Obliczenia równolegle w architekturze SMP

- **Przykład: przeglądanie stron internetowych przez browser zajmuje dość sporo czasu. Sposoby przyspieszenia:**
  - **przewidujemy, który strony będą następnymi, i przygotujemy zadanie na pobieranie z wyprzedzeniem (prefetching)**
  - **uruchamiamy kilka przeglądarek jednocześnie, i dokpóki jedna strona się ładuje, pracujemy z inną stroną (multithreading)**

- **Multithreading jest używany dla ukrycia zwłoki (hiding of latency). Rozważymy przykład: użytkownik drukuje duży dokument. W trybie jednowątkowym CPU będzie czekał na następne zadanie dopóki drukowanie się nie skończy. W trybie wielowątkowym można stworzyć osobny wątek, który weźmie na siebie drukowanie dokumentu, a CPU może się przełączyć do wykonania innego zadania. Analogiczna sytuacje powstaje przy zapisu / odczycie na /z dysk(u) dużego pliku.**

- **Wielowątkowość może być zrealizowana na podstawie bibliotek wielowątkowych (Win 32 platform – SDK, Posix Thread library), Intel ThreaD Building Block (TBB) oraz na podstawie dyrektyw kompilatora (OpenMP).**

- **Oprogramowanie na podstawie bibliotek wielowątkowych wymaga od programisty ręcznego tworzenia wątków, rzutowania na nich zadań równoległych, wprowadzenia synchronizacji na dolnym poziomie, jawnego wyznaczenia klas pamięci dla danych (stos wątku, Thread Local Storage, …), przyklejania wątków do fizycznych procesorów, zawieszania i obudzenia wątków, zmiany ich priorytetów, zniszczenia wątków i wielu innych rzeczy. Taki sposób oprogramowania nadaje możliwość napisania elastycznego, wysoko wydajnego kodu.**

- **Z innej strony bezpośrednie zastosowanie bibliotek wielowątkowych istotnie komplikuje kod programu w stosunku do programu sekwencyjnego. Zadania równoległe muszą być umieszczone w funkcjach wątków, które będą rzutowane na wątki. Trzeba przygotowywać specjalne struktury danych dla przekazania danych funkcjom wątków.**

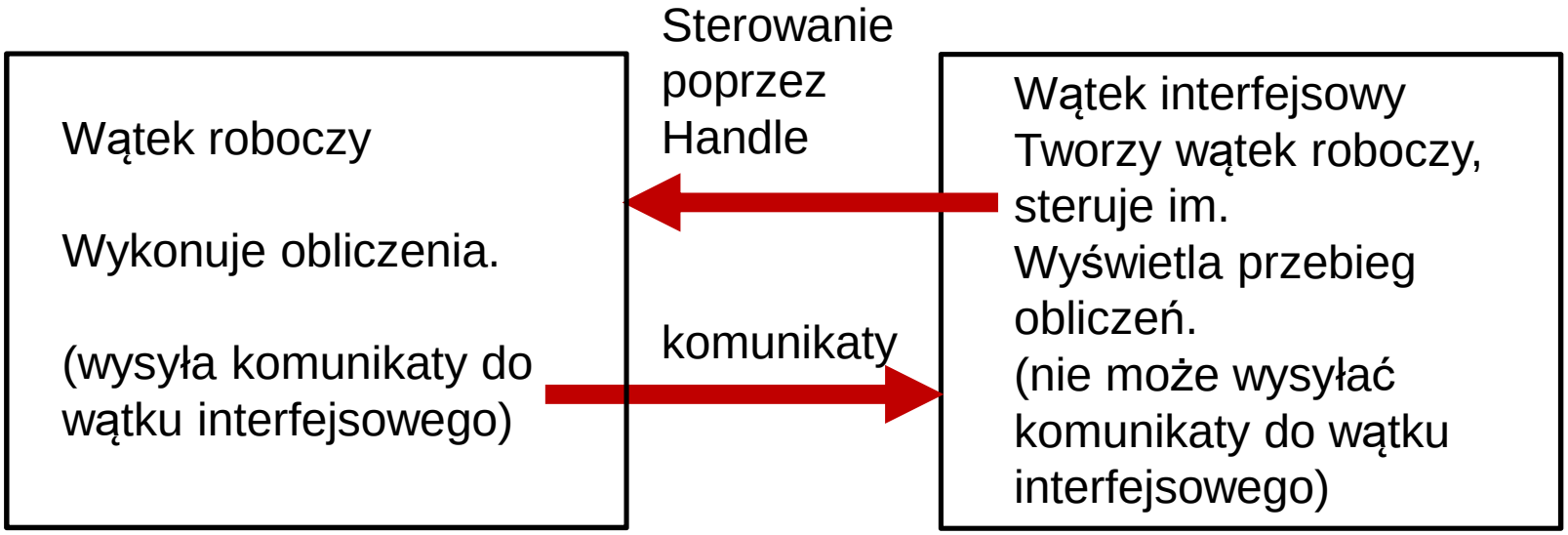
- **Oprogramowanie równoległe na podstawie dyrektyw kompilatora (OpenMP) nadaje możliwość ukryć wielu szczegółów i uniknąć w taki sposób komplikacji kodu, który są niezbędne przy użyci oprogramowania jawnego (na podstawie bibliotek wielowątkowych). Dyrektywy kompilatora robią kod przenośnym na różne systemy operacyjne, a jeśli kompilator nie wspiera OpenMP, to po prostu te dyrektywy pozostają pominięte.**

- **OpenMP nie nadaje możliwości jawnie zmieniać priorytety wątków, przyklejać wątki do procesorów fizycznych. OpenMP jest modelem wielowątkowym, który nadaje możliwość zrealizować tylko połączenie widłowe (fork-joint) pomiędzy wątkami typu master-worker. To oznacza, że zakres zastosowania OpenMP jest ograniczony w stosunku do jawnego oprogramowania wielowątkowego.**

- **Typy zrównoleglenia: zrównoleglenie funkcjonalne (zrównoleglenie zadań) - różne wątki wykonują różne rozkazy, algorytmy, zadania, i zrównoleglenie danych - różne wątki wykonują ten samy fragment kodu – zestaw rozkazów.**

- **Typowym przykładem zrównoleglenia zadań jest procesor tekstowy. Takie zadania, jak sprawdzenie pisowni, zapis czas od czasu, wydrukowanie dokumentu nie przerywają wprowadzenia nowych symbolów z klawiatury i nie przeszkadzają ich wyświetlaniu na monitorze, więc mogą być zrealizowane jako niezależne zadania funkcjonalne na podstawie wielowątkowości. Dla takich celów zastosowują oprogramowanie na podstawie bibliotek wielowątkowych.**

➢ **Drygi typowy przykład zrównoleglenia zadań – dwa wątki, jeden z których interfejsowy, a drugi – roboczy (obliczeniowy).**



Żeby przerwać obliczenia, wątek interfejsowy:

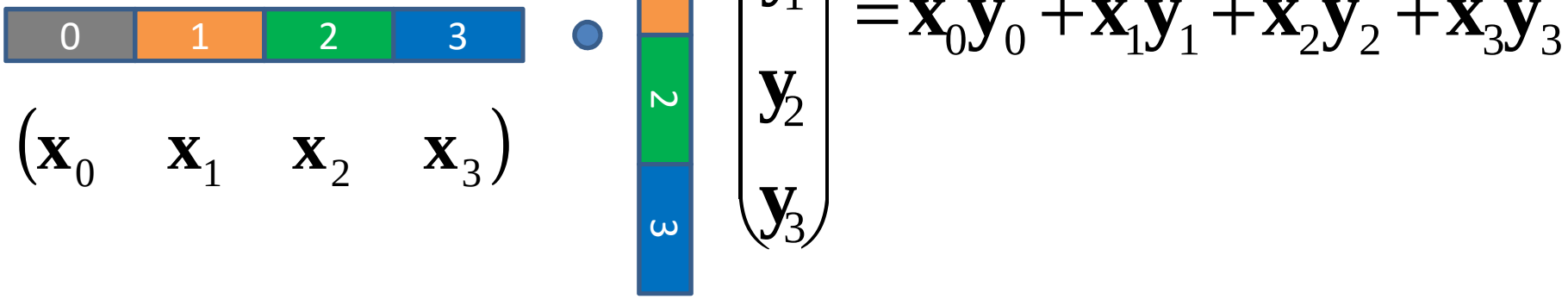
- Zawiesza wątek roboczy
- Wykonuje wszystkie czynności po dealokowaniu heap, zamykaniu plików.
- Niszczy wątek roboczy przez wywołanie TerminateThread(……).

Wątek roboczy może tworzyć i sterować teamem wątków potomnych. (Przykład)

- **Zrównoleglenie danych: ten samy algorytm (zestaw rozkazów) jest wykonany dla różnych danych. Na przykład:**
  - **obliczenie iloczynu skalarnego:**

$$a = \mathbf{x}^T \cdot \mathbf{y} = \sum_{ip=0}^{np-1} \sum_{i=\frac{N}{np}\cdot ip}^{\frac{N}{np}\cdot(ip+1)} x_i \cdot y_i$$

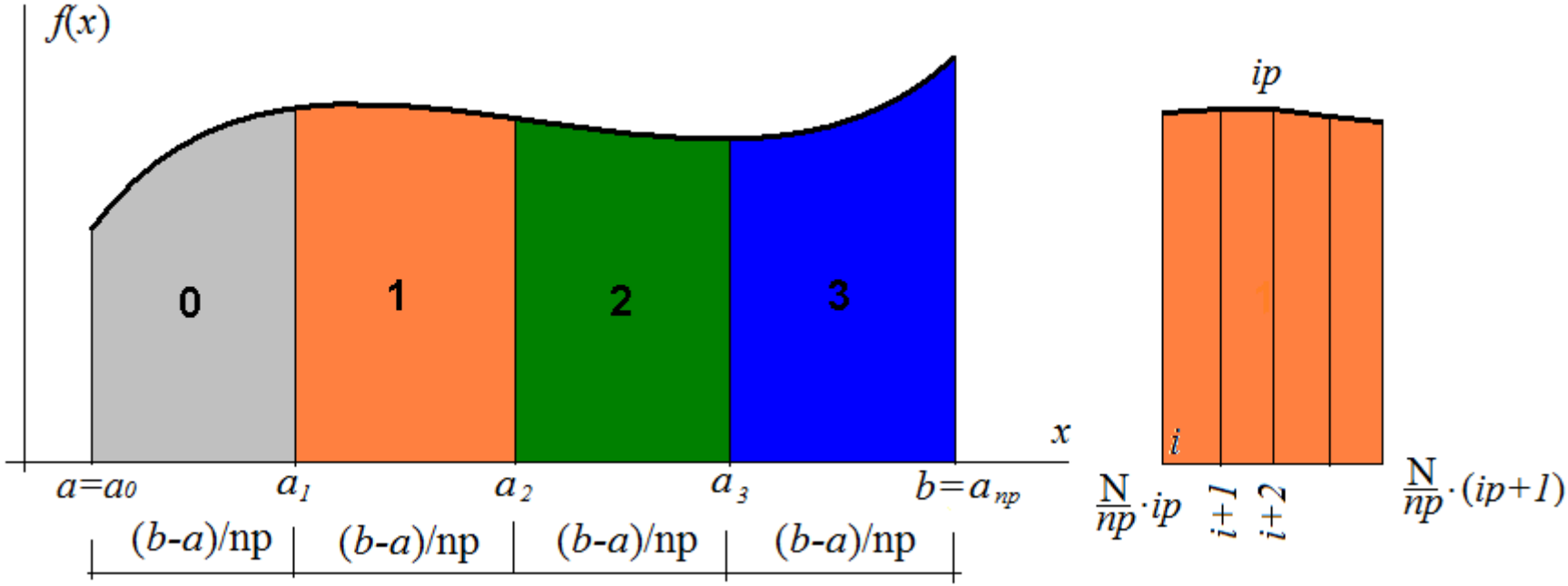
gdzie ip= numer wątku, np – ilość wątków



