1. Cel projektu

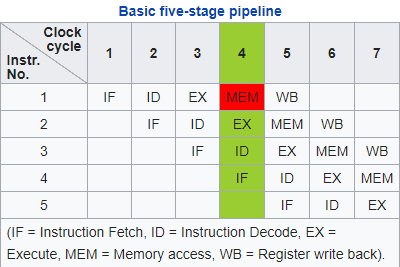
Celem projektu było przetestowanie działania technologii Hyper-threadingu na własnej implementacji wybranego algorytmu.

2. Czym jest Hyper-threading?

Hyper-threading to opracowana przez firmę Intel implementacja techniki simultaneous multi-threading (SMT), która jest obecnie najwyższą formą w etapach rozwoju wielowątkowości. W celu uzyskania odpowiedzi na pytanie, dlaczego ta technologia jest tak rozpowszechniona i chętnie stosowana najłatwiej będzie to osiągnąć poprzez przedstawienie krótkiej historii multi-threadingu. Następnie, aby poznać jego działanie zostanie opisana architektura procesora IA-32, która bezpośrednio odpowiada za działanie Hyper-threadingu. Na sam koniec będzie pokazanych kilka reguł programistycznych w celu wykorzystania zalet tej technologii do maksimum.

Podstawową rzeczą, z którą spotyka się na codzień użytkownik komputera są programy, w technice nazywanych procesami. Można je uruchamiać wiele na raz w ilościach ograniczonych przez pojemność pamięci RAM. W celu wykonania na nich operacji, muszą zostać przetworzone przez procesor i wykluczając częstotliwość taktowania pamięci, to od niego zależy efektywność działania aplikacji. Na wydajność procesora wpływa wiele czynników, nad którymi od lat pracują inżynierowie i programiści. Są to między innymi: częstotliwość taktowania zegara a więc także zasilanie, system chłodzący i wytrzymałość materiałów, z których jest zbudowany. Najtrudniejszą rzeczą do opracowania jest logika działania poszczególnych elementów procesora, organizacja ich efektywnej współpracy, przemieszczania danych i wykonywania na nich operacji. Naturalnym kierunkiem rozwoju technologii w celu zwiększenia szybkości wykonywania obliczeń jest zrównoleglenie operacji (parallelization). Do lat 90. było to osiągane poprzez rozwiązania hardwarowe w samym CPU (Central Processing Unit) i nazywane jest „instruction-level parallelism”, czyli równoległość na poziomie instrukcji, a procesor, który to realizuje nazywa się super-skalarnym. W tym dokumencie jest wymienionych kilka z nich, na których opiera się cały Hyper-threading.

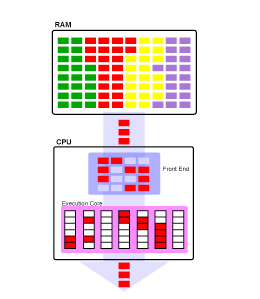
Wspomniane wcześniej programy to tak naprawdę instrukcje, zapisywane w czasie ich kompilacji. Są one przechowywane w pamięci RAM i czekają na swoją kolej do bycia zrealizowanym w jednostce. Każda z instrukcji wykorzystuje różne zasoby procesora, a wykonywanie jedna po drugiej powoduje ich marnowanie. Problem został rozwiązany poprzez zastosowanie techniki „pipeline”, która została przedsiawiona na rysunku:

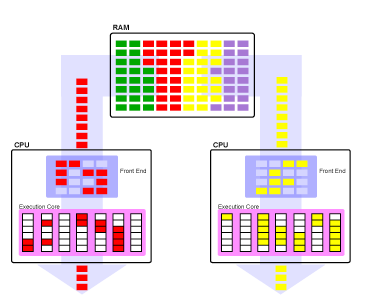


IF - pobranie instrukcji, ID – dekodowanie instrukcji, EX – operacja arytmetyczno-logiczna, MEM – dostęp do pamięci, WB – zapisanie wyniku do rejestru. W zaznaczonym na zielono 4. cyklu procesora 1. instrukcja uzyskuje dostęp do pamięci, a 5. jeszcze się nie rozpoczęła.

