PORR DOKUMENTACJA

Michał Przyłuski

3 lutego 2010

1 Cel projektu

Celem projektu była implementacja, sekwencyjna i równoległa, wybranych metod optymalizacji.

1.1 Metoda Jacobiego

Podstawową metodą iteracyjną szukania ekstremum (bez straty ogólności będziemy w dalszym ciągu zakładali, że jest to minimum) funkcji jest metoda Jacobiego [1]. Jej elementarna iteracja jest dana wzorem:

$$x_i^{(k+1)} = \operatorname{Arg}\min_{x_i \in X_i} F\left(x_1^{(k)}, \ldots, x_i, \ldots, x_m^{(k)}\right). \tag{1}$$

Algorytm Jacobiego jest metodą klasyczną, znaną od wielu lat. Łatwo można ją zrównoleglić, poprzez niezależne szukanie minimum po każdym X_i. Niestety, metoda ta ma umiarkowaną szybkość zbieżności.

1.2 Metoda Gaussa-Seidela

Pewnym rozwinięciem metody Jacobiego jest metoda Gaussa-Seidela [1]. Elementarna iteracja określona jest wzorem:

$$x_i^{(k+1)} = \operatorname{Arg}\min_{x_i \in X_i} F\left(x_1^{(k+1)}, \ldots, x_{i-1}^{(k+1)}, x_i, x_{i+1}^{(k)} \ldots, x_m^{(k)}\right). \tag{2}$$

1.3 Proksymalna metoda Gaussa-Seidela

Dalszym rozwinięciem metody GS jest proksymalna metoda Gaussa-Seidela. Czynnik proksymalny odpowiedzialny jest za uwypuklenie funkcji celu. Metoda ta została opisana w [2]. Jej iteracja dana jest wzorem:

$$x_i^{(k+1)} = \operatorname{Arg}\min_{x_i \in X_i} F\left(x_1^{(k+1)}, \ldots, x_{i-1}^{(k+1)}, x_i, x_{i+1}^{(k)} \ldots, x_m^{(k)}\right) + \tfrac{1}{2}\tau_i \|x_i - x_i^{(k)}\|^2. \tag{3}$$

Ponadto w pracy [2] wykazano, że metoda ta jest zbieżna.

1.4 Proksymalna metoda Jacobiego

Skrzyżowaniem powyższych metod jest proksymalna metoda Jacobiego. Jest to metoda Jacobiego, po dodaniu czynnika proksymalnego jak w proksymalnej metodzie GS. Dzięki temu, że nie uaktualniamy wektora $x^{(k)}$ po każdej iteracji, możliwe jest równoległe wykonywanie iteracji tej metody. Iterację można określić jako:

$$x_i^{(k+1)} = \operatorname{Arg}\min_{x_i \in X_i} F\left(x_1^{(k)}, \ldots, x_i, \ldots, x_m^{(k)}\right) + \tfrac{1}{2}\tau_i \|x_i - x_i^{(k)}\|^2. \tag{4}$$

Na podstawie [3] wydaje się, że cała klasa metod proksymalnych jest zbieżna.

2 Realizacja sekwencyjna

W pierwszej kolejności zrealizowana została implementacja sekwencyjna proksymalnej metody Jacobiego. Wykorzystano do tego celu solver Ipopt. Naturalną wydaje się następująca logiczna organizacja przebiegu programu:

- Dokonaj podziału wektora zmiennych na kilka podwektorów.
- Minimalizuj funkcję na kolejnych podwektorach (w dalszym ciągu będziemy nazywać tę operację „poditeracją"). Po każdej minimalizacji, nowe współrzędne są modyfikowane w punkcie startowym dla kolejnego wywołania procedury minimalizującej.
- Po przebiegnięciu przez wszystkie podwektory sprawdź warunek stopu algorytmu, tj. czy nowy punkt jest znacząco różny od starego (za granicę uznano $\|x^{(k+1)} - x^{(k)}\|^2 < 10^{-6}$).

Cały jeden przebieg wg. powyższego schematu będziemy nazywać „iteracją". Warto zauważyć, że w rzeczywistości jest to proksymalna metoda Gaussa-Seidela.