- **całkowanie numeryczne:**

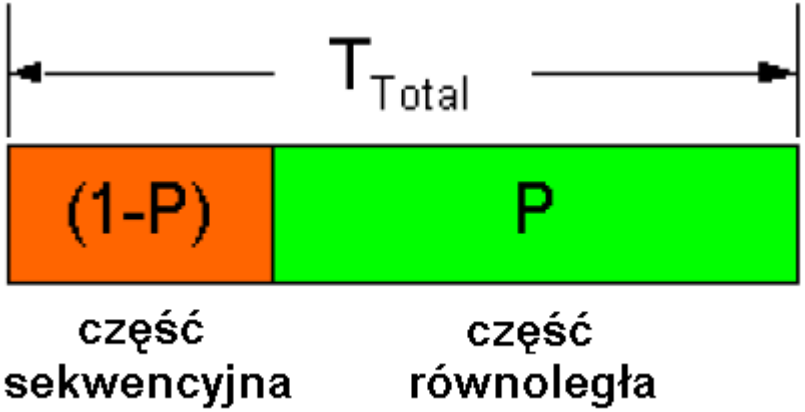
$$\int_a^b f(x)dx = \sum_{ip=0}^{np-1} \int_{a_{ip}}^{a_{ip+1}} f(x)dx \approx \sum_{ip=0}^{np-1} \sum_{i=\frac{N}{np}ip}^{\frac{N}{np}(ip+1)} \frac{f(x_i)+f(x_{i+1})}{2}\Delta x\,,$$

$$x_i = a + \frac{b-a}{np}\cdot ip + \left(i - \frac{N}{np}\cdot ip\right)\cdot \Delta x, \quad \Delta x = \frac{b-a}{N}$$



- **Zadania, który polegają na zrównolegleni danych, można rozwiązywać na podstawie OpenMP, a również, na podstawie bibliotek wielowątkowych.**

➢ **Prawo Amdahl'a – podstawa teoretyczna dla wyznaczenia górnej granicy wydajności algorytmów równoległych przy zrównolegleni danych.**
Zakładamy, że w podanym algorytmie część operacji nad danymi pozostała wykonana w trybie sekwencyjnym, a część – w trybie równoległym:



$$T_{Parallel} = \left\{ (1-P) + \frac{P}{n_p} \right\} T_{total} + O_{np}$$

T_{total} – czas wykonania algorytmu w trybie sekwencyjnym;

$T_{parallel}$ - czas wykonania algorytmu przy zastosowaniu zrównoleglenia;

P – cześć instrukcji, które są wykonane w trybie równoległym
$0 \leq P \leq 1$

n_p – ilość procesorów

O_{np} – wytarty na zrównoleglenie

- **Jeśli pominąć nakładem na zrównoleglenie O_{np}, to**

$$S = \frac{T_{total}}{T_{Parallel}} = \frac{1}{(1-P) + \frac{P}{n_p}}$$

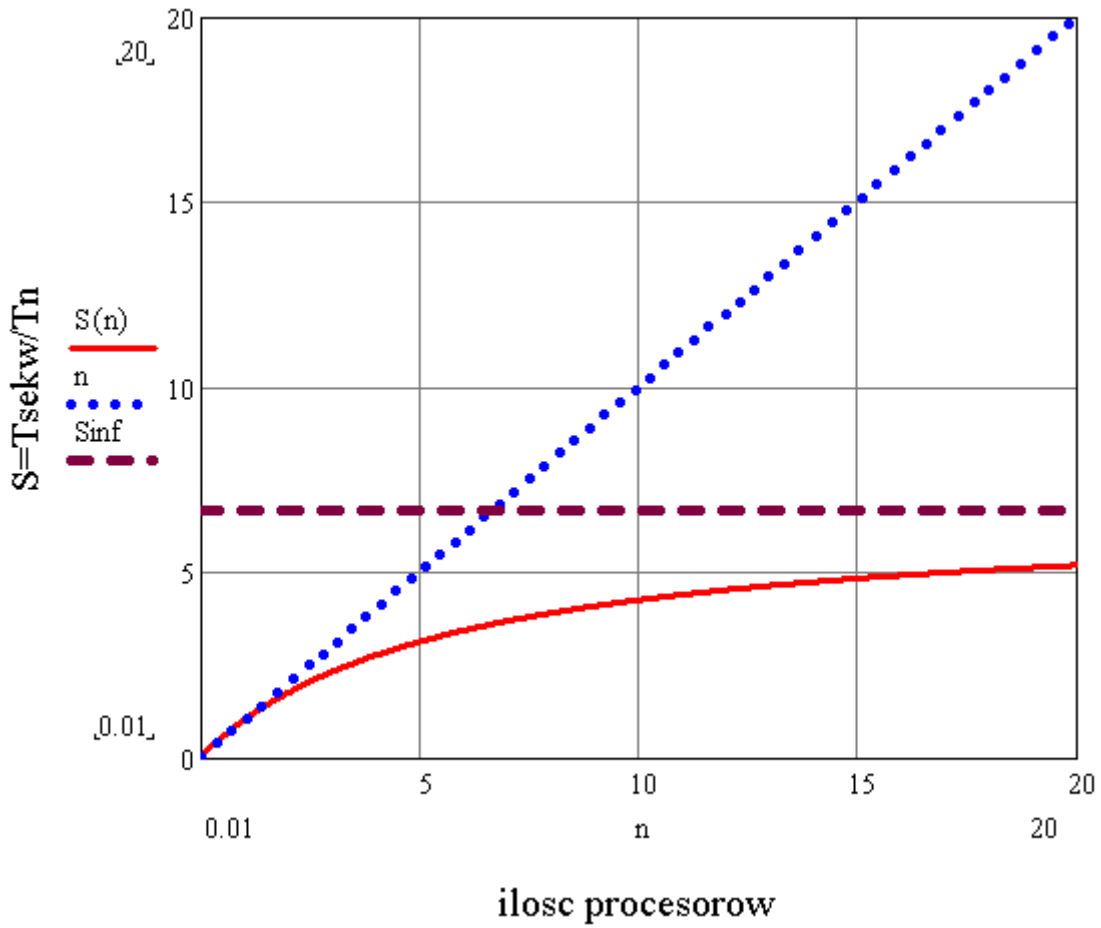
- **Faktor S (speed up) wskazuje, jak rośnie przyspieszenie obliczeń przy zwiększeniu ilości procesorów.**

- **Prawo Amdalh'a ustawia górną (teoretyczną) granice przyspieszenia dla danego algorytmu przy podanej ilości procesorów. Jeśli przyjąć $n_p \rightarrow \infty$, dostaniemy największe możliwe przyspieszenie dla danego algorytmu**

- **Prawo Amdalh'a oszacowuje teoretyczną granice przyspieszenia dla danego algorytmu wychodząc z samego algorytmu. Osobliwości pracy sprzętu nie są wzięte pod uwagę. (dot = X^T*Y)**

$$S_{inf} = \frac{T_{total}}{T_{Parallel}} = \frac{1}{(1-P)}$$

- **Przykład: dola instrukcji sekwencyjnych w algorytmie wynosi 15%. P = 0.85;**



Przyspieszenie tego algorytmu nie może przekroczyć 6.67

- **Rozdrobnienie (granularity) – jakościowa miara stosunku obliczeń do komunikacji**

  - **grube rozdrobienie (coarse granularity) – znaczna część roboty obliczeniowej jest wykonana pomiędzy zdarzeniami komunikacyjnymi. Stosuje się dla zrównoważenia obciążeń na wątki w przypadkach, kiedy nie daje się podzielić zadania dla wątków na równoważne z punktu widzenia obliczeń części.**

  - **drobne rozdrobienie (fine granularity) – odnośnie niewielka część roboty obliczeniowej jest wykonana pomiędzy zdarzeniami komunikacyjnymi. Stosuje się kiedy daje się podzielić zadania dla wątków na równoważne z punktu widzenia obliczeń części.**

- **Projektowanie aplikacji wielowątkowych składa się z**

  - **rozdzielenia zadania na tyle drobnych zadań równoległych, ile to jest możliwe (partitioning). Jedną z celów tu jest zabezpieczenie zrównoważonego obciążenia wątków (load balance) – czym drobniej podział, tym łatwiej to osiągnąć. Z drugiej strony – zbyt drobny podział powoduje duże narzuty OS na tworzenie i obsługę wątków. Dla tego często preferują odnośnie grube drobienie (coarse granularity).**

- **Projektowanie komunikacji z punktu widzenia danych i synchronizacji.**

- **Rozważanie wydajności algorytmu.**

➢ **Przykład: metoda iteracyjna dla rozwiązania układów równań liniowych algebraicznych. Stan danych na kroku iteracyjnym k+1 zależy od stany na krokach poprzednich – więc zrównolegleniu podlegają tylko funkcje, które działają na oddzielnym kroku. Dalej powstaje pytanie: ile przyjąć wątków? Od tego zależy grubość rozdrobienia. Jeśli czas przetwarzania jednej porcji danych jest mniejszy od narzutu, który system operacyjny wytraca na tworzenie i obsługę wątków, zwiększenie ilości wątków będzie powodowało spadek wydajności. To jest przykład zbyt drobnego rozdrobienia (too fine granularity).**

➢ **Tworzenie wątków wymaga pewnego narzutu czasy – dla OS Windows tworzenie wątku zgrubnie jest równoważnym ~1000 dzieleni liczb całych (int). OS musi również obsługiwać i planować wątki, wykonywać context switching. To wymaga nakładu pracy systemu operacyjnego – dla tego program równoległy powinien być dobrze projektowany, żeby minimalizować narzuty OS, który ograniczają skalowalność systemu obliczeniowego.**

- **Projektowanie komunikacji z punktu widzenia danych i synchronizacji.**

- **Rozważanie wydajności algorytmu.**

➢ **Przykład: metoda iteracyjna dla rozwiązania układów równań liniowych algebraicznych Ax = b.**

Stan danych na kroku iteracyjnym k+1 zależy od stany na krokach poprzednich – więc zrównolegleniu podlegają tylko funkcje, które działają na oddzielnym kroku. W pierwszej kolejce trzeba zrównoleglić procedurę mnożenia macierzy przez wektor, a dla metody gradientów sprzężonych z uwarunkowaniem wstępnym (*preconditioning gradient method*) - i procedury *triangular solution*, wykonane przy rozwiązaniu pomocniczego układu równań liniowych algebraicznych odnośnie preconditioning'u.
Ty preconditioning – operator B, wprowadzony dla przyspieszenia zbieżności. $B^{-1}Ax - B^{-1}b$, stąd powstaje zadanie pomocnicze $B \cdot z_k = r_k$, gdzie wektor resydualny $r_k = b - Ax_k$, x_k – przybliżenie rozwiązania na kroku *k*.

Dalej powstaje pytanie: ile przyjąć wątków? Od tego zależy grubość rozdrobienia.

Jeśli czas przetwarzania jednej porcji danych jest mniejszy od narzutu, który system operacyjny wytraca na tworzenie i obsługę wątków, zwiększenie ilości wątków będzie powodowało spadek wydajności. To jest przykład zbyt drobnego rozdrobienia (too fine granularity).

