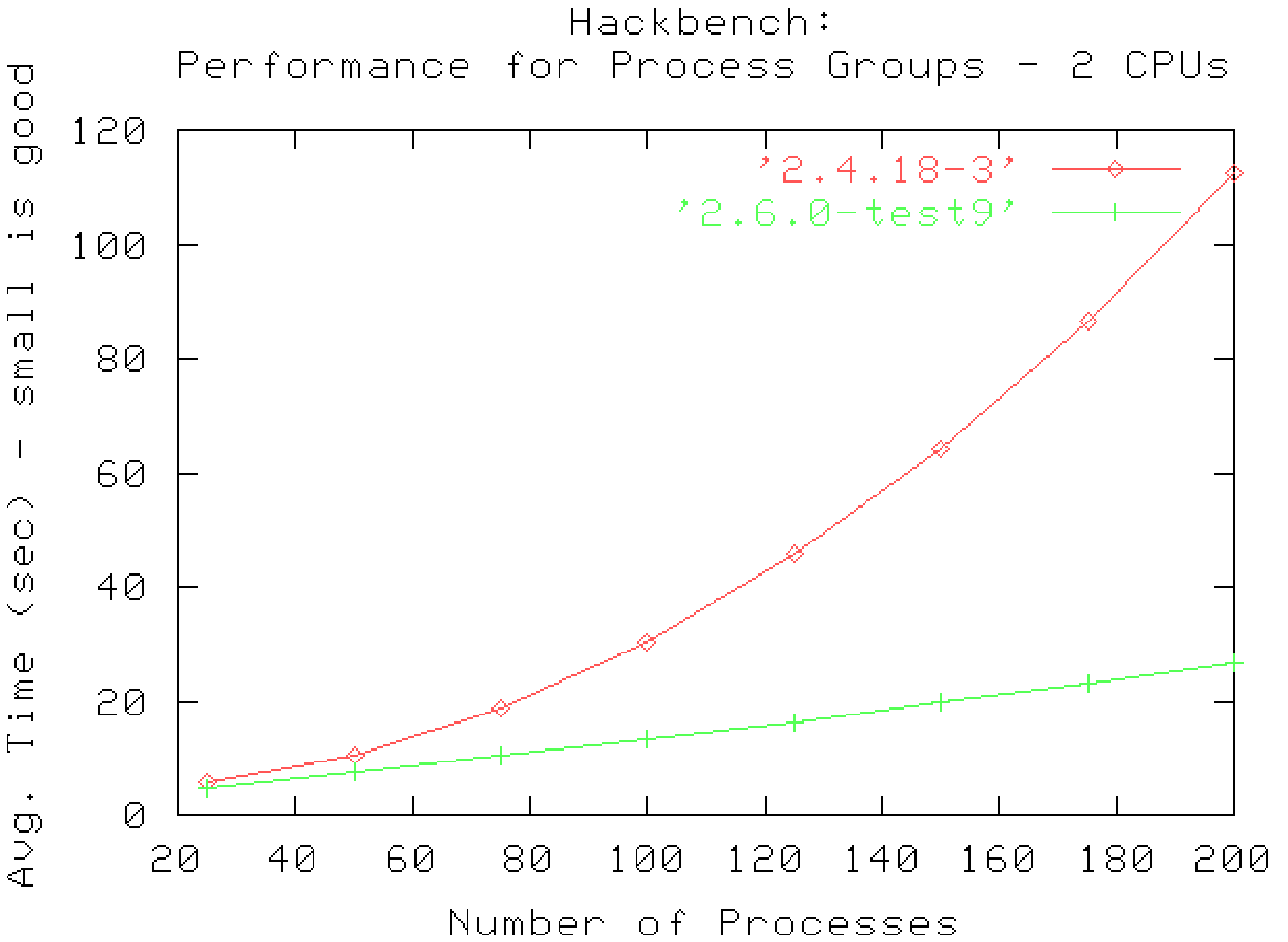
- Algorytm wyboru procesu z najwyższym priorytetem O(n) (przegląda całą listę)
- Długie kwanty czasu (210ms) w porównaniu z 2.6 (100ms)
- Odnawianie priorytetów O(n)
- Jądro 2.6 dla procesów RT jest niewywłaszczalne
- Silnie powiązanie interaktywności i sleep_time (jeśli proces interakcyjny potrzebuje wykonać jakąś pracę wymagającej długiego czasu procesora, straci wiele bonusów (brak interactive_credits)
- Globalna kolejka procesów gotowych dla wszystkich procesorów