W czasie realizacji wersji sekwencyjnej największą trudnością było opanowanie Ipopta. Krytyczne bowiem było takie przekazywanie informacji o bieżącym punkcie startowym oraz podwektorze, na którym minimalizujemy w aktualnej poditeracji (wywołaniu Ipopta), do procedur zwracających wartość funkcji celu, jej gradientu i macierzy drugich pochodnych. Wykorzystano do tego celu parametr `UserDataPtr`.

Ponadto, w przypadku bardziej złożonych funkcji testowych, dość trudne było bezbłędne wprowadzenie hesjanu. Rozważono możliwość wykorzystania aproksymacji drugich pochodnych, jednakże, występowały wtedy problemy ze zbieżnością (Ipopta, nie PGS). Jest to oczekiwane zachowanie, wzmiankowane w dokumentacji Ipopta [5, s. 4.2].

3 Zrównoleglenie OpenMP

Jak można przypuszczać, zrównoleglenie zrealizowano poprzez partycjonowanie dziedziny. Kolejne poditeracje mogą być od siebie wykonywane niezależnie. Naturalną

barierą jest zakończenie wszystkich poditeracji, i przejscie do sekwencyjnej części programu w celu scalenia podwektorów rozwiązań z poszczególnych podinteracji.

W celu efektywnego przydziału i przekazywania zadań została określona struktura, która jest przekazywana z warstwy logicznej (podział zadania) do warstwy obliczeniowej, a dalej poprzez parametr UserDataPtr wywołań IPOPTa aż do procedur obliczających wartości.

```
struct mydata {
int start;
int end;
Number* x;
};
```

Zauważmy, że zawarte w niej są wszystkie informacje potrzebne do wywołania IPOPTa. Pola start oraz end określają początek i koniec bloku składowych wektora x do minimalizacji. Pole x natomiast, zawiera aktualny punkt startowy. Co więcej, po szczęśliwym wykonaniu poditeracji, pole x struktury jest modyfikowane, a funkcja wołająca na tej podstawie uaktualnia swoją wiedzę o nowym punkcie startowym.

W czasie realizacji wersji równoległej opartej na OpenMP natrafiono na szereg trudności.

Niestety, implementacja OpenMP standardowo dostępna w środowisku linux (zwana gomp) nie należy do najlepszych. Brakuje w niej ważnej opcji — sprawiającej, aby wszystkie dynamiczne alokacje pamięci były realizowane w ramach lokalnej (dla każdego wątku) sterty. Przykładem biblioteki, która to udostępnia jest realizacja Intela [7] czy IBMa [6].

Zwykłe wywołania funkcji malloc() ze środka zrównoleglonej pętli są dość powolne. Wynika to z faktu, że pamięć ta jest alokowana na wspólnej stercie (globalnej), zatem moment alokacji musi byc zabezpieczony sekcją krytyczną, aby każdy wątek dostał odrębny fragment pamięci. O ile w przypadku tworzonych w pętli zmiennych można łatwo to usprawnić (poprzez przeniesienie ich alokacji do części sekwencyjnej) to w czasie testów okazało się, że jest to całkowicie nieistotne. Alokowane jest tam, bowiem, bardzo mało pamięci. Prawdziwy problem pojawia się w środku wywołań IPOPTa (IpoptSolve oraz CreateIpoptProblem), nad którymi nie mamy kontroli, a alokują one ogromne ilości pamięci.

Z tego względu zrezygnowano z sekwencyjnej alokacji pewnych drobnych fragmentów pamięci, gdyż przy zmniejszeniu czytelności kodu nie dało to żadnych mierzalnych korzyści.

Co więcej, w przypadku dużych (nie precyzujemy chwilowo co to oznacza) zadań problemem staje się alokacja pamięci sama w sobie. Z jednej strony, alokacja, i wypełnienie jej danymi, zawsze trwa, a jednocześnie problem serializacji dostępu do pamięci (na poziomie fizycznym) staje się bardziej dotkliwy, szyna systemowa ma bowiem ograniczoną przepustowość, a w przypadku systemów o większej niż jeden liczbie rdzeni pozostaje złożony problem arbitrażu na szynie oraz zapewnienia spójności pamięci podręcznej.