- **Tworzenie wątków wymaga pewnego narzutu czasy – dla OS Windows tworzenie wątku zgrubnie jest równoważnym ~1000 dzieleni liczb całych (int). OS musi również obsługiwać i planować wątki, wykonywać context switching. To wymaga nakładu pracy systemu operacyjnego – dla tego program równoległy powinien być dobrze projektowany, żeby minimalizować narzuty OS, który ograniczają skalowalność systemu obliczeniowego.**

- **Zrównoważenie obciążenia wątków (loading balance) jest bardzo ważne dla osiągnięcia wysokiej wydajności. Głównym celom tu jest minimalizacje czasu idle dla każdego wątku.**

- **Przyczyny idle są różne – pobieranie danych z pamięci, oczekiwanie na jakieś zdarzenie, lock przy synchronizacji, … . Trzeba minimalizować idle (czas jałowy) dla każdego wątku. Dla tego stosujemy takie techniki:, prefetch, uporządkowanie pobierania danych dla ułatwienia użycia pamięci podręcznej (usunięcie skoków, blokowanie cache, rejestrów).**

- **Dla zabezpieczenia poprawności danych globalnych jest koniecznie użycie synchronizacji. Przecież synchronizacja istotnie zmniejsza wydajność kodu i jego zdolność do przyspieszenia przy zwiększeniu ilości procesorów (speed up).**
  - **Projektujemy algorytm tak, żeby minimalizować synchronizacje.**
  - **Używamy dane lokalne wątku.**
  - **Dla dynamicznej alokacji pamięci – Thread Local Storage (TLS)**
  - **Dla synchronizacji w architekturze SMP – zmienne atomowe i Interlocked functions, albo sekcje krytyczne (spin-blocking):**

```
BOOL InitalizeCriticalSectionAndSpinCount( PCRITICAL_SECTION pcs, DWORD dwSpinCount);
DWORD SetCriticalSectionSpinCount( PCRITICAL_SECTION pcs, DWORD dwSpinCount);
```

dwSpinCount - tyle razy wątek będzie próbował wejść w zamkniętą sekcje krytyczną przed tym, jak przejść w stan idle. Jeśli mamy tylko jeden procesor, to dwSpinCount = 0 (domyślnie)

- **Kiedy wątek próbuje wstąpić w obszar, kontrolowany przez sekcją krytyczną, która jest zajęta innym wątkiem, OS przeprowadza jego z trybu użytkownika (user mode) do trybu jądra (kernel mode) – wątek przechodzi w stan oczekiwania. Procesor na to wytraca ~1000 taktów – to dość wysoka cena. Na komputerach wieloprocesorowych często się zdarza, że resurs, blokowany sekcją krytyczną, pozostanie zwolniony wcześniej, niż wątek będzie przeprowadzony w tryb jądra, i inny procesor odnowi wykonanie tego wątku.**

Przecież wielu taktów procesorów będzie wytracono na przeprowadzenie wątku w tryb jądra i odwrotnie – w tryb użytkownika. Dla tego i wymyślili spin-blokowanie – jest to skutecznie, jeśli wspólny resurs pozostaje blokowany w ciągu krótkiego odcinku czasu, który jest rzędu przeprowadzenia wątku w tryb jądra i odwrotnie.

- **Drugi sposób zmniejszyć narzuty, związane z sekcjami krytycznymi – używać non-blocking funkcje TryEnterCriticalSection, która nie przeprowadza wątek w tryb jądra, jeśli resurs jest zajęty, a nadaje możliwość wykonać korzystną robotę.**

```
while( !TryEnterCriticalSection(&cs))
{
    //do something …
}
//start of critical section block
my_shared_objekt  =  ……..
LeaveCritical section(&cs);
```

Sterowanie pamięcią

- **Dynamiczne alokowanie pamięci w stosie procesu jest kosztowna procedura, która wymaga synchronizacji wątków przy dostępie do wspólnego resursu – heap'u. Wywołanie**

PVOID HeapAlloc(HANDLE hHeap, DWORD fdwFlags, SIZE_T dwBytes);

z flagą != HEAP_NO_SERIALIZE **powoduje dostęp do heap'u tylko jednego wątku w trakcie alokowania dynamicznego. Inne wątki, jeśli też mają zaalokować pamięć w tym samym heap'ie, będą czekali dopóki pierwszy wątek nie skończy wydzielenia pamięci.**

- **Dla tego, żeby uniknąć synchronizacji i przyspieszyć alokowanie pamięci dynamicznej, można dla każdego wątku stworzyć swój własny stos z flagą** HEAP_NO_SERIALIZE **:**

HANDLE HeapCreate(HEAP_NO_SERIALIZE , 0, 0);

- **Jeśli HANDLE tego stosu przechowywać w pamięci lokalnej wątku (TLS – Thread Local Storage), ten stos można używać dla dynamicznego wydzielenia i zwolnienia pamięci danego wątku.**

- **Najbardziej sensownym jest umieszczenie tej procedury w DLL:**

```
static DWORD tls_key;

BOOL APIENTRY DllMain( HMODULE hModule,   DWORD  ul_reason_for_call,   LPVOID lpReserved )
{
        switch (ul_reason_for_call)
        {
        case DLL_PROCESS_ATTACH:
                        if((tls_key = TlsAlloc()) == TLS_OUT_OF_INDEXES)
                        {
                                        cout << " TlsAlloc problem during dll_process_attach\n";
                        }
                        if(!TlsSetValue(tls_key, GetProcessHeap()))
                        {
                                        cout << "TlsSetValue problem during dll_process_attach\n";
                        }
                        break;
        case DLL_THREAD_ATTACH:
                        if(!TlsSetValue(tls_key, HeapCreate(HEAP_NO_SERIALIZE, 0, 0)))
                        {
                                        cout << "TlsSetValue problem during dll_thread_attach\n";
                        }
                        break;
        case DLL_THREAD_DETACH:
                        HeapDestroy(TlsGetValue(tls_key));
                        break;
        case DLL_PROCESS_DETACH:
                        TlsFree(tls_key);
                        break;
        }
    return TRUE;
}
```

```
__declspec (dllexport) void * thr_malloc(size_t n)
{
        return HeapAlloc(TlsGetValue(tls_key), 0, n);
}

__declspec (dllexport) void thr_free(void *ptr)
{
        return HeapFree(TlsGetValue(tls_key), 0, ptr);
}
```

- **Tu każdy wątek używa TLS dla przechowywania HANDLE do swojego własnego heap'u – nie powstaje problemu wspólnego resursu (heap) dla różnych wątków.**

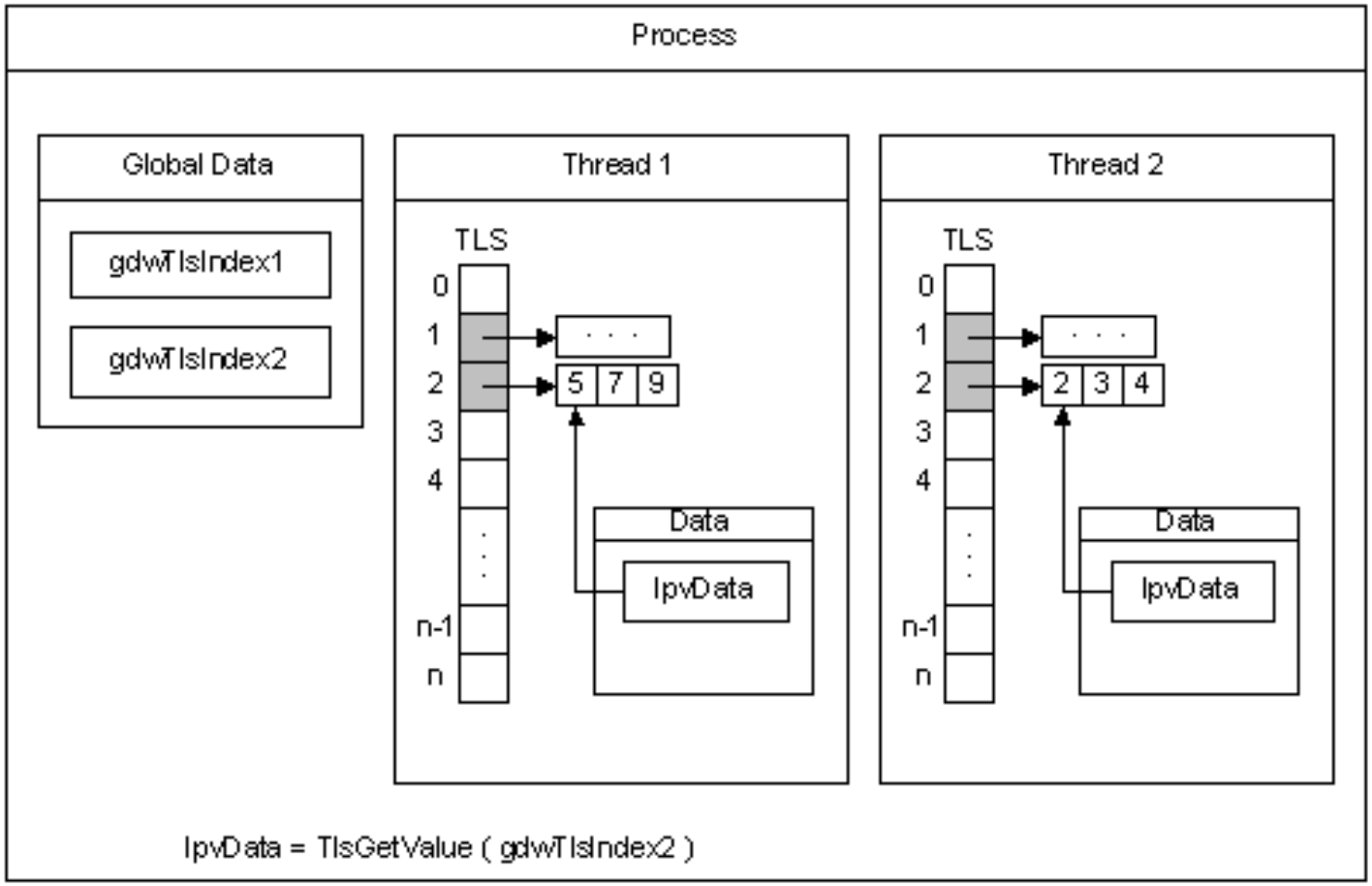
Pamięć lokalna wątku – Thread Local Storage

- **Zmienne statyczne i globalne są wspólne dla każdego wątku danego procesu. TLS nadaje możliwość stworzyć dla każdego wątku unikalne dane, do których proces może mięć dostęp, używając indeks globalny.**

- **Jeden wątek alokuje indeks, który może być użyty innym wątkiem dla dostępu do unikalnych danych, związanych z pierwszym wątkiem.**

- **Kiedy wątki są tworzone, OS alokuje tablice typu LPVOID dla TLS indeksów i inicjuje ją jako NULL. Przed tym, jak użyć indeks, on powinien być alokowany jednym z wątków przez wywołanie funkcji TlsAlloc().**

- **Każdy wątekk przechowuje swoje dane, które odpowiadają TLS indeksowi, w TLS Slot. Dane, związane z TLS indeksem, muszą być rzutowane do typu LPVOID, i wtedy można bezpośrednio umieścić ich w TLS Slot.**