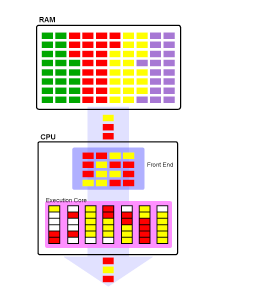
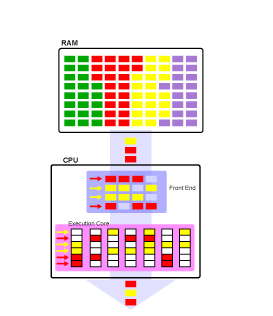
Jest to przykładowy prosty podział RISC. W procesorach Intel Xeon pipeline jest długości 31, w późniejszej części zostanie opisany dokładnie. Zasada działania polega na tym: kiedy pierwsza instrukcja została już pobrana, zasób procesora odpowiedzialny za tę operację jest już wolny i dlatego można przyjąć kolejną. Z różnych przyczyn, np. zależność jednej instrukcji od drugiej czy oczekiwanie na pobranie danych z pamięcimogą się pojawić tzw. „bańki” (bubbles), czyli momenty bezczynności zasobów procesora. W praktyce pojawiają się niemal nie ustannie i stanowią nawet 50% pipeline i stanowiło bezpośrednią przyczynę do zainteresowania się wielowątkowością. Innymi rozwiązaniami z zakresu ILP, które też pomagają w eliminacji tych baniek, stosuje się wiele mechanizmów, które zapobiegają częstemu sięgania do pamięci RAM, np. branch prediction (przewidywanie gałęzi), czyli pobieranie danych dla takich warunków, gdzie prawdopodobieństwo wykonania warunku „if” jest większe albo często się wykonuje w pętli. Inne rozwiązanie to „register renaming”, czyli zmiana nazw rejestrów, gdzie np. zamiast zgodnie z instrukcją obliczyć pierwsze działanie w rejestrze eax i potem w tym samym kolejne, obliczyć oba równolegle jedno w eax a drugie w ebx. Oczywiście obie instrukcje muszą być niezależne, gdyż w przeciwnym wypadku oba oczekiwałyby na siebie i doprowadziłoby to sytuacji „deadlock” i zawieszenia pracy systemu.

Należy wprowadzić termin „wątek” (thread). Wątek to jest podproces realizowany w ramach programu. Każdy proces składa się z conajmniej jednego wątku. W zależności od kodu i zamierzenia programisty, mogą realizować różne funkcje na różnych danych. Przykładem może być Microsoft Word, który ma jeden wątek dla ciągłego czytania znaków klawiatury, a drugi do zapisywania ciągów znaków w RAM. To daje wiekszą elastyczność w pisaniu programu, możliwość zrównoleglenia operacji a także lepszego zarządzania dużymi aplikacjami przez procesor.

Kiedy w latach 90. doprowadzono ILP niemal do perfekcji, zaczęto się zastanawiać nad innymi sposobami paralelizacji działań. Probemem było niewykorzystywanie wszystkich zasobów procesora, czyli wspomniane już wcześniej bańki. Na samym początku procesory były jednordzeniowe i mogły realizować tylko jeden wątek na raz, który w zależności od kodu, w różny sposób wykorzystywał możliwości CPU. Należało przy okazji rozwiązać problem, w jaki sposób użytkownik może uruchamiać wiele aplikacji na raz. Otóż to zadanie należy do systemu operacyjnego, który daje każdemu wątkowi określoną ilość czasu „time slice”. Kiedy czas się skończy, OS wysyłał sygnał przerwania, a cały bieżący stan programu (context), czyli stan rejestrów, flag, nr instrukcji itd. są zapisywane, a działanie zaczyna kolejny proces. Te przedziały czasu są na tyle małe, że użytkownik nie dostrzega tego, że są wykonywane sekwencyjnie. Tą sytuację przedstawia rysunek: 

Na rysunku została w skrócie przedstawiona zasada działania procesora jednordzeniowego. Jest on w stanie przetworzyć tylko jeden wątek na raz. Front Endem określone zostały określone operacje przygotowywujące instrukcje: pobranie, dekodowanie, zmiana kolejności itd., wykonywane w technice pipeline. Execution Core to kilka-kilkanaście ALU. Na rysunku są załadowane 4 programy do RAM a w tym momencie uruchomiony jest program czerwony, napisany w ten sposób, że spośród możliwych 4 instrukcji do wykonania w najlepszym wypadku wykonały się 3 na raz. Jak można zauważyć, obie części CPU nie są wykorzystywane w 100%. Zanim jednak rozwiązano ten problem, w międzyczasie pojawiła się architektura dwurdzeniowa. 