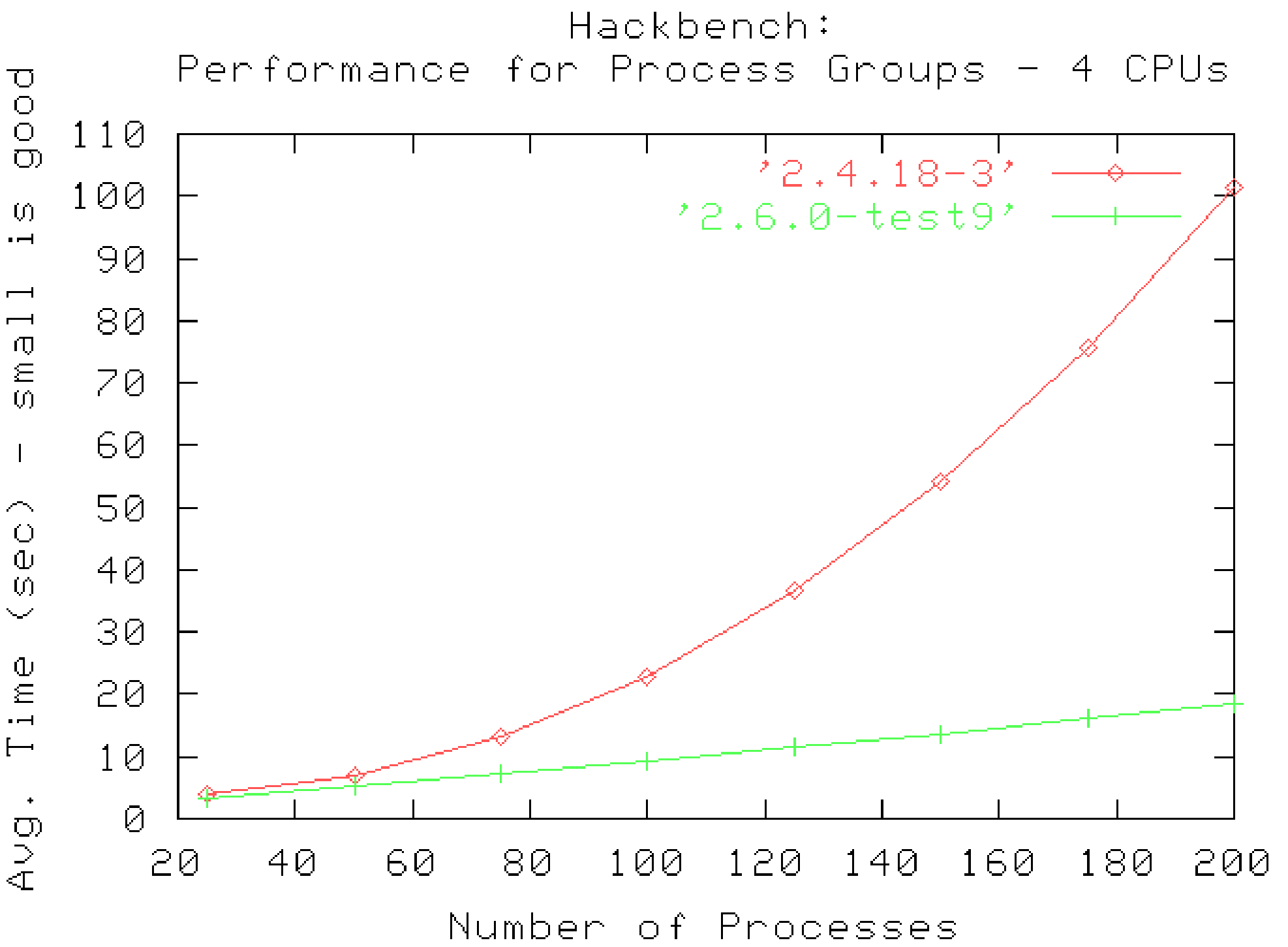
Testy porównujące

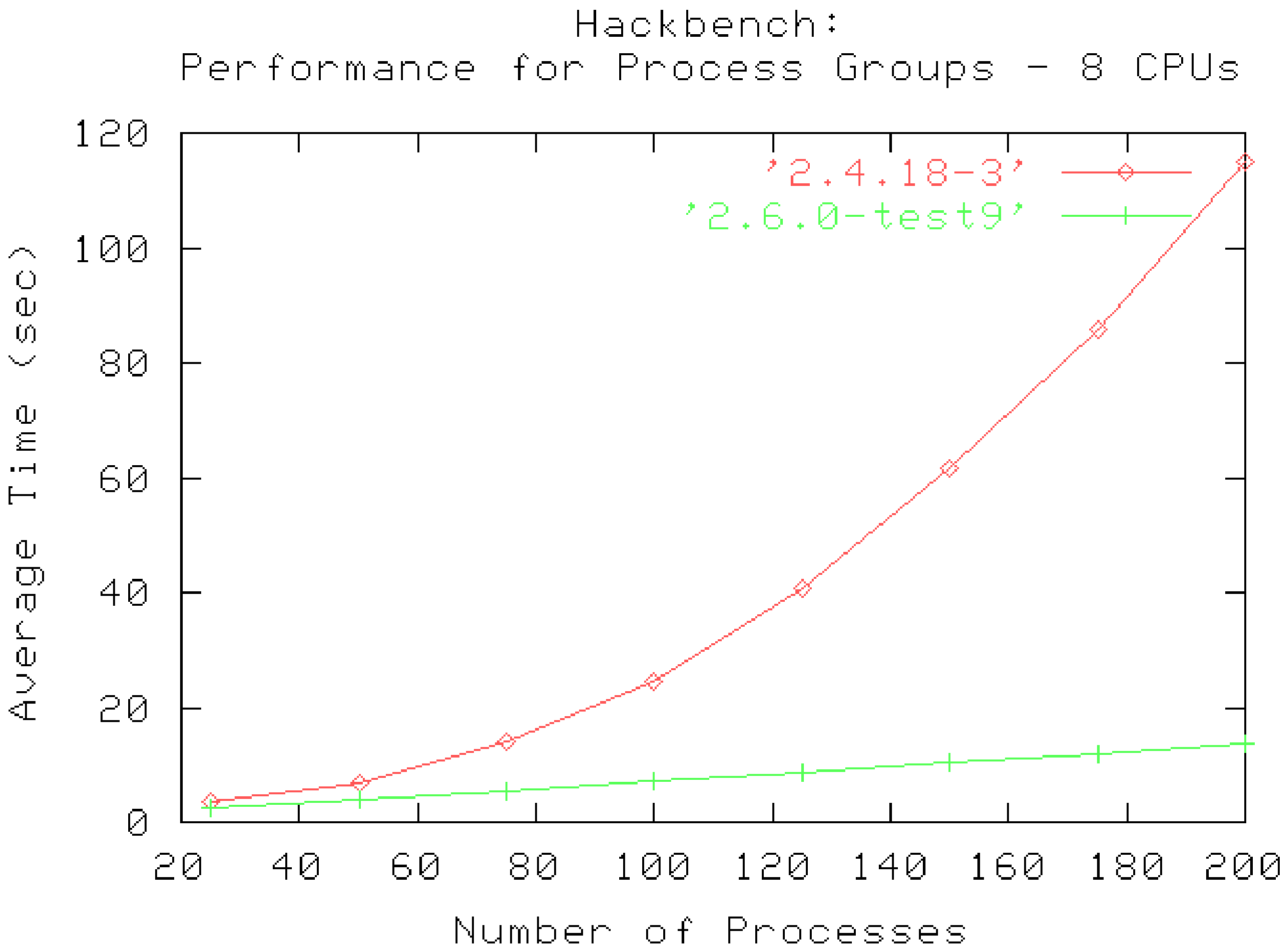
- Jak działa?
  - na maszynach o 1, 2, 4, 8 procesorach
  - dla 25, 50, 75, 100, ... par procesów klient/serwer
  - każdy klient wysyłał do każdego serwera 100 komunikatów, serwer nasłuchiwał
  - każdy test był wykonywany kilka razy
  - czas w slajdach jest średnim czasem trwania testu
- Jak liczba procesów wpływa na czas trwania testu (4 slajdy)
- Ile daje zwiększenie liczby procesorów - skalowalność. Wykonywano test zwiększając liczbę procesów w systemie, aż do błędu braku zasobów (3 kolejne slajdy)