4 Zrównoleglenie MPI

Kolejnym etapem było zrównoleglenie przy wykorzystaniu biblioteki MPI. Naturalne wydaje się wykorzystanie do komunikacji między procesami struktury mydata, określonej wcześniej.

Aby ją przesłać przy pomocy biblioteki MPI konieczne jest jej „upakowanie" przy pomocy funkcji MPI_Pack(). Warto zauważyć, że alternatywne rozwiązanie — zdefiniowanie bazującego na strutkurze typu własnego MPI — nie jest dobrym pomysłem w przypadku struktury zawierającej wskaźnik na obszar pamięci o zmiennym rozmiarze. Konieczne byłoby definiowanie typu danych MPI albo o największym interesującym nas rozmiarze problemu (i nie używanie pewnej jego części), albo określenie wielu typów, każdy dla innego problemu. Z tego względu wykorzystanie MPI_Pack() wydaje się najrozsądniejsze.

Komunikacja między procesami oparta jest o ideę zgodną z funkcją MPI_Allgather(). Oznacza to, że każdy proces dzieli się swoimi danymi ze wszystkimi innymi procesami. Ponieważ wywołania komunikacji zbiorowej są synchroniczne to przy wywołaniu funkcji MPI_Allgather() mamy naturalną barierę, wszystkie procesy oczekują na zakończenie obliczeń. W wyniku wywołania tej funkcji, po jej szczęśliwym powrocie, każdy proces posiada pełną wiedzę o problemie.

Niestety, w trakcie testowania okazało się, że istnieją pewne różnice między OpenMPI oraz mpich.

5 Interfejs

Program wywołuje się w trybie wsadowym. Przyjmuje on dwa argumenty: rozmiar zadania oraz liczbę, na ile podwektorów ma być podzielony wejściowy wektor.

Przykładowa interakcja użytkownika w programem znajduje się w załącznikach.

6 Testowanie

W czasie testowania mechanizm sprawdzania pierwszych oraz drugich pochodnych, zaimplementowany w Ipopcie, okazał się niezwykle przydatny.

Dla wszystkich funkcji testowych uzyskiwano zadowalającą zbieżność po 4-6 iteracjach.

Jak wcześniej wspomniano, wraz ze wzrostem liczby wątków rośnie zapotrzebowanie na pamięć.

Testy przeprowadzono dla 3 funkcji, każda w wersji bez ograniczeń oraz z ograczeniami pudełkowymi, dla 6 (czasem 5) rozmiarów danych oraz 10 (względnie 4) wersji podziałów wektora zmiennych.

Pierwsza funkcja, zwana dalej **sumsqr** określona jest wzorem:

$$F(x) = \sum_{i=1}^{n} x_i^2. \tag{5}$$

Programy startowały z punktu $x = (2, \ldots, 2)$.

Drugą funkcją jest *Diagonal quadratic function* określona w [4] wzorem:

$$F(x) = \sum_{i=3}^{n} \tag{6}$$

a zadanie startowano z $x = (3, \ldots, 3)$. Zwano ją dalej **dqf**.

Ostatnia funkcja testowa to rozszerzona funkcja Rosenbrocka:

$$F(x) = \sum_{i=1}^{n/2} 100(x_{2i} - x_{2i-1}^2)^2 + (1 - x_{2i-1})^2 \tag{7}$$

Za punkt startowy przyjęto $x = (-1.2, +1.0, \ldots, -1.2, +1.0)$. Informacje o niej również pochodzą z [4]. Będziemy ją nazywać **extros**.

Minimum wszystkich powyższych funckji wynosi 0. Funkcja (7) osiąga swoje minimum w punkcie $x = (+1.0, \ldots, +1.0)$, podczas gdy (5) i (6) w $x = (0, \ldots, 0)$. Warto nadmienić, że dla problemu sumsqr oraz dqf hesjan jest macierzą diagonalną, W przypadku extros hesjan jest blokowo diagonalny (bloki 2×2).

W przypadku problemów ograniczonych, dodane zostało sztuczne ograniczenie

$$-10 \leqslant x_i \leqslant 10 \quad \text{dla} \quad i \in \{1, \ldots, n\},$$

a do nazwy dodano **-b**.