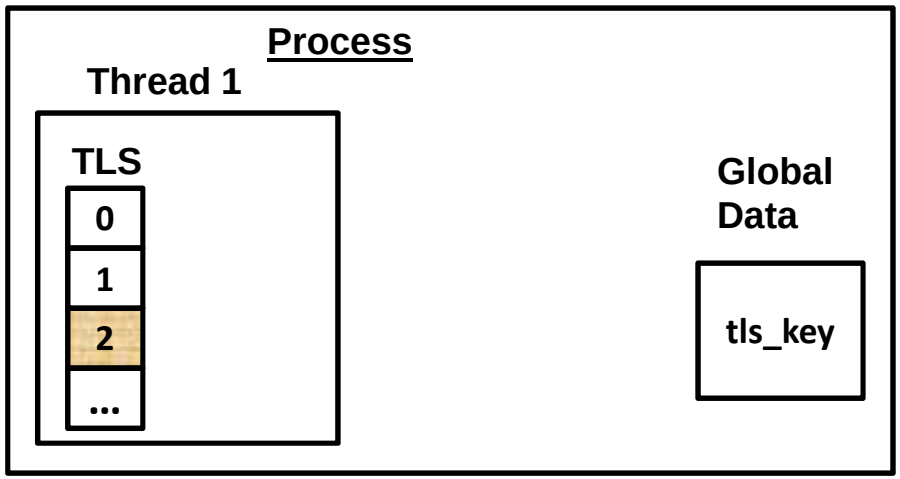


- **W tym przykładzie proces ma dwa wątki, Thread1, Thread2, i alokuje 2 indeksy dla użycia w TLS, gdwTlsIndex1 i gdwTlsIndex2.**

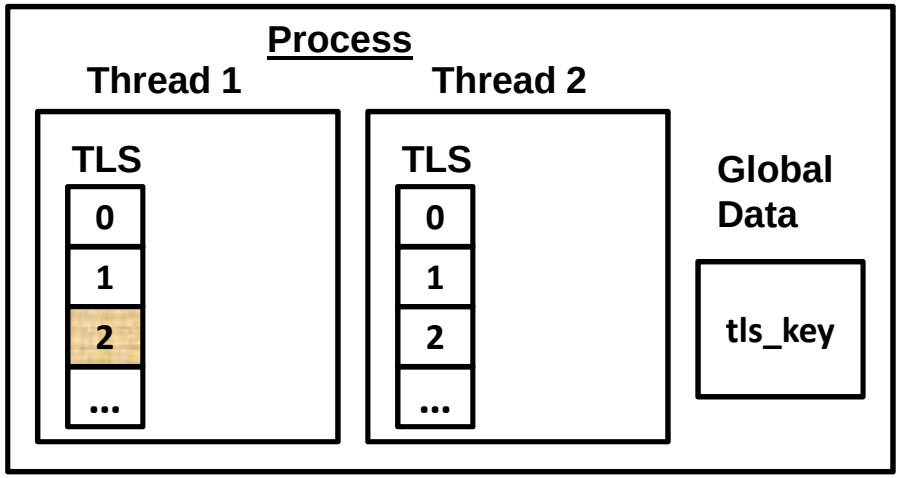
- **Każdy wątek alokuje dwa bloki pamięci (jeden blok dla każdego indeksu), w których przechowuje dane, a wskaźniki do tych bloków zapisuje w odpowiednie TLS Sloty.**

- **Żeby się dostać do danych, związanych z indeksem, wątek pobiera wskaźnik do bloku pamięci z TLS Slotu, i przypisuje jego wartość do zmiennej lokalnej lpvData.**

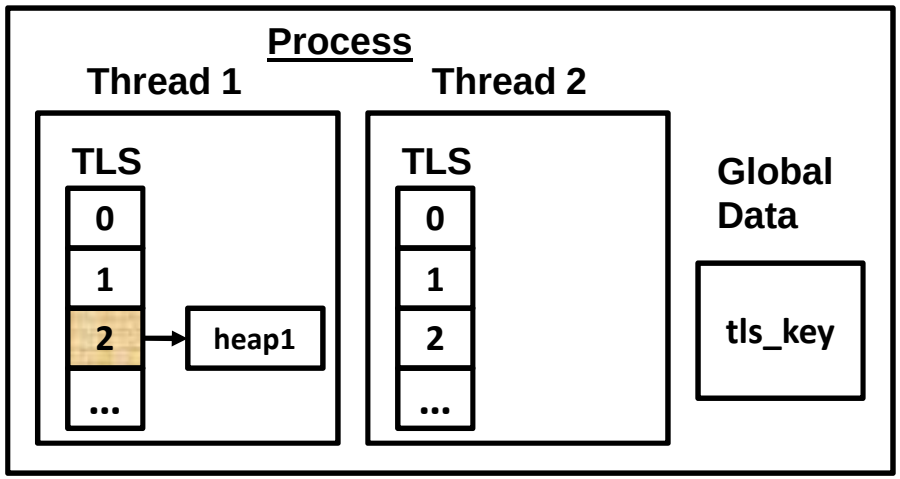
- **Spojrzeć jeszcze raz kod wydzielenia pamięci w DLL (przykład ParMV)**



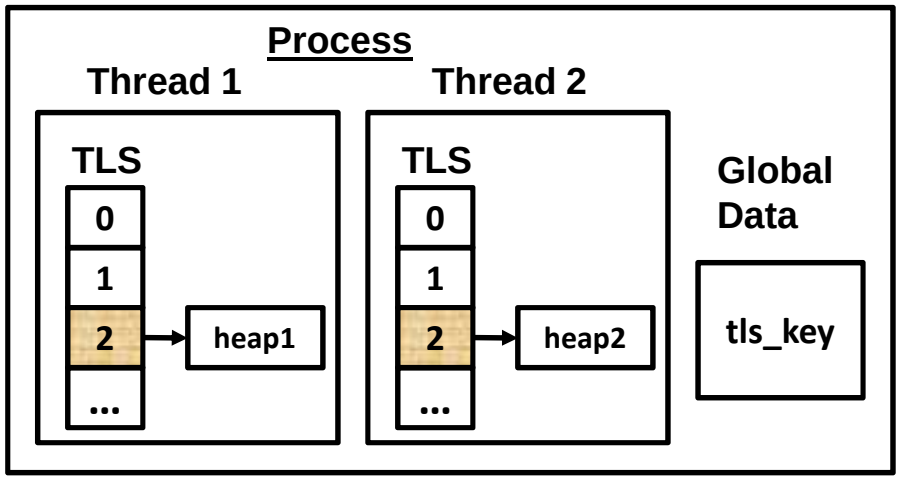
1. wątek 1 procesu alokuje TLS indeks – w tablice TLS wydziela pozycje: tls_key = TlsAlloc(); Zmienna globalna tls_key dostaje numer elementu w tablice (tls_key = 2)



2. Powstaje wątek 2. On ma swój własny slot TLS, który od razu po tworzeniu jest pusty.

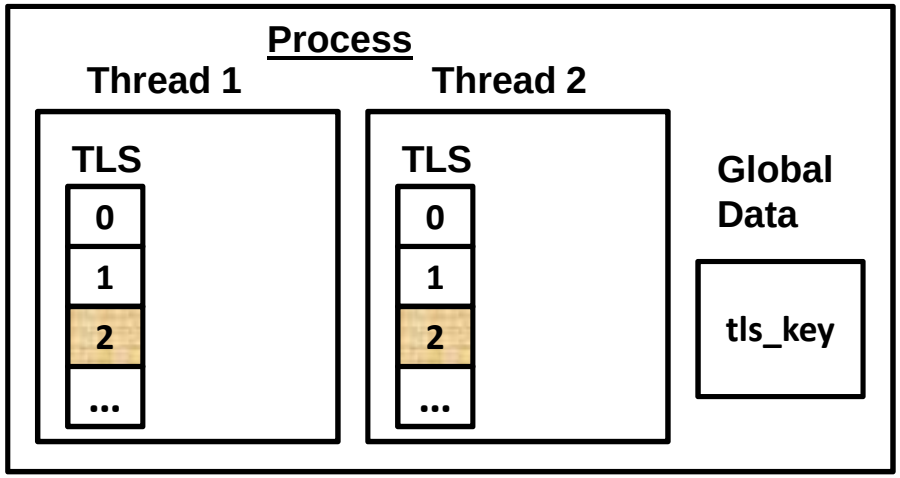


3. Wątek 1 umieszcza HANDLE heap_1 w element TLS tablicy o numerze tls_key z : TlsSetValue(tls_key, HANDLE heap_1)



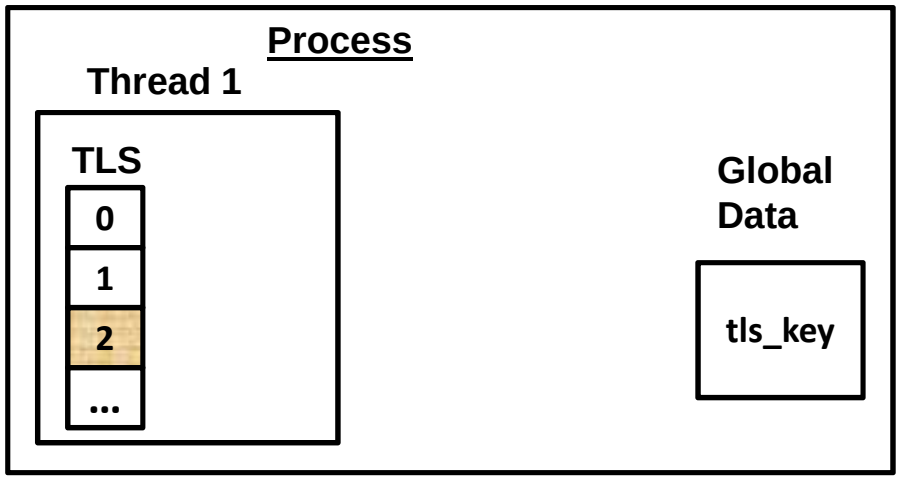
4. Wątek 2 umieszcza HANDLE heap_2 w element TLS tablicy o numerze tls_key z : TlsSetValue(tls_key, HANDLE heap_2)

Teraz w każdym wątku mamy dostęp do swojego własnego heap przez zmienną globalną tls_key: HANDLE heap_1 = TlsGetValue(tls_key);
HANDLE heap_2 = TlsGetValue(tls_key);
Każdy wątek pobiera element swojej własnej tablicy TLS o numerze tls_key i dla tego dostaje dostęp do swojego własnego heap.

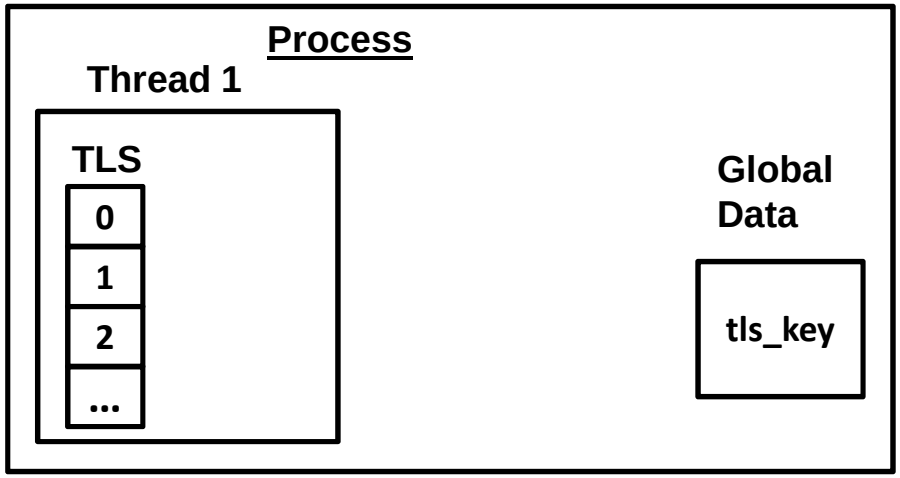


5. Przed zakończeniem pracy każdy wątek niszczy swój własny heap:

HeapDestroy(TlsGetValue(tls_key));



6. Przy zakończeniu pracy wątek 1 niszczy swoje resursy, a również i slot TLS:



7. Wątek 1 zwalnia element tablicy TLS o numerze tls_key – ustawia ten element na NULL:

TlsFree(tls_key);

Teraz ten element slotu TLS może być użyty dla innych celów.