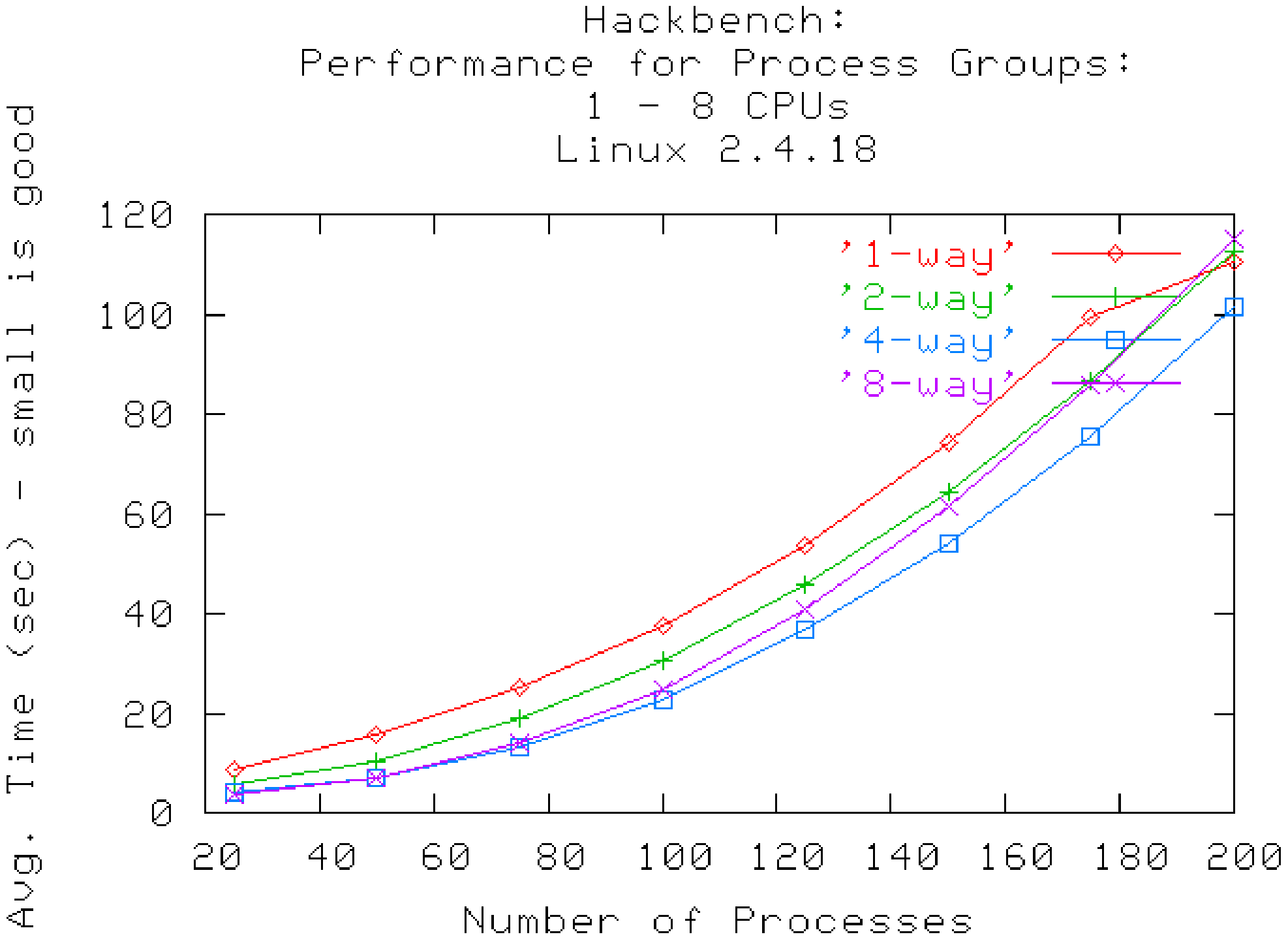
http://developer.osdl.org/craiger/hackbench