Problemy rozwiązywano w 6 wymiarach przestrzeni: 20160, 200160, 69160, 2000160, 4000320 oraz 16001280 zmiennych. Będą one w dalszym ciągu skrótowo nazywane odpowiednio **20k**, **200k**, **700k**, **2M**, **4M** oraz **16M**.

Liczba zmiennych została dobrana w taki sposób, aby dla każdego rozmiaru bloku, dla funkcji extros, każdy podproblem zaczynał od $x = (-1.2, \ldots)$, czyli w szczególności nie od $x = (+1.0, \ldots)$.

Zadania dzielono na dziesięć sposobów, tj. na 1, 2, 3, 4, 6, 8, 12, 15, 16 lub 20 bloków. Każdy blok był jednakowego rozmiaru. Nie zmieniano wartości zmiennej środowiskowej OMP_NUM_THREADS, a zatem zawsze była stała pula wątków, za zadania „nadmiarowe" były kolejkowane.

7 Wyniki

Pierwsze 6 tabeli przezentuje czasy działania programu na maszynie **sun16**. Jest to 8-procesorowy serwer Sun X4600, wyposażony w 32GB pamięci. Wykorzystano w nim procesory Opteron 8218, łącznie ma on zatem 16-rdzeni. Każdy z nich taktowany jest zegarem 2.6GHz. Sun podaje w [8], że 1GHz Hypertransport jest w stanie osiągnąć przepustowość 8GB/s.

Drugim komputerem, na którym testowano programy, była **kylie**. Jest to stacja robocza oparta o procesor Intel Q6600 (4 rdzenie) posiadająca 4GB pamięci. Każdy rdzeń pracuje z częstotliwością 2.4GHz. Zgodnie z informacją [9] przepustowość dostępu do pamięci to ok. 12.8GB/s. Kolejne 6 tabeli zawiera odpowiednie czasy dla kylie.

Zarówno sun16, jaki i kylie, pracowały pod kontrolą sytemu operacyjnego linux (Fedora 11 na sunie, Fedora 7.92 na kylie). Wyposażone były w zbliżone wersje gcc (4.4.1 oraz 4.1.2) oraz biblioteki czasu wykonania.

W przypadku OpenMP programy były kompilowane przy pomocy:
gcc -fopenmp -O3 -march=core2 -fomit-frame-pointer -pipe -DNDEBUG -pedantic-errors -Wimplicit -Wparentheses -Wsequence-point -Wreturn-type -Wcast-qual -Wall -Wno-unknown-pragmas –std=gnu99 -I/usr/local/include/coin -c $i.c
gcc -fopenmp -O3 -march=core2 -fomit-frame-pointer -pipe -DNDEBUG -pedantic-errors -Wimplicit -Wparentheses -Wsequence-point -Wreturn-type -Wcast-qual -Wall -Wno-unknown-pragmas –std=gnu99 -Wl,–rpath -Wl,/usr/local/lib -o $i $i.o -L/usr/local/lib -lipopt -lm -ldl -L/usr/lib/gcc/x86_64-redhat-linux/4.1.2 -L/usr/lib/gcc/x86_64-redhat-linux/4.1.2/../../../../lib64 -L/lib/../lib64 -L/usr/lib/../lib64 -lgfortranbegin -lgfortran -lm -lgcc_s -lstdc++ -lm

Dla sun16 linie te, aczkolwiek nieco odmienne, wyglądają równoważnie.

W przypadku kompilacji wersji bez OpenMP usunięta została opcja „-fopenmp". Do kompilacji w środowisku MPI wykorzystano opakowanie gcc zwane **mpicc**, bez dodatkowych zmian w opcjach, a następnie **mpirun**.

Pogrubioną czcionką zostały zaznaczone najlepsze czasy dla każdego rozmiaru zadania.