Tym razem mogą się wykonywać dwa wątki na raz. Poza tym to zaoszczędza dodatkowy czas związany z opróżnianiem CPU dla następnego wątku w związku z time-slice. Każdy zapis stanu i odczyt wymaga użycia pamięci. Dodatkowy rdzeń zmniejsza częstotliwość przełączeń pomiędzy wątkami. Problemem jednak jest to, że dwukrotne zwiększenie ilości rdzeni dubluje też marnowanie mocy obliczeniowej związanej z ich nieefektywnym wykorzystaniem. Dlatego też opracowano technikę Superthreaded CPU.

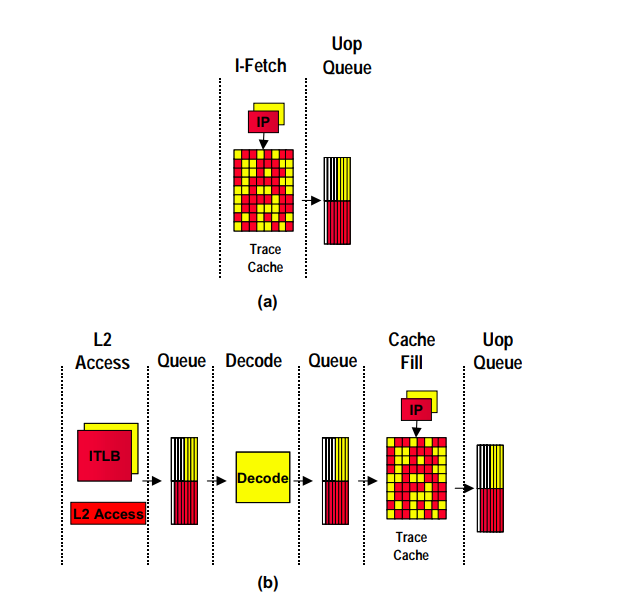
W poprzednich przypadkach time-slice określał czas życia jednego wątku, który po zakończeniu przydziału był zapisywany z powrotem do RAM, nawet jeśli się nie wykonał. W superthreadingu załadowany fragment instrukcji wąteu pozostaje już cały czas wewnątrz aż do jego wykonaniai dopiero po tym następuje zapis danych. Do systemu operacyjnego należy teraz wyłącznie sprawiedliwy rozdział wątków, które znajdą się we Front Endzie. To sprawia, że jest zaoszczędzone mnóstwo czasu związanego z ciągłym odczytywaniem i zapisem danych. Należy jednak zwrócić uwagę, że w pipeline mogą znaleźć się instrukcje związane tylko z jednym wątkiem, a co za tym idzie też i w ALU. Ostateczne rozwiązanie tej kwesti przyniósł dopiero Hyper-threading.

Na rysunku powyżej pokazana jest idealna sytuacja użycia tej technologii. Pipeline jest wypełniony maksymalnie i niewiele ALU pozostaje nieużytych. Diagramy, które pojawiały się do tej pory pokazywały ogólną ideę tego, co dzieje się w środku CPU. Jednak implementacja HT jest bardziej skomplikowana i by w pełni zrozumieć, w jaki sposób procesor jest w stanie rzeczywiście efektywnie wykorzystywać wszystkie zasoby procesora w pipelinie, zarządzać pamięcią itd., należy przyjrzeć się jeszcze bliżej jego architekturze.

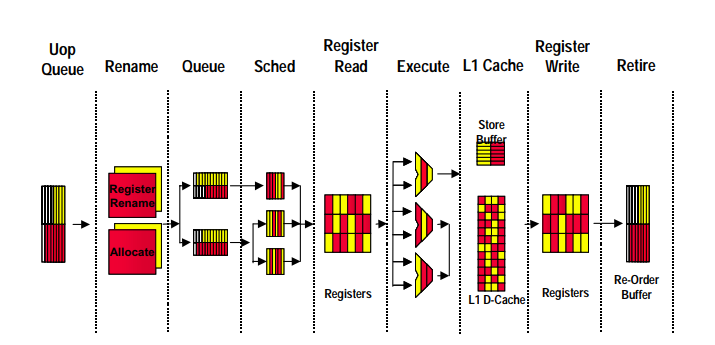
Technologia HT sprawia, że jeden rdzeń fizyczny jest widziana przez OS jako dwa logiczne (wirtualne), co ułatwia pracę. Jest to możliwe dzięki powieleniu kilku elementów procesora. Jednak nie jest celem tworzenie wielu rdzeni, tak więc niektóre są wspólne.