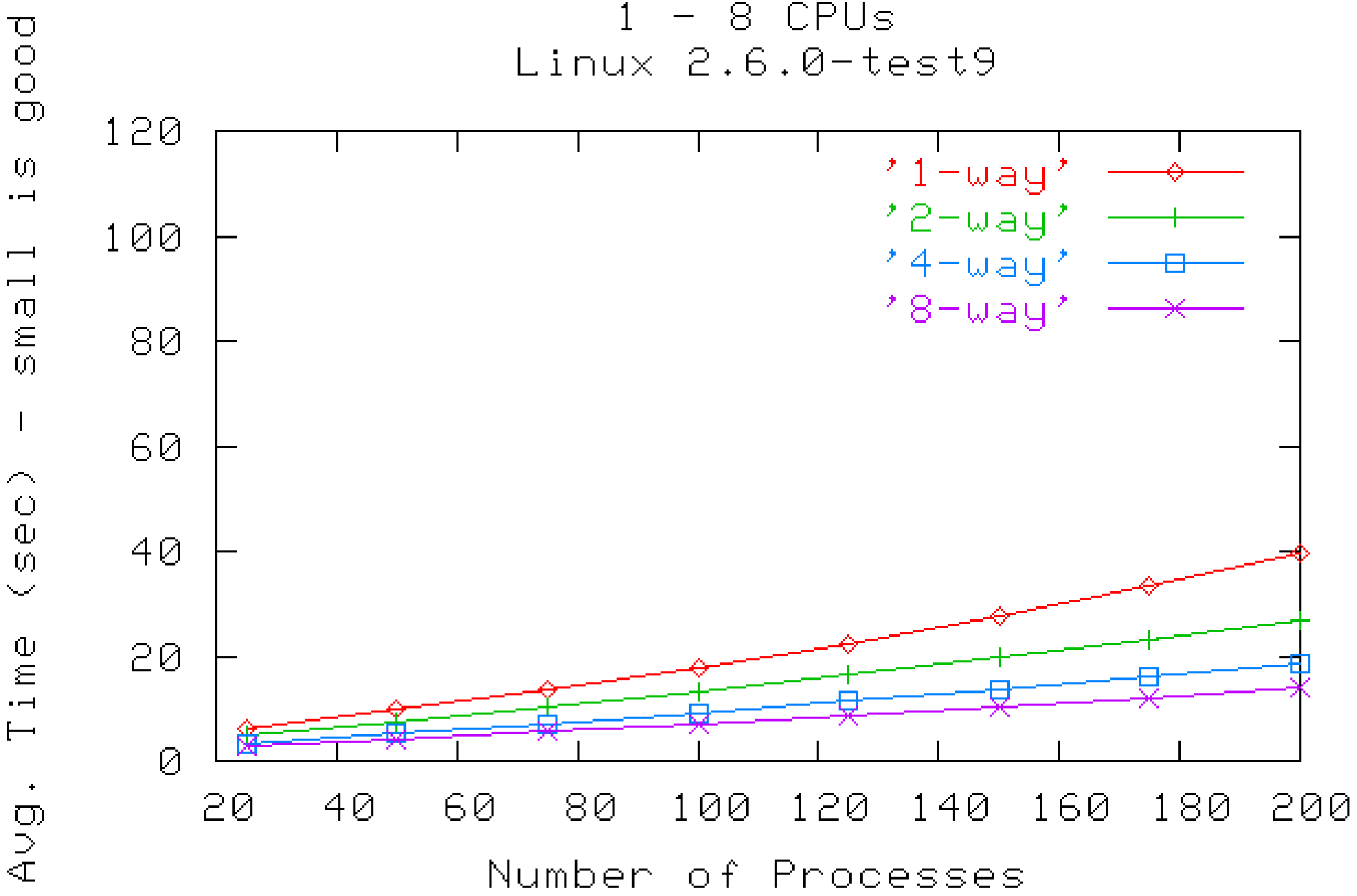

Hackbench:
Performance for Process Groups - 1 CPU
Avg. Time (sec) - small is good
Number of Processes
'2.4.18-3'
'2.6.0-test9'
0
20
40
60
80
100
120
20
40
60
80
100
120
140
160
180
200