	20k	200k	700k	2M	4M	16M
1thr	**0.193s**	1.868s	6.048s	17.593s	33.751s	2m41.153s
2thr	0.225s	1.436s	4.261s	13.027s	25.512s	2m13.142s
3thr	0.287s	**1.211s**	**3.821s**	11.209s	22.384s	2m1.495s
4thr	**0.194s**	1.303s	3.933s	**10.795s**	**20.817s**	**1m53.331s**
6thr	0.256s	1.377s	3.879s	10.933s	21.327s	1m58.896s
8thr	0.238s	1.324s	4.498s	12.042s	23.451s	2m13.246s
12thr	0.360s	2.367s	7.547s	23.073s	45.578s	4m46.254s
15thr	0.348s	3.556s	11.881s	33.308s	1m9.505s	7m24.960s
16thr	0.489s	3.734s	13.136s	38.112s	1m9.565s	8m7.849s
20thr	0.500s	4.783s	15.360s	45.676s	1m26.327s	10m21.883s

Tabela 1: Wyniki gs-sumsqr na sun16

	20k	200k	700k	2M	4M	16M
1thr	**0.119s**	0.805s	3.193s	8.750s	17.176s	1m22.022s
2thr	0.253s	**0.617s**	2.190s	6.481s	13.293s	1m2.144s
3thr	0.259s	0.745s	**2.086s**	5.674s	11.688s	55.528s
4thr	0.201s	0.672s	2.098s	**5.440s**	**10.745s**	**52.107s**
6thr	0.123s	0.641s	2.127s	5.532s	10.940s	53.063s
8thr	0.198s	0.683s	2.166s	6.296s	11.610s	59.155s
12thr	0.239s	1.295s	4.028s	12.044s	23.162s	2m9.202s
15thr	0.290s	1.838s	5.924s	16.317s	34.919s	3m30.187s
16thr	0.393s	1.983s	6.260s	18.559s	35.192s	3m42.458s
20thr	0.318s	2.408s	8.185s	22.277s	44.070s	4m6.741s

Tabela 2: Wyniki gs-dqf na sun16

	20k	200k	700k	2M	4M	16M
1thr	0.423s	5.132s	16.101s	54.251s	1m35.606s	8m42.488s
2thr	0.360s	3.067s	10.924s	30.401s	1m9.708s	5m38.460s
3thr	0.393s	**2.448s**	9.043s	25.492s	52.573s	4m39.305s
4thr	**0.291s**	2.716s	8.705s	25.161s	47.877s	3m43.561s
6thr	0.322s	2.455s	**7.940s**	**23.679s**	**43.914s**	**3m29.072s**
8thr	0.327s	2.466s	8.140s	23.745s	51.207s	3m49.799s
12thr	0.391s	3.353s	12.443s	36.377s	1m14.888s	6m16.560s
15thr	0.441s	4.523s	15.900s	49.597s	1m37.320s	8m47.612s
16thr	0.430s	4.532s	17.592s	50.404s	1m41.615s	9m13.859s
20thr	0.661s	6.128s	22.306s	1m1.186s	2m4.351s	12m1.724s

Tabela 3: Wyniki gs-extros na sun16

	20k	200k	700k	2M	4M	16M
1thr	1.979s	46.051s	3m26.530s	3m44.354s	8m36.537s	-
2thr	1.203s	32.795s	1m43.248s	4m26.625s	7m44.119s	-
3thr	1.524s	10.373s	1m4.929s	3m16.615s	6m20.373s	-
4thr	1.359s	8.435s	1m9.455s	**1m43.580s**	5m33.588s	-
6thr	0.833s	7.401s	47.940s	2m10.471s	4m43.945s	-
8thr	0.682s	**5.875s**	**36.556s**	1m57.123s	**1m57.312s**	-
12thr	**0.374s**	6.445s	39.111s	3m3.356s	4m39.670s	-
15thr	0.599s	7.095s	43.725s	2m47.452s	5m56.536s	-
16thr	0.519s	7.186s	39.250s	2m43.574s	4m58.185s	-
20thr	0.806s	8.950s	41.760s	3m10.714s	5m42.490s	-