Elementy powielone:

* Rejestry ogólnego przeznaczenia (eax,ebx...)
* Rejestry segmentowe (cs,ds,ss...)
* Rejestry flagowe i wskaźnika instrukcji
* Rejestry FPU
* Rejestry MMX i XMM
* Wiele innych rejestrów

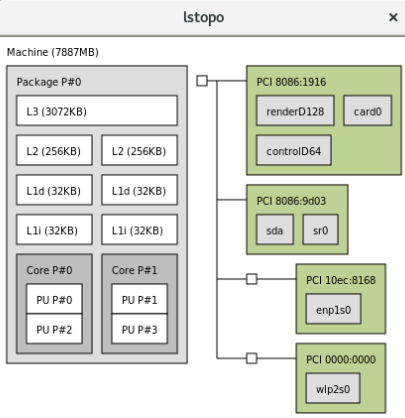
Dzięki temu każdy rdzeń logiczny ma na tyle dużą samodzielność, by kontrolować przepływ danych. Z części wspólnych wciąż pozostają pamięć Cache oraz jednostka wykonywawcza. Zostanie teraz pokazane w jaki sposób krok po kroku są przetwarzane informacje w procesorze. Jest to dokładne objaśnienie pipelinu w Hyper-threadingu, jaka za nim stoi architektura i jak to jest zorganizowane. Najpierw Front End:

Podpunkty a) oraz b) rozróżniają sytuacje, kiedy dana instrukcja znalazła się w pamięci cache L1 czy też należy sięgnąć do L2. Najpierw pierwszy przypadek: Trace Cache (TC) to inna nazwa L1, która przechowuje zdekodowane instrukcje wielu wątków, dla ułatwienia są tu tylko dwa: żółty i czerwony. IP to instruction pointer, czyli kod instrukcji do wykonania. Uop queue to kolejka mikrooperacji, rozdzielonych zgodnie z wątkami. Jeżeli jednak wymagana instrukcja nie została znaleziona w L1, wtedy jest ona szukana w pamięci L2. Ze względu na segmentację pamięci, która pozwala organizować pamięć tak, by różne rdzenie procesora widziały tylko określone części pamięci, wymagane jest tłumaczenie adresów wirtualnych na fizyczne, a takie są stosowane w pamięci L2 oraz RAM. Następnie instrukcje należy zdekodować, jeżeli są skomplikowane uruchamiana jest pamięć ROM mikrokodu. Można zauważyć, że podczas pobierania instrukcji z L2, muszą one przejść przez L1. Podobnie też jest z zapisem, gdzie zapis do L1 automatycznie jest też wykonywany w L2. Oba procesory logiczne mają równorzędny dostęp do TC, przy czym tylko jeden na raz. Jeżeli tylko jeden pracuje, ma pełen dostęp do L1. Instrukcje są alokowane dynamicznie zgodnie z algorytmem least-recently-used (LRU). Kiedy już są gotowe mikrooperacje, nadszedł moment na część zwaną „out-of-order excecution engine”, która składa się z alokacji, register renaming, scheduling i wykonywania.



Zadaniem alokatora jest przydzielenie dla każdej z mikrooperacji konkretnego zasobu procesora potrzebnego do wykonywania, np. rejestru. Do wyboru jest 128 rejestrów liczb całkowitych i zmiennoprzecinkowych, 126 wejść re-order buffer czy też 24 wejścia store-buffer. Register rename było opisywane wcześniej: zapewnia paralelizację wykorzystywanych rejestrów. Następnie mikrooperacje trafiają do kolejki, gdzie będą oczekiwały na działanie najważniejszego elementu Hyper-threadingu – schedulera. 5 schedulerów układa kolejność różnych typów mikrooperacji. To one wykrywają zależności pomiędzy wątkami i instrukcjami. Zapewniają zrównoleglenie operacji, odkładają na później operacje, które musiałyby oczekiwać na wynik innych. Nie ma znaczenia, do jakiego wątku należy dana instrukcja. Aby jednak uniknąć deadlocka, każdy procesor logiczny ma ograniczoną liczbę wejść do schedulerów. Dopiero wtedy mikrooperacje trafiają do rejestrów i oczekują na obliczenie w ALU. Wyniki następnie trafiają do store buffera, który ma za zadanie eliminować wstrzymania związane z brakiem dostępu do pamięci cache. Potem mikrooperacje są zapisywane do rejestrów a potem jest odtwarzana ich kolejność.