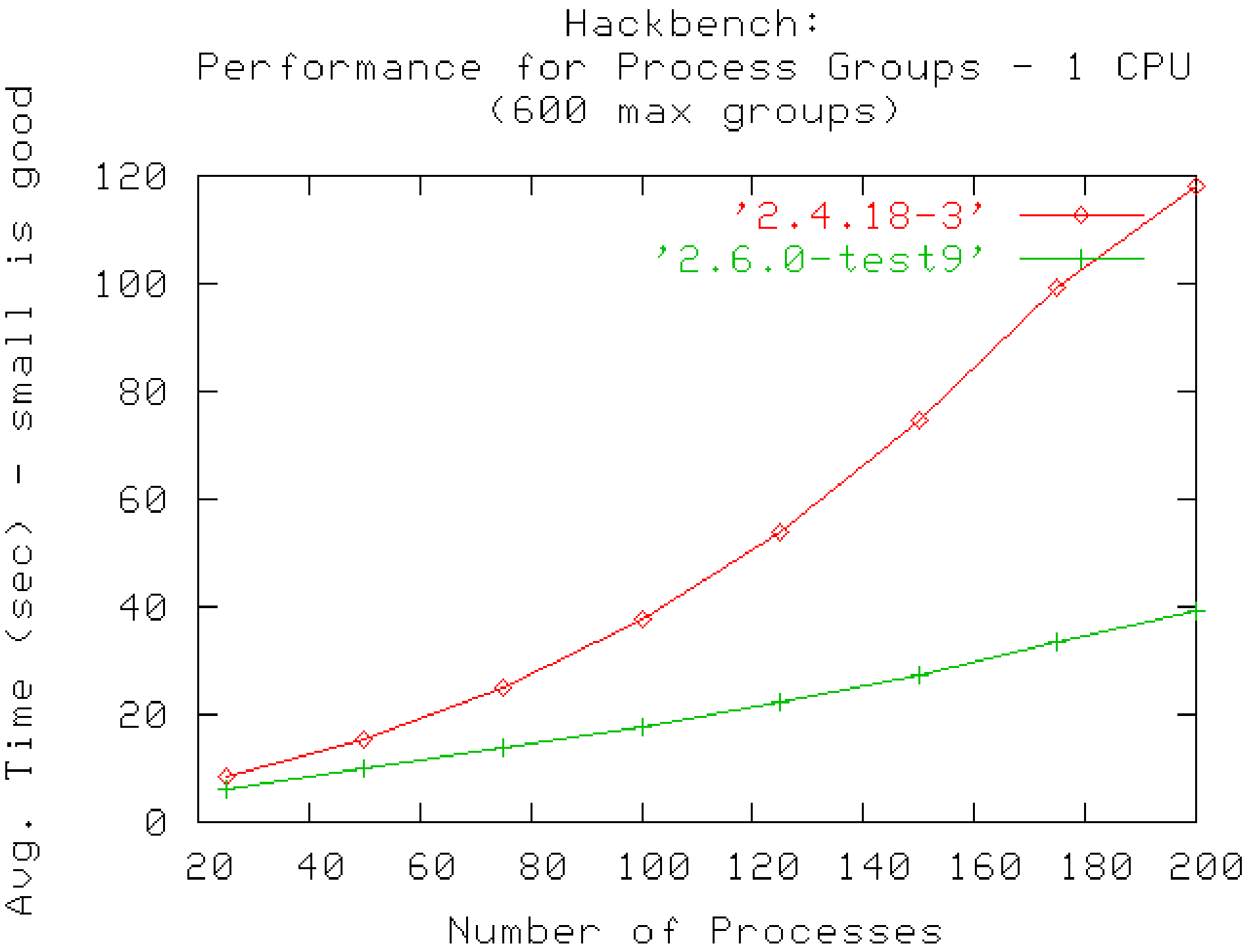

Hackbench:
Performance for Process Groups - 2 CPUs
'2.4.18-3'
'2.6.0-test9'
Avg. Time (sec) - small is good
Number of Processes
0
20
40
60
80
100
120
140
160
180
200