Tabela 4: Wyniki gs-sumsqr-b na sun16

	20k	200k	700k	2M	4M	16M
1thr	0.974s	12.777s	34.079s	1m33.483s	4m59.737s	**23m59.099s**[1]
2thr	0.552s	7.404s	33.629s	2m38.984s	2m37.801s	43m18.987s[1]
3thr	0.468s	5.937s	24.182s	1m50.620s	**1m46.476s**	25m16.600s[1]
4thr	0.348s	4.418s	19.095s	1m42.585s	3m20.340s	-
6thr	0.320s	2.788s	**12.220s**	1m26.802s	2m12.122s	-
8thr	0.268s	**2.391s**	20.213s	**1m8.339s**	2m20.444s	-
12thr	0.392s	2.686s	23.206s	1m25.259s	2m43.880s	-
15thr	0.305s	3.043s	24.043s	1m32.332s	2m56.402s	-
16thr	0.305s	3.217s	21.924s	1m22.462s	2m49.246s	-
20thr	0.401s	4.117s	24.166s	1m18.845s	3m8.269s	-

Tabela 5: Wyniki gs-dqf-b na sun16

	20k	200k	700k	2M	4M	16M
1thr	22.676s	24.836s	2m41.108s	5m28.437s	14m2.517s	-
2thr	11.891s	15.896s	50.040s	4m29.655s	8m45.854s	-
3thr	11.421s	11.284s	41.756s	1m59.793s	6m50.311s	-
4thr	11.102s	9.247s	35.798s	2m28.652s	5m55.134s	-
6thr	00.838s	7.959s	27.213s	1m45.877s	5m17.647s	-
8thr	00.700s	**6.435s**	**23.866s**	**1m39.975s**	**3m59.269s**	-
12thr	**0.693s**	9.297s	28.758s	2m8.176s	4m39.532s	-
15thr	0.710s	9.200s	32.508s	2m8.775s	5m5.453s	-
16thr	0.713s	8.997s	30.216s	2m6.827s	4m56.170s	-
20thr	0.933s	10.071s	36.544s	2m24.330s	6m49.485s	-

Tabela 6: Wyniki gs-extros-b na sun16

	20k	200k	700k	2M	4M
1thr	0.190s	0.861s	3.394s	10.096s	20.103s
2thr	0.077s	**0.777s**	**2.960s**	**9.283s**	**18.565s**
3thr	**0.072s**	0.820s	3.223s	9.721s	19.469s
4thr	0.077s	0.941s	3.703s	11.352s	23.064s
6thr	0.105s	1.316s	5.013s	14.728s	28.901s
8thr	0.121s	1.534s	5.868s	17.611s	35.474s

Tabela 7: Wyniki gs-sumsqr na kylie

[1]Maszyna była jednocześnie obciążana przez lagrange_r.

	20k	200k	700k	2M	4M
1thr	0.054s	0.450s	1.763s	5.167s	10.358s
2thr	0.042s	**0.385s**	**1.509s**	**4.658s**	**9.374s**
3thr	**0.041s**	0.416s	1.635s	4.967s	9.871s
4thr	0.044s	0.481s	1.893s	5.746s	11.654s
6thr	0.061s	0.678s	2.465s	7.619s	15.063s
8thr	0.067s	0.780s	2.968s	8.716s	17.781s

Tabela 8: Wyniki gs-dqf na kylie

	20k	200k	700k	2M	4M
1thr	0.254s	2.775s	10.873s	31.944s	1m4.829s
2thr	0.161s	**1.886s**	**8.075s**	**24.831s**	**50.257s**
3thr	0.138s	1.933s	8.427s	26.121s	52.609s
4thr	**0.125s**	2.105s	9.022s	28.155s	56.974s
6thr	0.185s	2.666s	10.839s	33.447s	1m9.993s
8thr	0.173s	2.917s	12.115s	38.560s	1m18.408s

Tabela 9: Wyniki gs-extros na kylie

	20k	200k	700k	2M	4M
1thr	1.272s	38.134s	1m48.986s	2m49.033s	6m3.014s
2thr	0.710s	16.363s	1m7.782s	3m56.512s	6m26.142s
3thr	1.002s	8.077s	56.491s	3m3.783s	**5m27.659s**
4thr	0.818s	7.715s	**55.314s**	**2m55.941s**	6m17.057s
6thr	0.710s	8.206s	47.190s	3m22.048s	5m56.775s
8thr	**0.585s**	**7.492s**	42.520s	3m17.055s	5m44.765s