Do tej pory były przedstawiane sprzętowe rozwiązania, które firma Intel implementuje już na etapie produkcji procesorów, dzięki czemu programiści na codzień nie muszą zastanawiać się, w jaki sposób to działa. Najważniejszą rzeczą, którą należy pamiętać jest to, że widząc 2 procesory z włączonym HT tak naprawdę jest to tylko 1, z efektywniejszym zarządzaniem zasobami. We współczesnym świecie, kiedy programowanie wielowątkowe jest już szeroko przyjętą normą, na programiście spoczywa efektywne wykorzystanie zasobów procesora. Procesor i mechanizmy optymalizujące program nie zawsze są w stanie przewidzieć zachowania programu, co w najgorszym przypadku prowadzi to serializacji elementów programu tak, gdzie nie jest to konieczne. W ciągu lat pojawiło się wiele rad dotyczących efektywnego programowania.

Przede wszystkim należy sprawdzić, czy technologia HT jest włączona oraz jakie są identyfikatory procesorów. Są one już odgórnie przypisane dla każdego z nich. Na systemie Linux, na którym były wykonywane testy, można to bardzo łatwo sprawdzić, instalując program Hwloc, a następnie wywołując komendę lstopo:

Na powyższym rysunku można sprawdzić wygląd architektury komputera. Na tym można zauważyć, że są dwa rdzenie fizyczne, a każdy wewnątrz ma po 2 logiczne. W tym przypadku są oznaczone numerami 0,2 oraz 1,3. Przy okazji można sprawdzić, w jaki sposób pamięć cache wpływa na szybkość programu.

Innym ważnym elementem jest efektywne programowanie wielowątkowe. Oczywista jest potrzeba umiejętności wyszukiwania elementów programu, które można wykonać równolegle. Ważna jest również synchronizacja wątków, jeżeli są w jakiś sposób zależne. Na programiście spoczywa ustawienie odpowiednich blokad wątków. Służą one do sytuacji, gdzie jeden wątek czeka na drugi. Moment czekania tak naprawdę zużywa w największym stopniu zasoby procesora, gdyż cały czas sprawdza, czy już może się wykonać. Programista można wybrać dwa sposoby, by zaoszczędzić zasoby: ustawić pętlę z uśpieniem wątku lub użyć specjalnych klas, np. mutex. Inną dobrą praktyką jest zwiększyć lokalność zmiennych, ponieważ zazwyczaj są też wtedy bliżej fizycznie, co zaoszczędza użycie magistrali danych. Należy też pamiętać o tym, że oba procesory logiczne używają tej samej pamięci cache. Dlatego najlepszą możliwą sytuacją dla HT jest taka, gdzie są 2 różne zadania, gdzie każde wykorzystuje inne zasoby procesora ale oba pracują na tych samych danych. W przypadku dostępu do cache może zdarzyć się sytuacja, że oba wątki będą cały czas pobierać różne dane i nawzajem sobie przeszkadzać. Nazywa się to „trashing” i polega to na tym, że jeden procesor logiczny, np. CPU 0 pobiera dane z RAM. Następnie drugi procesor, CPU 1 sprawdza, czy jego dane są w cache. Jeśli to są zupełnie inne dane, niż te na których pracuje CPU 0, otrzymuje informacje cache miss i wysyła żądanie pobrania wartości z RAM. Potem CPU 0 ma cache miss, pobiera z RAM i tak cały czas. Pobranie wartości z RAM to sytuacja, którą wszyscy starają się unikać, gdyż jest to marnowanie zasobów procesora, gdyż pobranie może wynosić nawet 200 cyklów procesora. W takiej sytuacji najlepiej przydzielić zadanie do innego procesora fizycznego, o ile jest dostępny. W przypadku tych samych danych HT jest nieco lepszy, ponieważ odciążą magistralę łączącą oba rdzenie i nie dochodzi do sytuacji, gdzie Store Buffer, który ma obsługiwać przenoszenie danych pomiędzy oboma cache, zostanie przepełniony i spowoduje wstrzymanie na kilka cykli pracy procesora.

3. Program

W tym rozdziale zostanie opisany program, który został użyty do testowania Hyper-threadingu. Składa się z jednego pliku main.cpp, zawierającego dwie funkcje: main() i funkcja(). Program używa POSIX, co oznacza, że można go skompilować tylko na systemie operacyjnym LINUX. Wybranym algorytmem jest mnożenie macierzy.