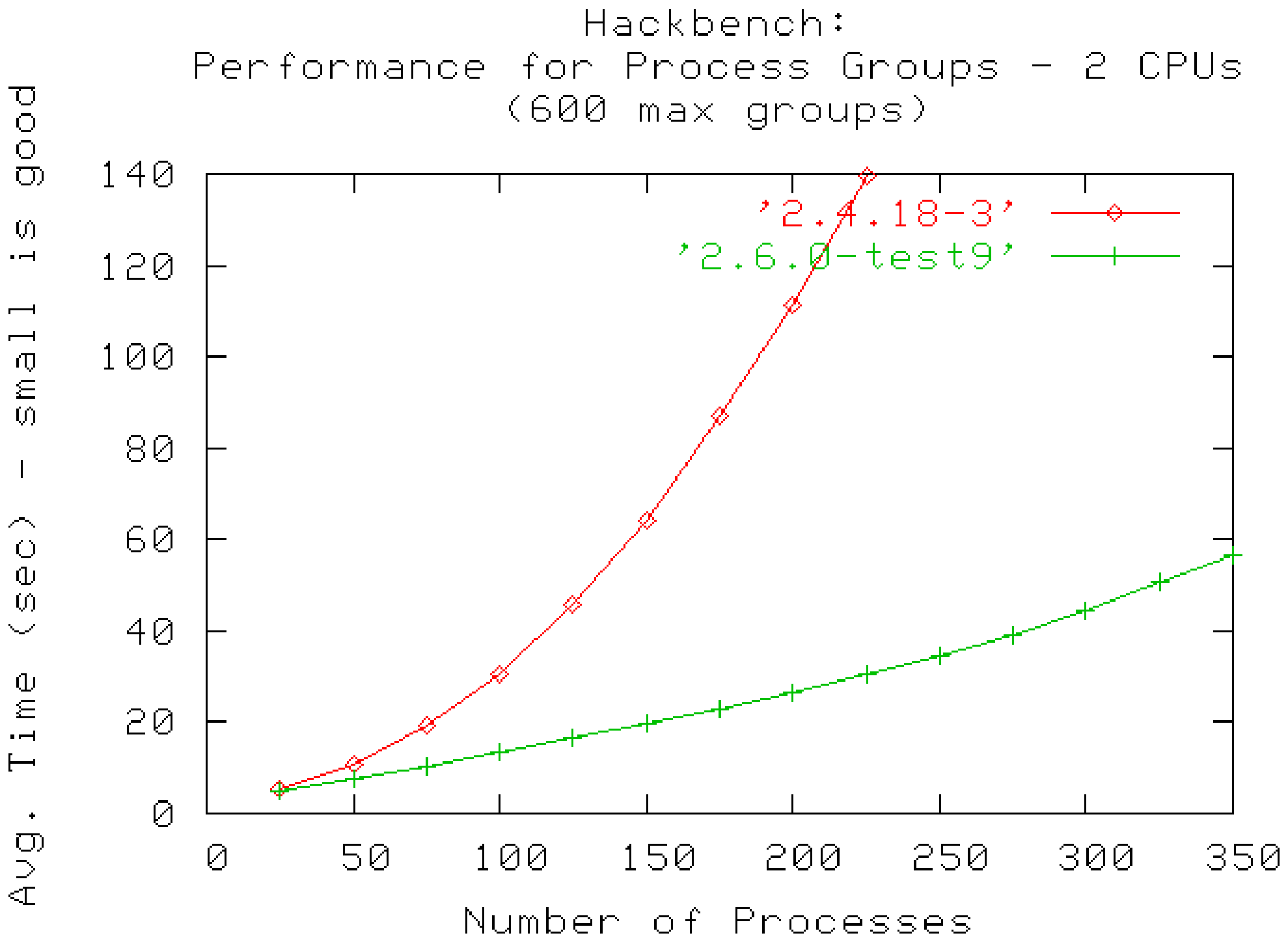

Hackbench:
Performance for Process Groups - 4 CPUs
Avg. Time (sec) - small is good
Number of Processes
'2.4.18-3'
'2.6.0-test9'
0
10
20
30
40
50
60
70
80
90
100
110
20
40
60
80
100
120
140
160
180
200