Tabela 10: Wyniki gs-sumsqr-b na kylie

	20k	200k	700k	2M	4M
1thr	0.569s	8.081s	25.414s	**1m13.400s**	3m20.129s
2thr	0.344s	4.962s	24.326s	2m4.228s	2m0.966s
3thr	0.232s	3.936s	20.210s	1m35.338s	**1m44.140s**
4thr	**0.196s**	3.709s	19.845s	1m37.884s	2m52.218s
6thr	0.242s	3.442s	**16.798s**	1m37.133s	2m36.845s
8thr	0.207s	**3.149s**	23.145s	1m33.561s	2m59.906s

Tabela 11: Wyniki gs-dqf-b na kylie

	20k	200k	700k	2M	4M
1thr	1.491s	18.643s	1m52.772s	3m17.693s	9m26.491s
2thr	1.224s	10.704s	43.013s	3m13.283s	7m3.987s
3thr	0.874s	9.340s	38.982s	**2m1.756s**	**6m0.211s**
4thr	**0.731s**	8.997s	**38.877s**	2m40.287s	6m12.476s
6thr	0.903s	9.049s	39.815s	2m29.320s	6m10.818s
8thr	0.781s	**8.747s**	38.992s	2m30.929s	6m14.257s

Tabela 12: Wyniki gs-extros-b na kylie

Ponadto, na kylie dla problemu extros-b, na 4 rdzeniach, uzyskano:

- 200k — 10,192s
- 700k — 40,309s
- 2M — 2m49s

8 Wnioski

Okazuje się, że Q6600 ma nieznacznie (chociaż mierzalnie) większą wydajność (per rdzeń) niż Opterony.

Głównym ograniczniem w rozwiązywaniu dużych zadań jest dostępność pamięci. Na kylie nie jest możliwe uruchomieni zadania 16M, gdyż już zadanie 4M potrzebuje prawie 4GB pamięci. Okazuje się, że zapotrzebowanie na pamięć zależy także od liczby podziałów, a zatem wątków. Przykładowo, zadanie extros 16M w przypadku uruchomienia na 15 wątkach zaalokowoało ponad 25GB („virtual"), w tym 18GB „RSS", a wersja 1 wątkowa zaledwie 3GB i 4GB odpowiednio.

Warto dodać, że takie nadmierne rozwątkowanie powoduje wzrost obciążenia systemu, zajętego synchronizacją alokacji pamięci. Z tego może wynikać brak znaczącej poprawy dla ponad 8 wątków. Aczkolwiek, może to również być spowodowane faktem, że serwer sun16 ma w rzeczywistości 8 procesorów, a więc komunikacja 8 interfejsów do pamięci, a to może ograniczać przepustowość.

Stwierdzono także, że problemy z ograniczeniami, nawet w prostej postaci pudełkowej, znacznie utrudniają obliczenia w trakcie danej poditeracji. Z tego względu na

problemach z ograniczeniami lepiej widoczne bywa przyspieszenie uzyskane przez zrównoleglenie zadania.

Warto zauważyć, że pewne rozmiary danych dla podinteracji są bardziej wydajne niż inne. Możemy czasem zaobserwować swoisty „sweet-spot" dla rozmiaru bloku ok. 500000 zmiennych, jak w gs-sumsqr-b dla 2M i 4M na 4 i 8 wątkach.

Zastanawiające jest, iż czasem (dla małych zadań) szybsze jest wykonanie z większą liczbą podziałów niż dostępna liczba rdzeni. Może to być spowodowane lepszym wykorzystaniem czasu procesora po zakończeniu obliczeń jednego z bloków. Stwierdzono bowiem, że niektóre poditeracje kończą się szybciej niż inne (przy zachowaniu pełnej poprawności wyniku). Wtedy w miejsce takiej poditeracji, w programie nadmiernie rozwątkowanym, wejdzie nowy blok danych. Taka stuacja ma jednak tylko miejsce dla małych zadań. Zaobserwowano także pewne flutuacje wyników, które mogą być spowodowane drobnymi zmianami w obciążeniu maszyny (cron? zewnętrzne zapytania?).