- **Wielu algorytmów przypuszcza usunięcie synchronizacji, jeśli użyć TLS zamiast danych wspólnych (shared data). To może istotnie zmniejszyć narzut i podnieść wydajność algorytmu.**

- **Przy użyci TLS dane mogą się pozostać w pamięci podręcznej procesorów znacznie dłuższej od przypadku, kiedy są używane dane wspólne, jeśli procesory nie mają wspólnej pamięci podręcznej.**

- **Dane wspólne przy modyfikacji i zapisu jednym z procesorów powodują zgłoszenia o niepoprawności (invalidation) pamięci podręcznej innych procesorów – powstaje przeładowanie cache - linii. Przy użyci TLS taka sytuacja jest wykluczoną – TLS data mogą być umieszczone w pamięci podręcznej tylko jednego procesora.**

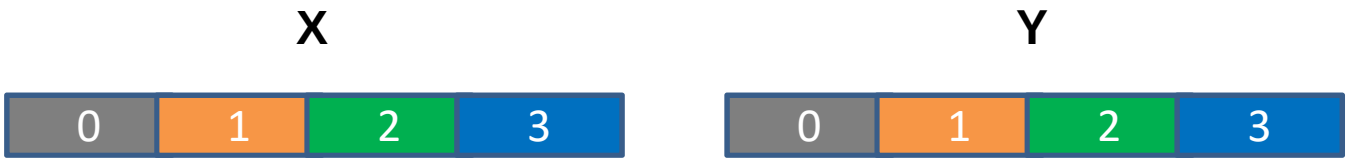
- **Osobliwości Hyper-Threading'u (procesory Intel). Hyper-Threading – taka technologia, kiedy dwa logicznych procesory rozdzielają pomiędzy sobą jeden cache L1. Częsty dostęp do adresów pamięci wirtualnej, nie rozdzielonych wartością 64 KB, może powodować istotny spadek wydajności. Stack potoku zwykłe nie jest wyrównany po granicach 64 KB, i jeśli w tym samym cache znajdują się stack'i dwóch potoków, przyczyna cache-konfliktów jest oczywista - steki dwóch potoków będą mieli wspólną stronę pamięci o rozmiarze 64 KB. Rozmiar 64 KB zależy od typu procesora.**

- **Jednym ze sposobów, który nadaje możliwość poprawić sytuacje, jest wprowadzenie stack offsets, które w sposób sztuczny rozdzielają przestrzeń adresową pomiędzy danymi każdego potoku. Szczegóły są podane w podręczniku użytkownika korporacji Intel: Developing Multithreaded Applications: A Platform Consistent Approach.**

- **Przykład: obliczenie iloczynu skalarnego: dot_prod = X^T*Y**

- **Rozdzielenie danych pomiędzy wątkami:**

Coarse granularity

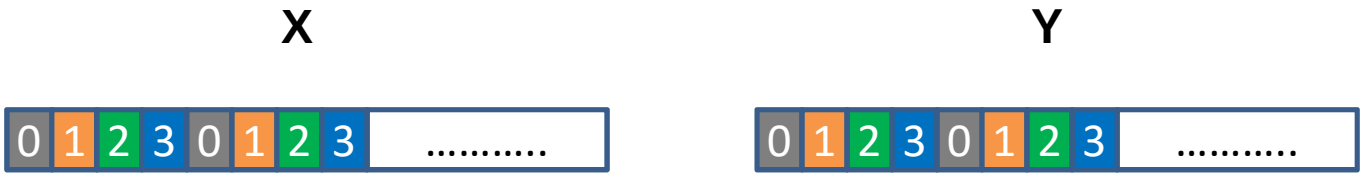


o **Ilość podziałów danych jest taka sama, jak ilość procesorów. Zabezpiecza load balance (zrównoważenie pomiędzy wątkami).**

o **Narzut OS na tworzenie i planowanie wątków jest minimalny. Związek komunikacyjny pomiędzy wątkami jest słaby. Synchronizacja – na poziomie sygnałów przy zakończeni wątków. Ilość operacyj etapu „reduction” (sumowanie wyników działania każdego wątku po ich zakończeniu – sekwencyjna część kodu) – minimalna.**

o **Dane w pamięci dla każdego wątku są umieszczone ciągłe – brak skoków danych, minimalna ilość przeładowań pamięci podręcznej.**

Coarse granularity



- **Ilość podziałów danych istotnie większa od ilości procesorów. Zabezpiecza load balance (zrównoważenie pomiędzy wątkami). Ilość wątków jest taka sama jak ilość procesorów.**

- **Narzut OS na tworzenie i planowanie wątków jest minimalny. Synchronizacja – na poziomie sygnałów przy zakończeni wątków. Ilość operacyj „reduction” (sumowanie wyników działania każdego wątku po ich zakończeniu – sekwencyjna część kodu) – minimalna. Tu wątki nie komunikują pomiędzy sobą przy przejściu od jednego podziału danych do innego – dla tego i coarse granularity.**

- **Dane dla każdego wątku są pobierane ze skokami – w przypadku dużych skoków może powstać częste przeładowanie pamięci podręcznej, wtedy dostaniemy drastyczny spadek wydajności.**

Fine granularity

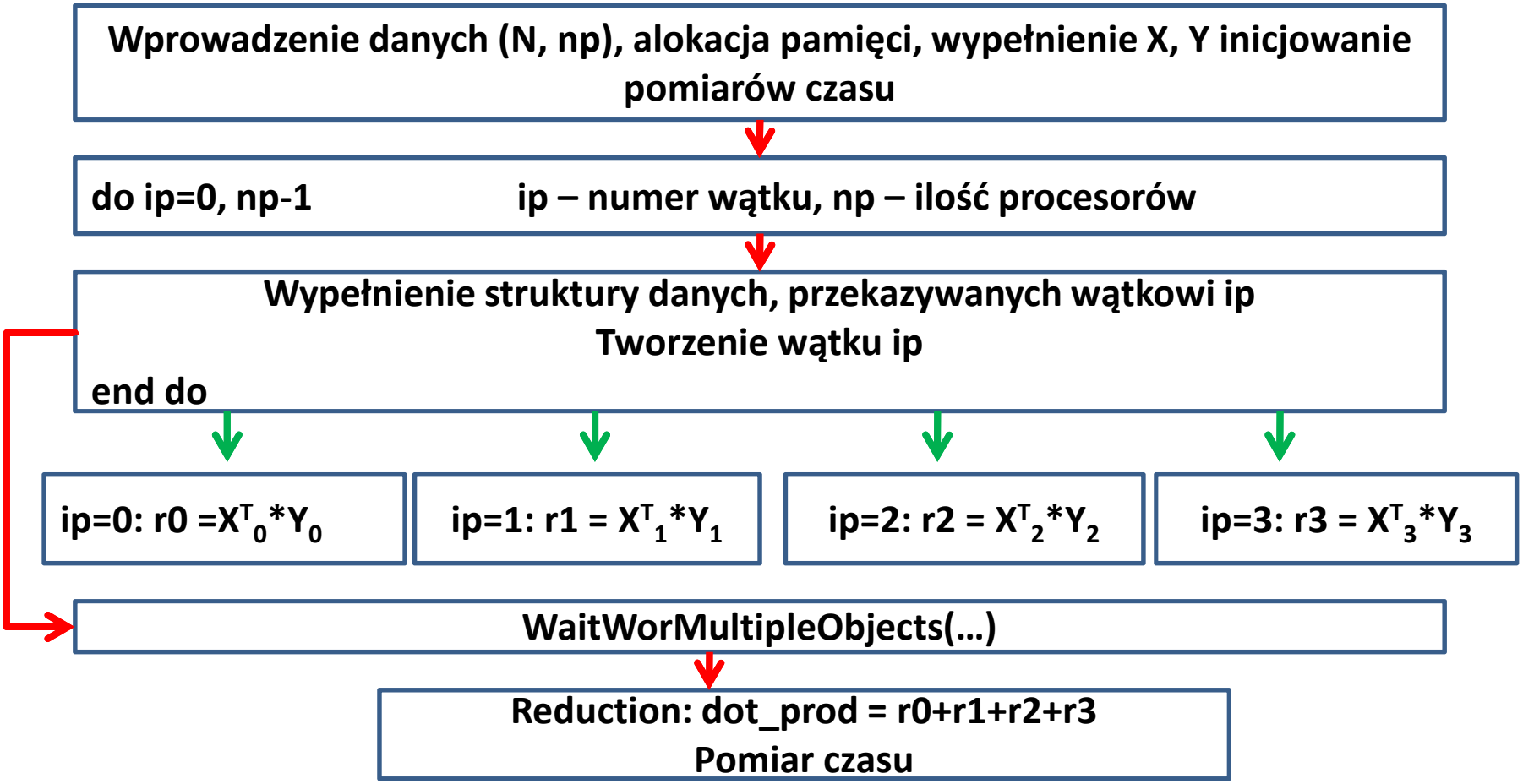
- **Ilość podziałów danych istotnie większa od ilości procesorów. Zabezpiecza load balance (zrównoważenie pomiędzy wątkami). Ilość wątków jest taka sama jak ilość podziałów danych.**

- **Narzut OS na tworzenie i planowanie wątków jest znacznie większy od poprzednich przypadków. Synchronizacja – na poziomie sygnałów przy zakończeniu wątków. Ilość operacji etapu „reduction” (sumowanie wyników działania każdego wątku po ich zakończeniu – sekwencyjna część kodu) – też jest większa.**

- **Dane dla każdego wątku są pobierane ciągłe, przecież w pamięci podręcznej każdego procesora przy dużej objętości danych powstanie przeładowanie przy podłączeniu wątku do procesora - powoduje zmniejszenie wydajności wydajności.**

Wybieramy pierwszy schemat podziału danych:

Schemat blokowy algorytmu: - wątek pierwotny - wątek potomne



Uwagi:

- **Tablicy X, Y w obszarze każdego wątku potomnego są tylko do odczytu – tablicy można bez problemów rozdzielić pomiędzy wątkami:**

adres_local_X = adres_X[ip*local_N],

gdzie local_N = N/np .