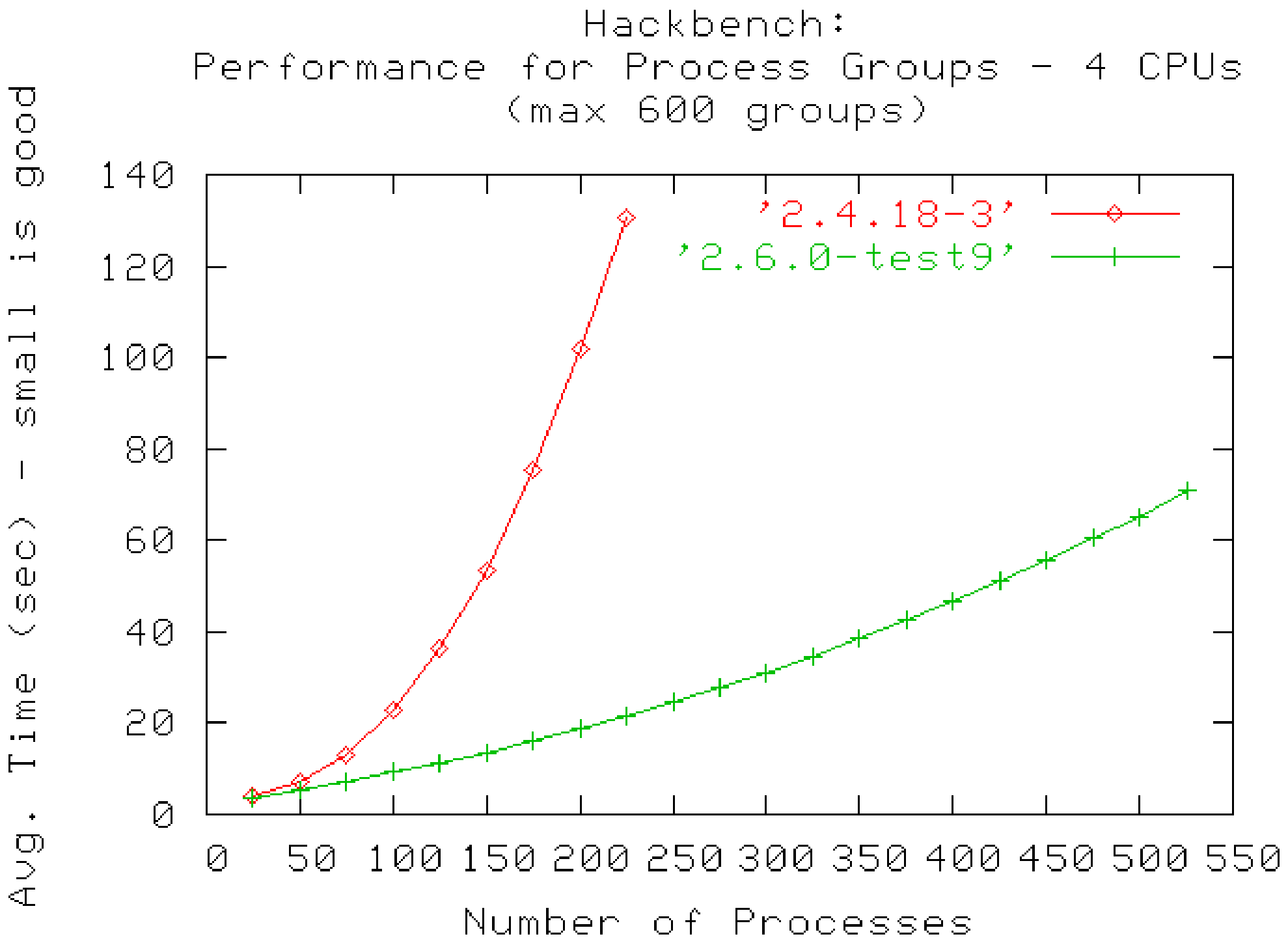

Hackbench:
Performance for Process Groups - 8 CPUs
'2.4.18-3'
'2.6.0-test9'
Average Time (sec)
0
20
40
60
80
100
120
20
40
60
80
100
120
140
160
180
200
Number of Processes


Hackbench:
Performance for Process Groups:
1 - 8 CPUs
Linux 2.4.18
Avg. Time (sec) - small is good
Number of Processes
'1-way'
'2-way'
'4-way'
'8-way'
0
20
40
60
80
100
120
20
40
60
80
100
120
140
160
180
200


Hackbench:
Performance for Process Groups:
1 - 8 CPUs
Linux 2.6.0-test9
Avg. Time (sec) - small is good
Number of Processes
'1-way'
'2-way'
'4-way'
'8-way'
0
20
40
60
80
100
120
20
40
60
80
100
120
140
160
180
200


Hackbench:
Performance for Process Groups - 1 CPU
(600 max groups)
Avg. Time (sec) - small is good
Number of Processes
'2.4.18-3'
'2.6.0-test9'
0
20
40
60
80
100
120
20
40
60
80
100
120
140
160
180
200


Hackbench:
Performance for Process Groups - 2 CPUs
(600 max groups)
Avg. Time (sec) - small is good
Number of Processes
'2.4.18-3'
'2.6.0-test9'
0
20
40
60
80
100
120
140
0
50
100
150
200
250
300
350


Hackbench:
Performance for Process Groups - 4 CPUs
(max 600 groups)
Avg. Time (sec) - small is good
Number of Processes
'2.4.18-3'
'2.6.0-test9'
0
20
40
60
80
100
120
140
0
50
100
150
200
250
300
350
400
450
500
550