Dalszy wzrost szybkości obliczeń można uzyskać poprzez wykorzystanie specjalistycznej biblioteki algebry liniowej (np. Intel Math Kernel Library), która posiada procedury algebry liniowej (wykorzystywane przez IPOPTa) ręcznie optymalizowane, a przez to szybsze niż standardowa wersja biblioteki napisana w C lub Fortranie. Z tego względu opcje kompilacji programu końcowego (takie jak -O3) nie mają żadnego znaczenia, gdyż program ten zajmuje się tylko przesuwaniem niewielkiej ilości danych między strukturami, a nie prawdziwymi obliczeniami.

MPI jest bardzo wygodnym mechanizmem, oferując przy tym zbliżoną wydajność do wątków realizowanych przez OpenMP.

9 Skrypty

Do testowania i przetwarzania jego wyników konieczne było użycie skryptów, ze względu na znaczną liczbę danych.

Dane zbierano przy pomocy:

```
for k in gs-sumsqr gs-dqf gs-extros gs-sumsqr-b gs-dqf-b \
                                          gs-extros-b
do
   for i in 20160 200160 691200 2000160 4000320 16001280
      do
         for j in 1 2 3 4 6 8 12 15 16 20
            do
               echo "[TESTING... time ./$k $i $j]"
               R=`(time ./$k $i $j) 2>&1 | grep real \
                                    | awk '{print $2}'`
               echo "[DONE $k $i $j $R]"
            done
      done
done
```

W kolejnym etapie dane te zostały poddane obróbce w OpenOffice Calc, poprzez załadowanie do niego wyniku działań poniższych komend.

```
cat dane | grep DONE | awk -F] '{print $1}' \
   | awk '{print $2 " " $3 " " $4 }'

cat dane | grep DONE | awk -F] '{print $1}' \
   | awk '{print $5}' \
   | sed 's/\([0-9].*\)m\([0-9.]*\).*/\1*60+\2/' \
   | awk '{print $1}' | bc | sed 's/\./\,/'
```

Następnie narysowane zostały wykresy.

Tabele LaTeX-owe przygotowano skryptem:

```
for k in gs-sumsqr gs-dqf gs-extros gs-sumsqr-b gs-dqf-b \
                                          gs-extros-b
do
   echo $k
   for i in 1 2 3 4 6 8 12 15 16 20
      do
         cat dane | egrep " $k " | grep DONE \
            | awk -F] '{print $1}' | awk '{print $4 " " $5}' \
            | egrep "^$i " | awk '{printf $2 " & "  }'
         echo
      done
done | sed 's/&\ $/\\\\\\\hline/g'
```

10 Podsumowanie

Projekt zrealizowano w znacznym stopniu pomyślnie. Stwierdzono zauważalne przyspieszenie wersji wielowątkowej ponad sekwenycjną. dochodzące do 6.

Co więcej, znacznie skuteczniej zrównoleglają się zadania trudne. Trudne, czyli intensywne obliczeniowo (takie jak zadania ograniczone), ale nie ogromne pamięciowo. Największe problemy cierpią z powodu znacznych kosztów operacji pamięciowych. Krótkie zadania natomiast, trudno jest przyspieszyć poprzez zrównoleglenie, gdyż narzut związany z komunikacją między wątkami staje się znaczący.

Literatura

[1] Bertsekas D. P., Tsitsiklis J. N.: *Parallel and Distributed Computation: Numerical Methods*, Athena Scientific, Belmont 1989

[2] Grippo L., Sciandrone M.: *On the convergence of the block nonlinear Gauss-Seidel method under convex constraints*, Operations Research Letters, 26, 127–136 (2000)

[3] Patriksson M.: *Cost Approximation Algorithms* w: Encyclopedia of Optimization, February 17, 2000

[4] Yamakawa E., Fukushima M.: *Testing Parallel Variable Transformation*, Computational Optimization and Applications, 13, 253–274 (1999)

[5] *Introduction to IPOPT: A tutorial for downloading, installing, and using IPOPT*, rev. 1585, October 29, 2009

[6] http://publib.boulder.ibm.com/infocenter/comphelp/v7v91/index.jsp?topic=/com.ibm.vacpp7a.doc/getstart/overview/apx_openmp.htm

[7] http://mtzweb.stanford.edu/programs/documentation/ifc/c_ug/linux307.htm

[8] http://www.sun.com/servers/x64/x4600/specs.xml

[9] http://en.wikipedia.org/wiki/List_of_device_bandwidths