//*main.cpp*

#include<iostream>

#include<cstdlib>

#include<ctime>

#include<chrono>

#include<sys/resource.h>

#include<pthread.h>

#include<thread>

#include<algorithm>

using namespace std;

void funkcja()

{

const int x = 1000, y = 1000, z = 1000;

//rezerwacja pamięci na macierze

int p[x][y];

int r[z][y];

int s[x][z];

//wypełnienie czynnikow losowymi liczbami

for (int i = 0; i < x; i++)

for (int j = 0; j < y; j++)

p[i][j] = rand() % 31;

for (int i = 0; i < z; i++)

for (int j = 0; j < y; j++)

r[i][j] = rand() % 31;

//obliczenie iloczynu

std::thread thread1,thread2;

thread1=std::thread([&p,&r,&s,x,y,z](){

auto start = std::chrono::system\_clock::now();

int t = 0;

for (int i = 0; i < x; i++)

for (int j = 0; j < z; j++)

{

for (int k = y/2; k < y; k++)

t += p[i][k] \* r[j][k];

s[i][j] = t;

t = 0;

}

auto stop = std::chrono::system\_clock::now();

std::chrono::duration<double> dur = stop - start;

std::cout<<dur.count()<<std::endl;

});

thread2=std::thread([&p,&r,&s,x,y,z]{

auto start = std::chrono::system\_clock::now();

int t = 0;

for (int i = 0; i < x; i++)

for (int j = 0; j < z; j++)

{

for (int k = 0; k < y/2; k++)

{

t += p[i][k] \* r[j][k];

}

s[i][j] = t;

t = 0;

}

auto stop = std::chrono::system\_clock::now();

std::chrono::duration<double> dur = stop - start;

std::cout<<dur.count()<<std::endl;

});

cpu\_set\_t cpuset; //zmienna (maska) przechowująca aktywowane rdzenie

CPU\_ZERO(&cpuset); //wyzerowanie maski

CPU\_SET(0, &cpuset); //tutaj sie dodaje rdzeń, u mnie od 0 do 3, dodaję //rdzen nr 0

CPU\_SET(2, &cpuset);

int er = pthread\_setaffinity\_np(thread1.native\_handle(),

1, &cpuset); //aktywuje wątek

int er2 = pthread\_setaffinity\_np(thread2.native\_handle(),

1, &cpuset);

if (er != 0) {

std::cerr << "Error calling pthread\_setaffinity\_np: " << er << "\n"; //błąd jesli nie aktywowało

}

if (er2 != 0) {

std::cerr << "Error calling pthread\_setaffinity\_np: " << er2 << "\n";

}

thread1.join();

thread2.join();

}

int main()

{

srand(time(NULL));

const rlim\_t kStackSize = 110110110000;

struct rlimit rl;

int result;

result = getrlimit(RLIMIT\_STACK, &rl);

if (result == 0)

{

if (rl.rlim\_cur < kStackSize)

{

rl.rlim\_cur = kStackSize;

result = setrlimit(RLIMIT\_STACK, &rl);

cout<<result<<endl;

}

}

funkcja();