- **Przy dostępie do elementów tablicy w obszarze każdego wątku synchronizacja nie jest wymagana (indeksy elementów tablicy – dane lokalne wątku)**

- **Cache-konfliktów nie powstaje – dane są tylko do odczytu**

- **Wynik działania każdego wątku – $r_{ip} = X^T_{ip} * Y_{ip}$ – definiujemy jako zmienna lokalna wątku, a po obliczeniu wartość r_{ip} przypisujemy do pola struktury danych, przekazywanych wątkowi ip:**

```
r = 0.0;
for(i=0; i<local_N; i++)
      r += X[i]*Y[i];
dane_ptoku_ip->wynik_potoku = r;
```

- **Synchronizacja: pierwotny wątek jest zawieszony dopóki watki potomne nie skończą swojej pracy. Każdy wątek potomny działa niezależnie – speed up algorytmu jest ograniczony tylko przepustowością magistrali-pamięci głównej**

- **Przykład DotProd, OMP_DotProd**

Wyniki numeryczne

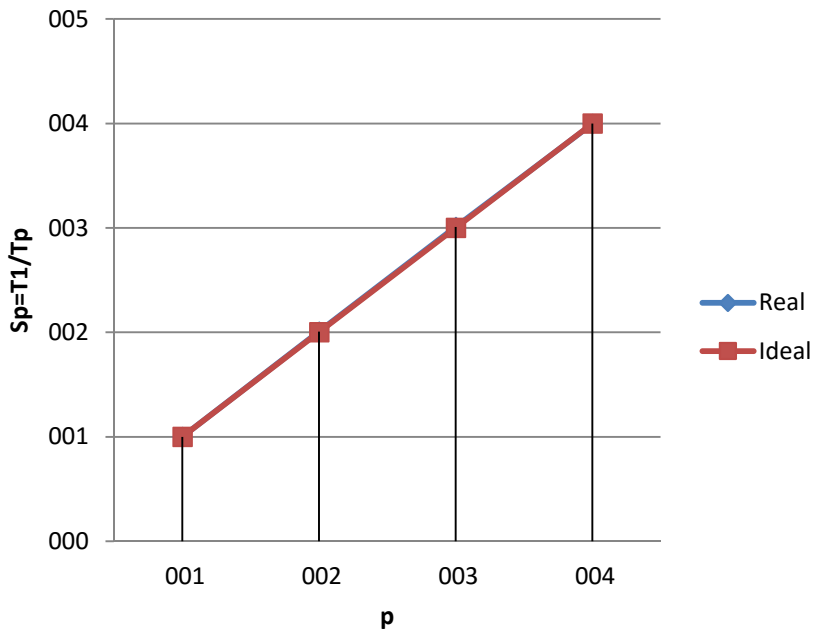
- **Dane sprzętu:**

Procesor – Intel® Core™2 Quad CPU Q6600 @2.40 GHz – 4 procesory
Cache – 4xL1: 32 KB (code), 4xL1: 32 KB (data), 2xL2:4096 KB
Pamięć: DDR2 800 MHz 4 GB

- **Dla wiarygodności pomiarów czasu pozostała wprowadzona powtarzalność obliczeń na poziomie każdego wątku potomnego (tylko w obszarze równoległej części kodu):**

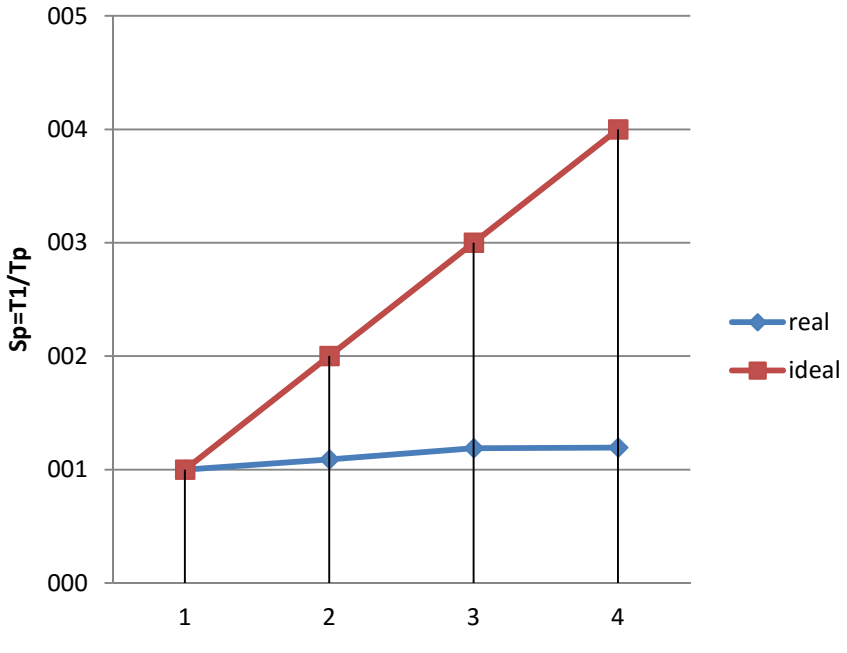
```
for(it=0; it<ptrDat->ntimes; it++)  //powtarzamy mnożenie ntimes razy
{
        rrr = 0.0;
        for(i=0; i<loc_N; i++)
        {
                rrr += X_loc[i]*Y_loc[i];
        }
        ptrDat->res = rrr;
}
```

- **Przedstawiamy wyniki wersji release (/O2)**



Rozmiar wektorów N=100 000, ntimes = 100 000

Speed up jest idealny



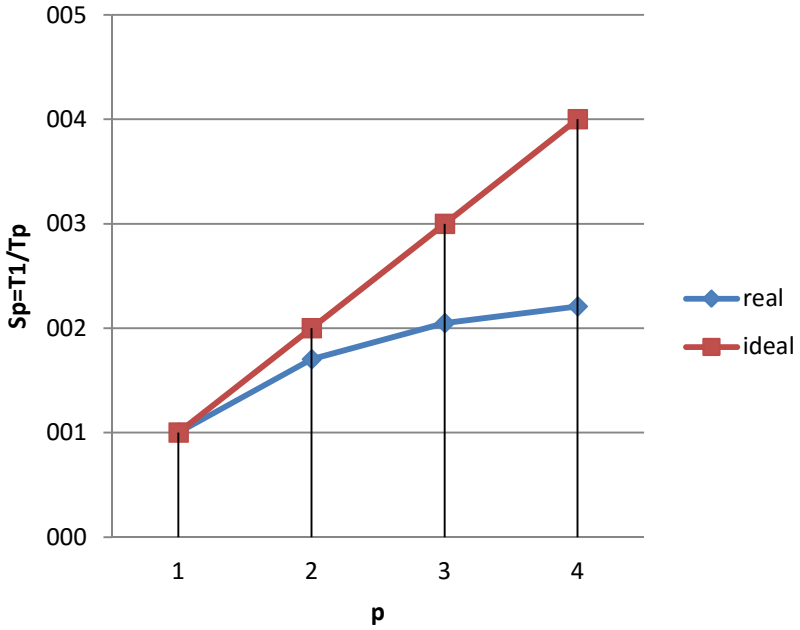
Rozmiar wektorów N=100 000 000, ntimes = 100

Speed up jest bardzo słaby

- **Ilość operacji zmiennoprzecinkowych w tych dwóch przykładach jest taka sama – $2 \times N \times ntimes = 2x10^{10}$.**

- **W pierwszym przykładzie rozmiar tablic X, Y wynosił 1.53 MB, co się mieści w cache L2 podanego komputera. Dla tego, że przedstawiony algorytm nie wymaga przeładowania cache linii przy powtarzani mnożenia, wydajność i speed up są wysokie.**

- **W drugim przykładzie rozmiar tablic X, Y wynosił 1 526 MB. W trakcie mnożenia linie cache L2 pozostają wielokrotnie przeładowane, co drastycznie zmniejsza wydajność.**



Debug wersja,

Rozmiar wektorów N=100 000 000, ntimes = 100

Speed up jest istotnie lepszy niż w wersji release.

Wydajność algorytmu przy różnych rozmiarach tablic X, Y, MFLOPS

Ilość wątków, np	100 000 x 100 000	100 000 000 x 100 release	100 000 000 x 100 debug
1	1 594	680	373
2	3 200	742	634
3	4 796	810	764
4	6 369	814	823

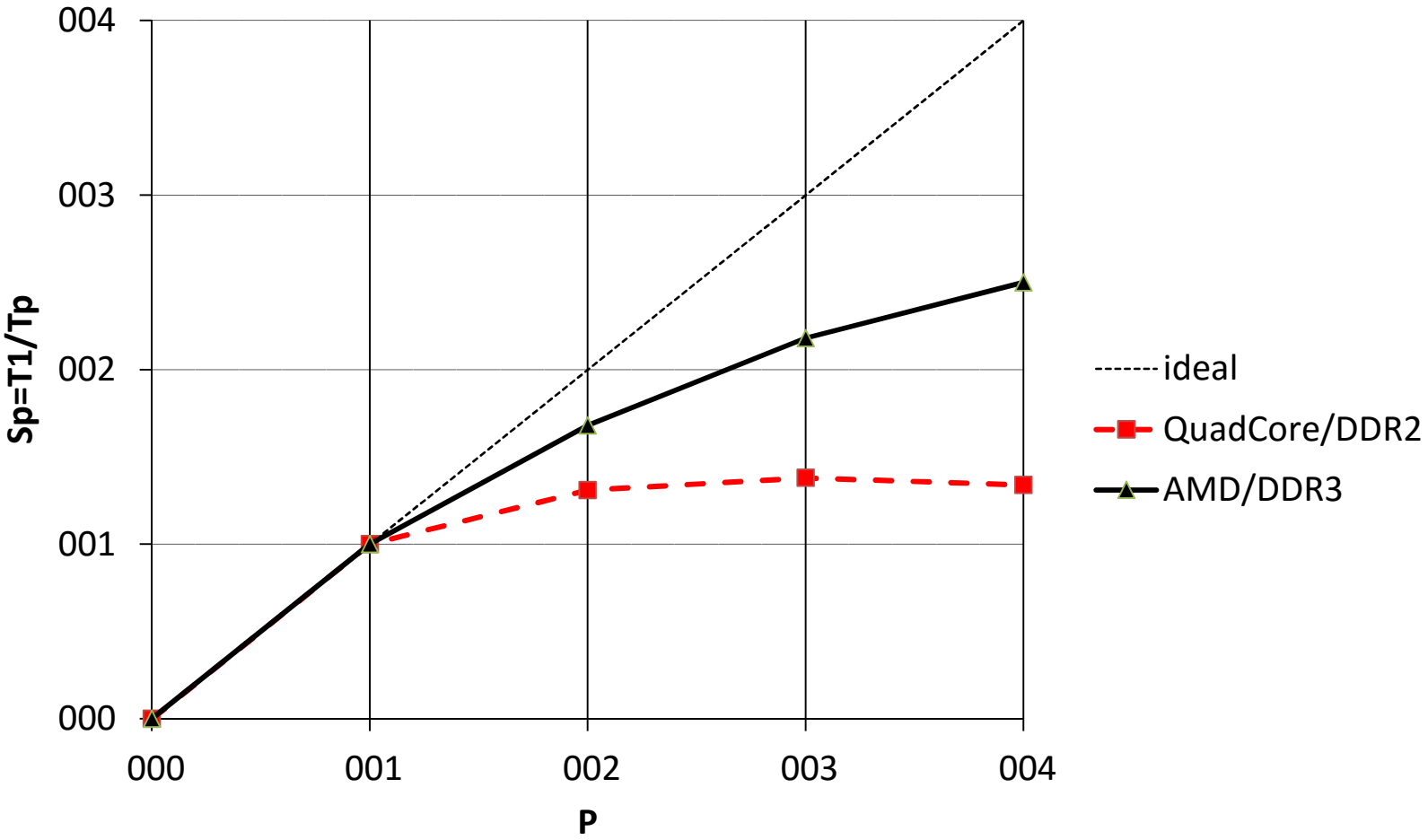
Obliczenia na różnych komputerach

AMD Phenom™ II x4 995 3.2 GHz; L1: 4x64 KB L2: 4x512 KB L3: 6 MB
Memory: DDR3 669 MHz 8 GB, OS: Windows Vista x64

Test dotProd = X*Y (N = 100 000 000 ntimes = 50 ia32)

Proc	QuadCore (2.4 GHz)		I7 (3.2 GHz)		AMD Phenom IIx4 995	
	Time, ms	$Sp=T_1/T_p$	Time, ms	$Sp=T_1/T_p$	Time, ms	$Sp=T_1/T_p$
1	15 719	1			16 880	1
2	14 531	1.08			9 828	1.72
3	14 051	1.12			7 410	2.23
4	13 766	1.14			6 428	2.63

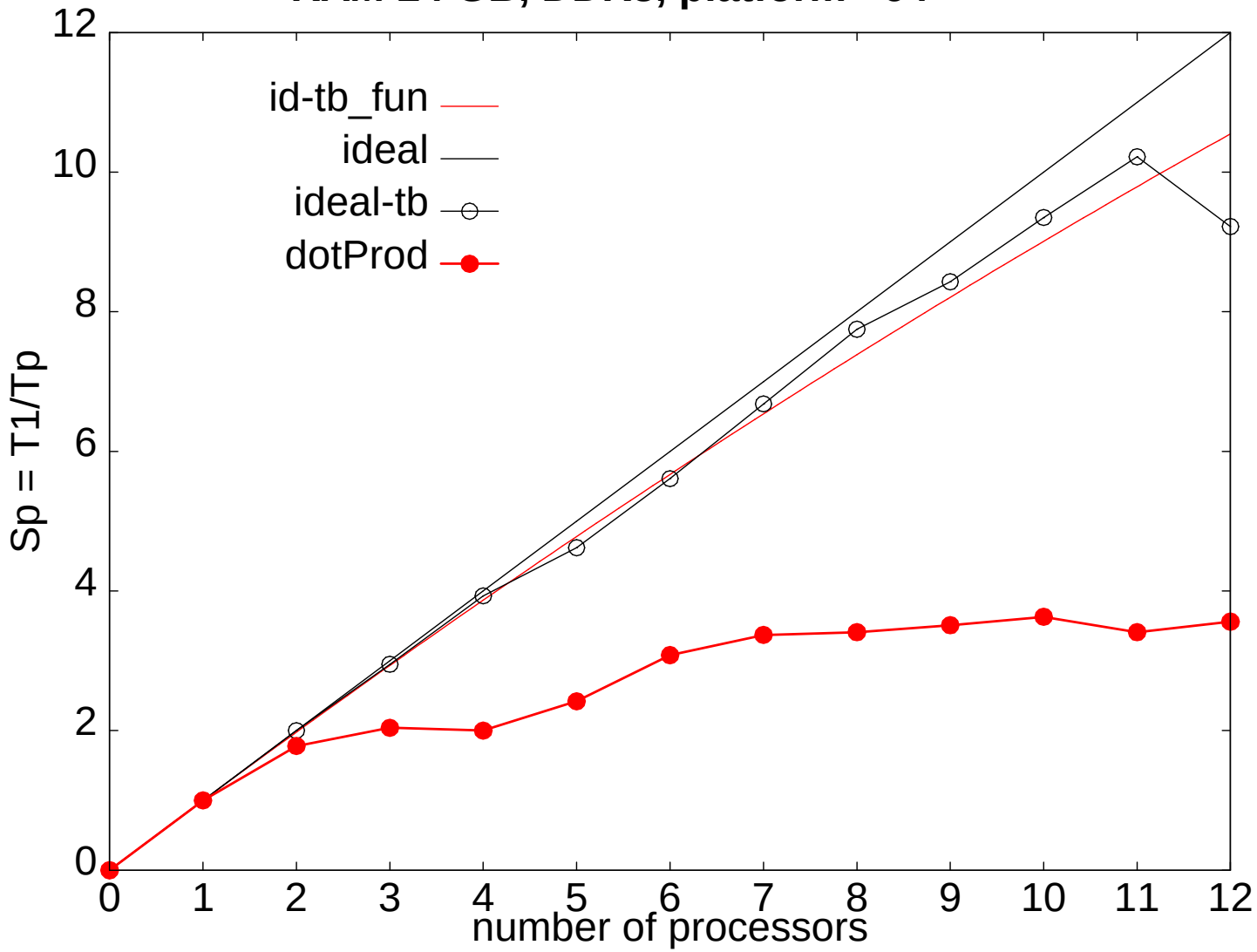
dot=X*Y, Przyspieszenie czasu obliczeń przy zwiększeniu ilości wątków dla różnych komputerów



dot=X*Y, N = 100 000 000, ntimes = 40
Intel Xeon X5660 @ 2.8 GHz /3.2 GHz (2 proc × 6 cores = 12 cores),
RAM 24 GB, DDR3, platform ×64

Nos of Processors	Computing time, s	$Sp=T_1/T_p$
1	8.611	1.000000E+00
2	4.851	1.775098E+00
3	4.228	2.036660E+00
4	4.305	2.000232E+00
5	3.557	2.420860E+00
6	2.792	3.084169E+00
7	2.558	3.366302E+00
8	2.527	3.407598E+00
9	2.450	3.514694E+00
10	2.371	3.631801E+00
11	2.527	3.407598E+00
12	2.418	3.5612

dot=X*Y, N = 100 000 000, ntimes = 40
Intel Xeon X5660 @ 2.8 GHz /3.2 GHz (2 proc × 6 cores = 12 cores),
RAM 24 GB, DDR3, platform ×64



- **Te procesory używają tryb turbo-boost: przy obciążeniu niewielkiej ilości rdzeni częstotliwość procesora jest podniesiona do 3.2 GHz; przy obciążeniu dużej ilości rdzeni częstotliwość procesorów spada do wartości nominalnej – 2.8 GHz. Dla tego prosta "*ideal*" nie opisuje idealnego przyspieszenia systemu równoległego na takim komputerze, ponieważ nie biorę pod uwagę zmienność częstotliwości procesorów.**

- **Krzywa "*id-tb fun*" jest aproksymacją idealnego przyspieszenia w trybie turbo boost parabolą kwadratową, która przechodzi przez punkty (0; 0), (1;1), (12; 10.5), gdzie wartość 10.5 = 12*2.8/3.2 (zakładamy że przy obciążeniu wszystkich rdzeni częstotliwość procesora zmniejszy się do 2.8 GHz, a przy obciążeniu tylko jednego rdzenia częstotliwość wynosi 3.2 GHz).**

- **Krzywa "*ideal-tb*" jest wynikiem testowania programu, który intensywnie liczy funkcje trygonometryczne i praktycznie nie obciąża układ pamięci. Można uważać, że speed-up dla takiego algorytmu będzie bardzo przybliżony do przyspieszenia idealnego w trybie turbo boost.**

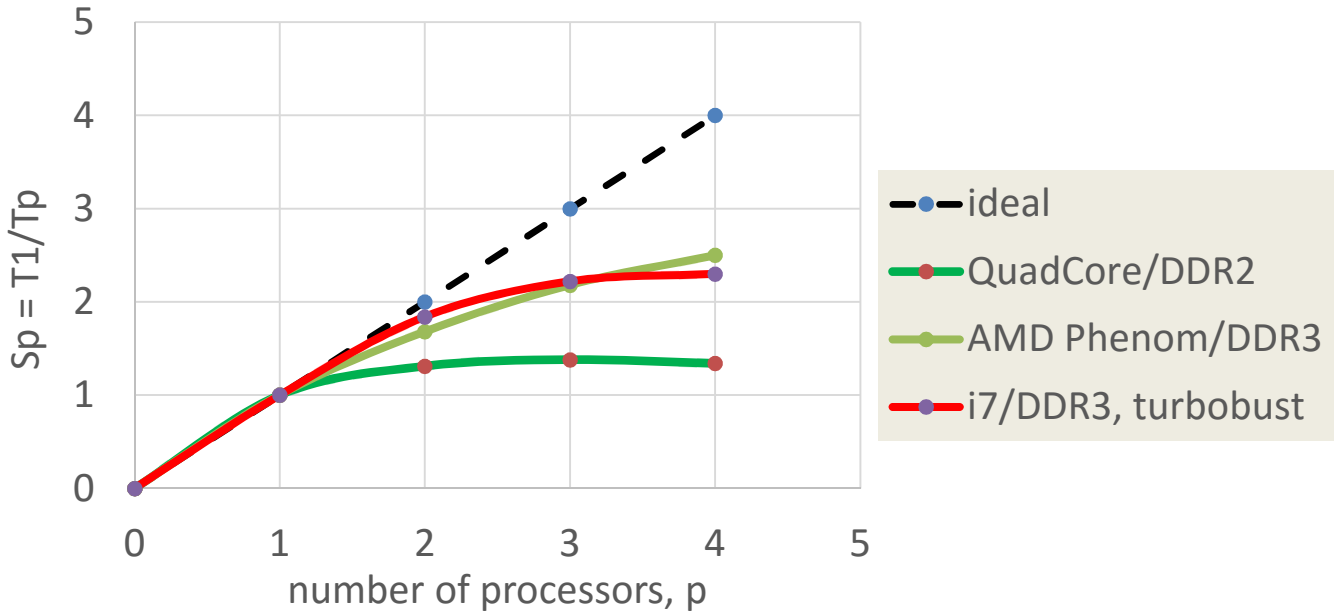
- **Współczynnik wydajności q = f/m (f – ilość operacji arytmetycznych, m – ilość przesyłek danych) dla podanego algorytmu wynosi 1. To oznacza, że na każdą przesyłkę danych wypada jedna operacja arytmetyczna. Założymy, że trwałość jednej operacji arytmetycznej wynosi 1 ns, a trwałość przesyłki jednego słowa double – 100 ns. To oznacza, że 99 ns procesor będzie czekał następnej przesyłki danych.**

Teraz zwiększamy ilość procesorów do 2 - za 1 cykl każdego procesora będzie wykonywana 1 operacja arytmetyczna – razem 2 operacji arytmetyczne. Magistral musi obsłużyć 2 procesora i przesłać 2 kolejne słowa double. Jeśli magistral nie ma zapasu przepustowości, będzie potrzebne dla tego 200 ns. Czas oczekiwania każdego procesora będzie zwiększony praktycznie dwukrotnie, przecież ilość operacji arytmetycznych, wykonywanych każdym procesorem, zmniejszyłaś też dwukrotnie, więc ogólny czas wykonania zadania pozostaje taki samy, jak w przypadku obliczeń sekwencyjnych.

- **Podobna sytuacja występuje i dla algorytmów q = 2 . (Mnożenie macierzy przez wektor, mnożenie macierzy przez macierz, algorytm naiwny, i t. d.). Typowy speed-up przedstawiona na podanym poniżej wykresie.**

Test ParMV: y = A*x (N = 10 000; ntimes = 50; ia32 application)

Proc	QuadCore (2.4 GHz)		I7 (3.4/2.4 GHz)		AMD Phenom IIx4 995	
	Time, ms	$Sp=T_1/T_p$	Time, ms	$Sp=T_1/T_p$	Time, ms	$Sp=T_1/T_p$
1	9 469	1	5 716	1	8 424	1
2	7 250	1.31	3 104	1.84	5 023	1.68
3	6 884	1.38	2 508	2.22	3 868	2.18
4	7 078	1.34	2 480	2.30	3 370	2.50



Powstaje pytanie: a istnieją takie algorytmy, które można łatwo przyspieszyć przy zrównolegleniu na komputerach wielordzeniowych?