}

Przechodząc program po kolei: na samym początku jest ustawiany seed dla funkcji losującej, która będzie generowała liczby dla macierzy. Następnie cały kod, aż do wywołania funkcji, dotyczy rezerwacji stosu – obszaru pamięci używanego między innymi przez tablicę statyczną. Ze względu na niekiedy używanych w testach jej olbrzymie rozmiary, rezerwacja większej pamięci była konieczna. W tym przypadku jest rezerwowane 110110110000 bajtów, czyli ok. 100MB. Następnie wywoływana jest główna część tego programu. Zmienne x,y,z to wymiary macierzy gotowych do przemnożenia. Są wypełniane liczbami od 0 do 30. Następnie obliczenie iloczynu jest rodzielone pomiędzy wątki. W celu zmniejszenia odległości pomiędzy zmiennymi, każdy wątek otrzymuje te same wiersze w macierzach, które są pomiędzy nie rozdzielane. Macierze są mnożone wierszowo, tzn. jedna z nich została transponowana. Jest to związane z budową pamięci, gdzie dostęp do jej elementów jest wierszowy. Transponowanie macierzy zwiększyło wydajność obliczeń o ok. 70%. Każdy wątek ma swój licznik czasu, a więc jest tyle pomiarów ile wątków. Po części wątkowej, jest sterowanie przydziałem wątków do konkretnych rdzeni procesora. Cpu\_set\_t to zmienna (maska), która zachowuje się jak tablica typu bool. Jeśli na komputerze są 4 procesory logiczne, to będzie miała 4 elementy. Wybranie procesora oznacza wstawienie w odpowiednie miejsce jedynki i tak np. chcąć wybrać procesory (licząc od 0) 1 i 3, zmienna przyjmie postać 1010. Modyfikacja tych wartości odbywa się za pomocą specjalnych metod, które są widoczne dalej. CPU\_ZERO ustawia wszystkie bity w masce na 0. CPU\_SET to ustawienie wybranego bitu na 1, co jest równoznaczne z wyborem rdzenia. W przykładzie zostają wybrane procesory 0,2. Numery procesorów są tożsame z tymi, na których pracuje Linux, a więc z tym co pokazuje komenda lstopo, czy też htop pokazujący podgląd użycia rdzeni. Samo ustawienie maski jednak nie przydziela wątków, czyni to dopiero funkcja pthread\_setaffinity\_np, gdzie parametrami są: identyfikator wątku, na ilu rdzeniach ma się znaleźć i wg. jakiej maski ma to przydzielić. Mając maskę w przykładzie 0101, pthread przydziela po kolei wątki. System operacyjny dba o to, że jeżeli jeden rdzeń został już zajęty, kolejny wątek zostanie przydzielony do innego. Można oczywiście ustawić maskę jeszcze raz po pierwszym wywołaniu pthread\_setaffinity i potem wywołać tę funkcję jeszcze raz dla innej maski. Funkcja zwraca 0, jeśli powiodła się operacja przydzielenia wątku. Na sam koniec wątki są synchronizowane tak, aby główny wątek wykonywania, czyli main() zaczekał na wykonanie obu wątków. W przypadku tego programu, mnożenie macierzy zostało podzielone pomiędzy dwa wątki i oba zostały zredystrybuowane na rdzenie 0,2. Patrząc na rysunk uzyskane z lstopo, można dojść do wniosku, że wykorzystywana jest technologia Hyper-threading. Wątki można dodawać, wystarczy zmienić zakresy wierszy macierzach oraz przydzielić go do konkretnego procesora. Jest tutaj pełna dowolność i kilka wątków można przydzielić do jednego procesora, co zostało wykorzystane w testach.

5. Wnioski

Na podstawie wykresów można dojść do wniosków dotyczących tego, czy Hyper-threading zawsze zwiększa wydajność wykonywania obliczeń. Analiza zostanie rozpoczęta porównując dane w obrębie konkretnej ilości wątków. Wynik wykonania ostatniego wątku, po dodatniu kilkudziesięciu nanosekund jest też czasem wykonania całego programu.

1 wątek

Stanowi on defacto próbę kontrolną, dzięki któremu można porównać ogólnie zalety wielowątkowości i wykorzystania wielu rdzeni do obliczeń. W tym przypadku rozpiętość czasowa wynosi od 3,61\*10-6s do 87,6s.

2 wątki

Są 3 przypadki: powtórzenie poprzedniego punktu, różne rdzenie fizyczne oraz wykorzystanie Hyper-threadingu. Nie dziwi fakt, że oba zadania zostały wykonane w identycznym czasie. Drugi przypadek udowadnia, że praca rzeczywiście została sparalelizowana a zamiast 90s wynik uzyskano w 45s. Porównując 1. przypadek i 3. nasuwa się wniosek, że Hyper-threading w ogóle nie poprawił wydajności.

3 wątki

Porównując przypadki 1 i 2 , powtarza się poprzednia sytuacja, tylko dodatkowo występuje wątek 3. Dlatego też nie dziwi fakt, że HT nie sprawuje się dobrze. Czas wynosi teraz 60s, ponieważ jest mniejsza porca macierzy do wykonania. Przypadek 3 potwiedza poprawność obliczeń, gdyż dla jednego rdzenia niezależnie od wątków (które nie inicjalizują się długo) czas musi wyjść taki sam. W przypadku czwartym, przesunięcie wątku 3. z innego rdzenia na ten sam spowodował ogólny spadek wydajności. Co ciekawe wątki 1 i 2 są przez procesor traktowane jako jeden z większą ilością instrukcji. Zachowując zasadę, że każdy procesor logiczny ma tyle samo zasobów, krótszy wątek 3 mógł się wykonać szybciej.

4 wątki

Występujące tutaj przypadki są de facto kombinacjami poprzednich sytuacji. Dla pojedynczych rdzeni fizycznych można porównać działanie Hyper-threadingu, czyli są to 1 i 2 wątek w przypadkach 1 i 3 kontra przypadek 4 oraz wątki 3 i 4 kontra przypadek 3 i 4, dają te same rezultaty, oczywiście dla zmniejszonych od poprzednich ilości wątków czasów. Przypadek drugi jest powtórzeniem czwartego z 3 wątków, gdzie 2 wątki w programie są traktowane jako jeden w procesorze logicznym, a watek na osobnym rdzeniu wygrywa. Przypadki 6, 7 i 8 jasno pokazują potrzebę zrównoleglania operacji. Przy okazji też kolejny raz widać brak wzrostu wydajności po zastosowaniu HT.

Objaśnienie wyników

Z otrzymanych wyników można dojść do wniosku, że Hyper-threading może być chybionym pomysłem, który nie wnosi nic pożytecznego. Jednakże producent zastrzegał, że wydajność w zależności od wielu czynników może się wahać od spadku wydajności aż do wzrostu do 30%. Aby zrozumieć otrzymane rezultaty należy jeszcze raz zwrócić się do schematu działania rdzenia z włączoną technologią HT. Przywołując ogólne działanie: kiedy instrukcja jednego wątku nie wykorzystuje całego pipelinu, inny wątek może uzupełnić braki (bańki). Przydaje się to w sytuacjach, kiedy np. jakiś wątek wykorzystuje 30% mocy obliczeniowej, więc inny może użyć reszty. Jednak w przypadku mnożenia macierzy sytuacja się zmienia, ponieważ jest on na tyle efektywny, że już pojedynczy wątek wykorzystuje całe 100%. Pipeline jest już zapełniony, a także wszystkie jednostki wykonawcze. Dlatego dodawanie kolejnego wątku do rdzenia fizycznego powoduje tak naprawdę zapychanie pękającego już w szwach rdzenia. Tak naprawdę tylko przeszkadza, gdyż zanieczyszcza on cache swoimi danymi. Dlatego dodawanie kolejnego wątku do rdzenia tak naprawdę zmusza do sekwencyjnego wykonywania obliczeń, dlatego nie ma różnicy między HT a pojedynczym rdzeniem. Drobne lepsze czasy są powodem tego, że mimo wszystko co jakiś czas takie bańki się pojawiają, jednak nie mogło ich być wiele, skoro dla macierzy o rozmiarze 3000 HT przyspieszył obliczenia o zaledwie sekundę. Poza tym optymalizacja pod względem lokalności zmiennych, a w związku z tym także operacje na pamięci cache była dobra, gdyż nie zanotowano spadków wydajności. Maksymalna długość wiersza, z której korzystały wątki wynosiła 3000, a więc kilka-kilkanaście wierszy z macierzy mieściło się w conajmniej L3. Drobne różnice w czasach pomiędzy wątkami w przypadkach symetrycznych wynikają z wielu błędów pomiarowych. System operacyjny co prawda oczyszczał rdzenie z innych programów (np. htop, który zużywał 0,7% rdzenia czy też innych systemów diagnozujących. Poza tym można też było zauważyć, że nie bierze pod uwagę HT, a więc jeżeli zostało uruchomione mnożenie na procesorze 0, to rzeczywiście inne programy zostały stamtąd zabrane ale na procesor 2, który jest w HT z 0, a jak zostało to wyżej opisane powoduje to spadek wydajności. Poza tym inicjalizacja kolejnych wątków zabierała w niektórych przypadkach nawet 4 sekundy, gdzie powodem była między innymi załadowanie pamięci RAM. Najtrudniej oszacować, jak spadła wydajność w 1. przypadku w 4 wątkach, gdyż OS nie miał już gdzie zabrać programów. Reasumując: paralelizacja wątków jest niemal zawsze dobra, dopóki tylko ma sens, a więc jeżeli są 4 procesory logiczne, to podział na 4 jest najlepszym rozwiązaniem. Jeżeli wątków jest mniej, najlepiej je rozdzielać pomiędzy różne rdzenie fizyczne. HT jest dobry, jeżeli jest stworzony odpowiedni algorytm – numeryczne wykorzystujące do pełna procesor nie nadają się. Jeżeli dodatkowo mają zaśmiecać pamięć cache, lepiej zastanowić się nad przepisaniem programu na sekwencyjny.