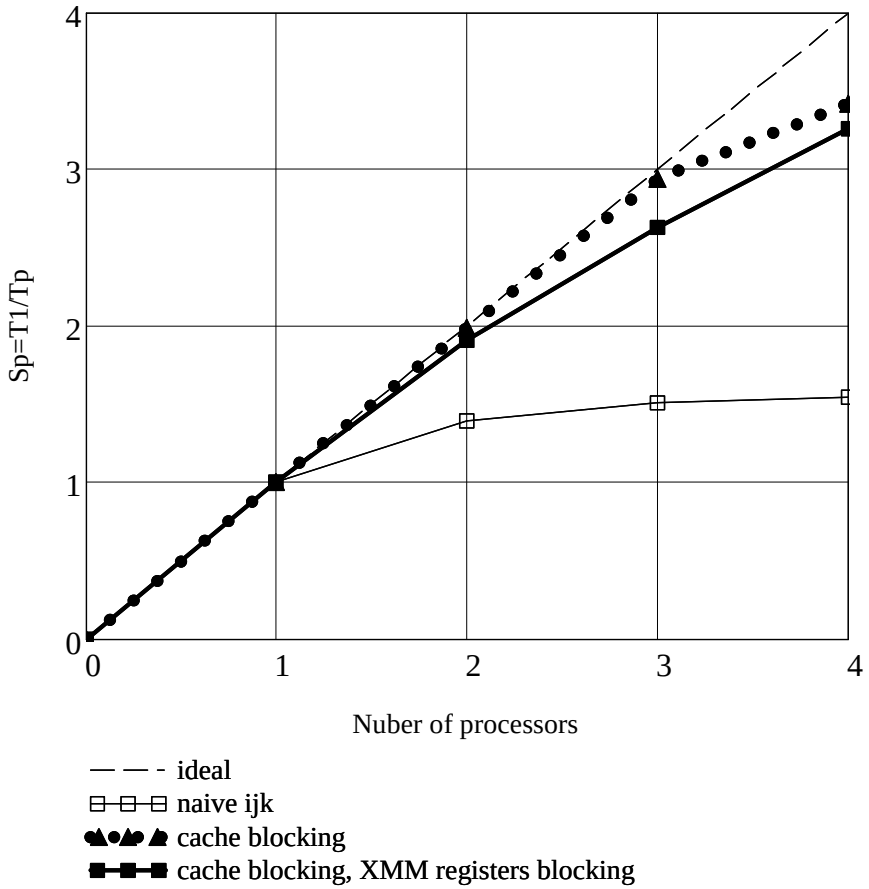
Tak, mnożenie macierzy przez macierz, algorytm blokowy. Oszacowanie współczynnika wydajności wynosi

$$q \sim \sqrt{M}$$

gdzie M – rozmiar cache L1. To oznacza, że na jedną przesłanie danych wypada $\sqrt{M}$ operacji arytmetycznych. Zaczynając od jakiejś wartości M procesor nie będzie miał pustych cyklów. Magistral nie będzie zbyt obciążoną, i takie algorytmy demonstrują dobry speed up

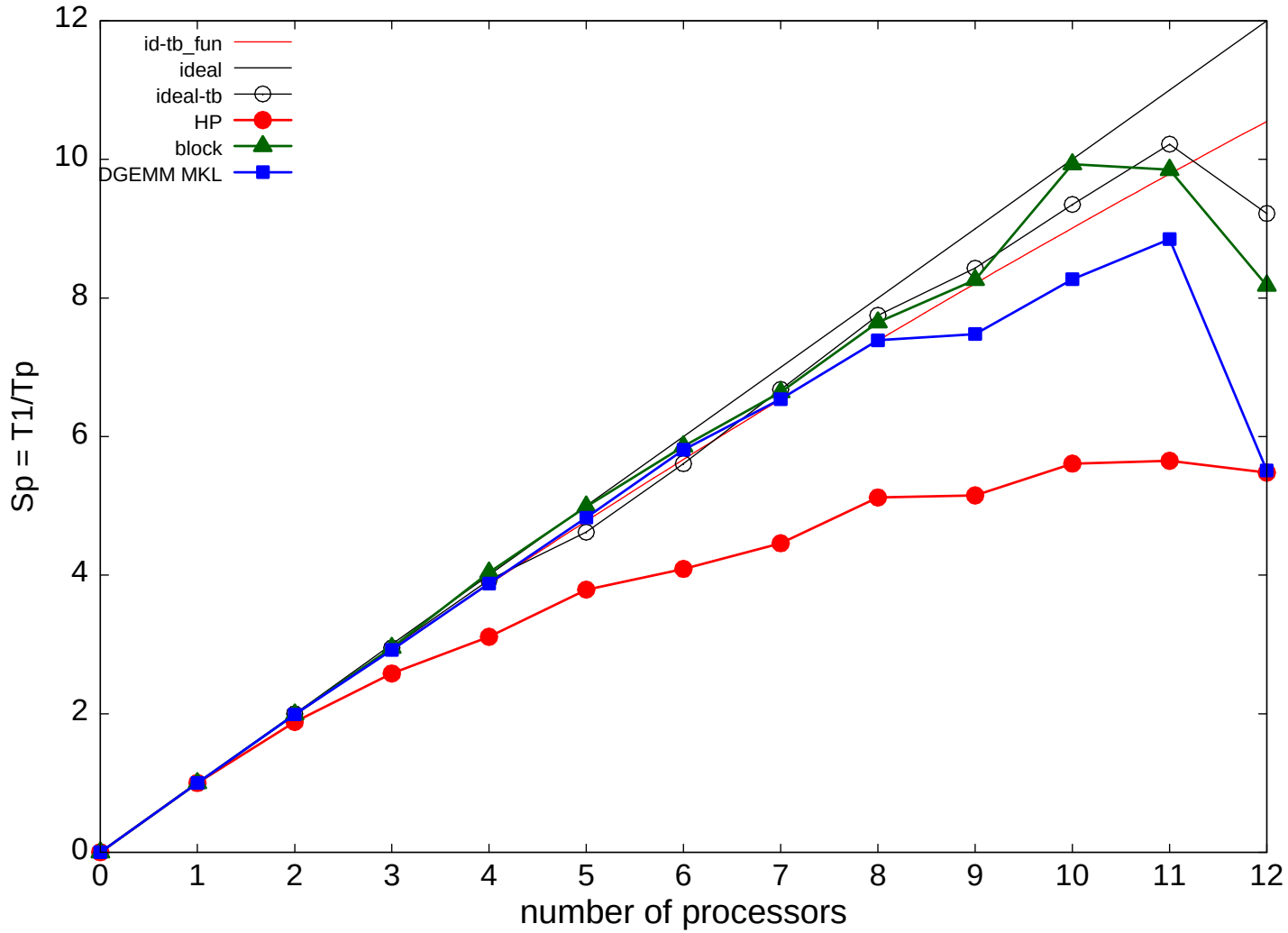
Wydajność (MFLOPS) różnych algorytmów zadania C = C + A*B przy zwiększeniu liczby procesorów (platforma x64, komputer z procesorem CoreQuad):

Algorytm	*np*=1	*np*=2	*np*=3	*np*=4
Naiwny (*ikj*)	1 104	1 532	1 662	1 706
Cache blocking only	1 925	3 818	5 649	6 577
Cache blocking, XMM register's blocking	7 239	13 813	19 026	23 567
DGEMM Intel MKL	8 984	17 792	17 811	29 293



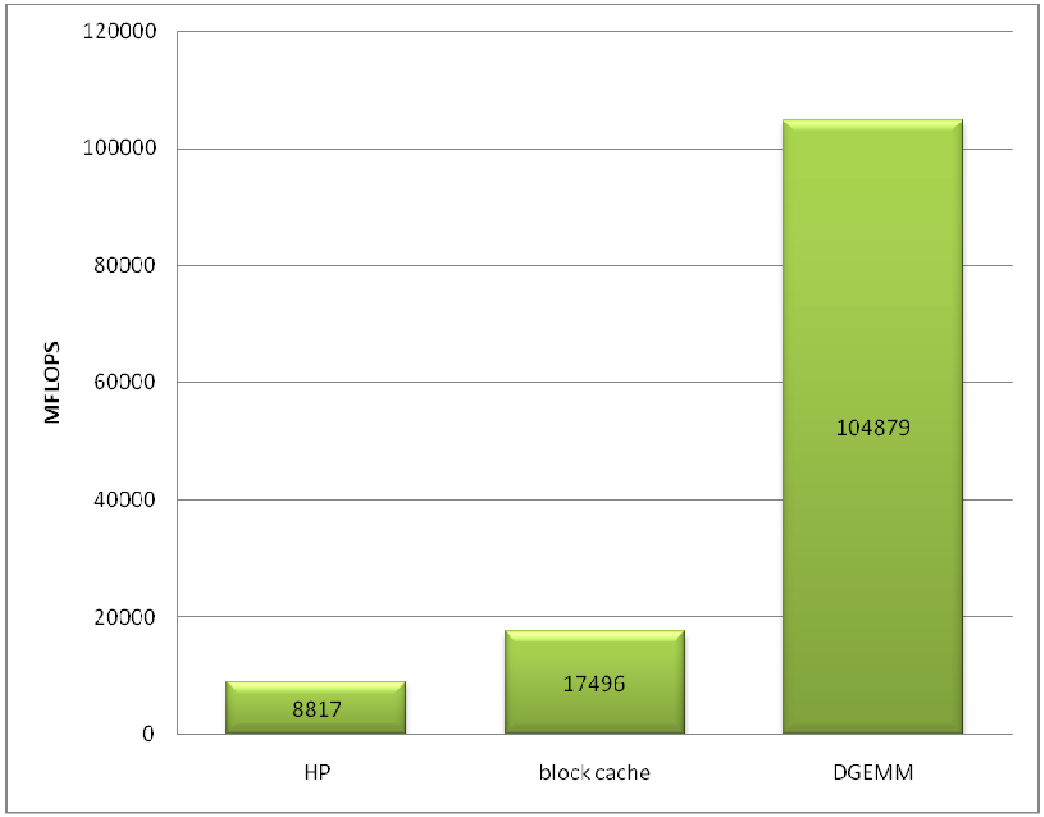
Trace 1 – idealny speed up, trace 2 – algorytm naiwny (*ikj*), trace 3 – cache blocking only, trace 4 – cache blocking, register's blocking

Wyniki testów w trybie wielowątkowym na komputerze klasy workstation z dwoma 6-rdzeniowymi procesorami Intel Xeon X5660 @ 2.8 GHz /3.2 GHz, RAM 24 GB, DDR3, platforma x64.



Wyniki testów w trybie wielowątkowym na komputerze klasy workstation z dwoma 6-rdzeniowymi procesorami Intel Xeon X5660 @ 2.8 GHz /3.2 GHz, RAM 24 GB, DDR3, platforma x64.

Maksymalna wydajność, osiągnięta dla każdej metody na tym komputerze:



Stąd wynika, że przy konstruowaniu algorytmów złożonych wszystko, co nie będzie doprowadzone do algorytmu mnożenia macierzy przez macierz, będzie przyspieszone słabo. Ten wniosek dotyczy algorytmów algebry liniowej dla macierzy gęstych, zrealizowanych dla komputerów wielordzeniowych klasy PC.

- **Mnożenie macierzy przez wektor: y = y + A*x . Jest skalowany dość słabo. Jeśli zadanie obliczeń technicznych może być sformułowane dla pakietu wektorów {x}, {y}, jego speed up i wydajność będzie wielokrotnie zwiększoną. Na przykład, rozwiązanie układu równań liniowych algebraicznych metodą iteracyjną. Często trzeba rozwiązywać to zadanie przy pakiecie prawych stron. Najbardziej pracochłonną procedurą jest mnożenie macierzy przez wektor {y} = {y} + A*{y} , które wielokrotnie się powtarza na każdej iteracji. Tu faktycznie mnożenie macierzy przez wektor pozostało zastąpione mnożeniem macierzy przez macierz.**

- **Rozwiązywanie układów równań liniowych algebraicznych metodą blokową Choleskiego (Gaussa).**

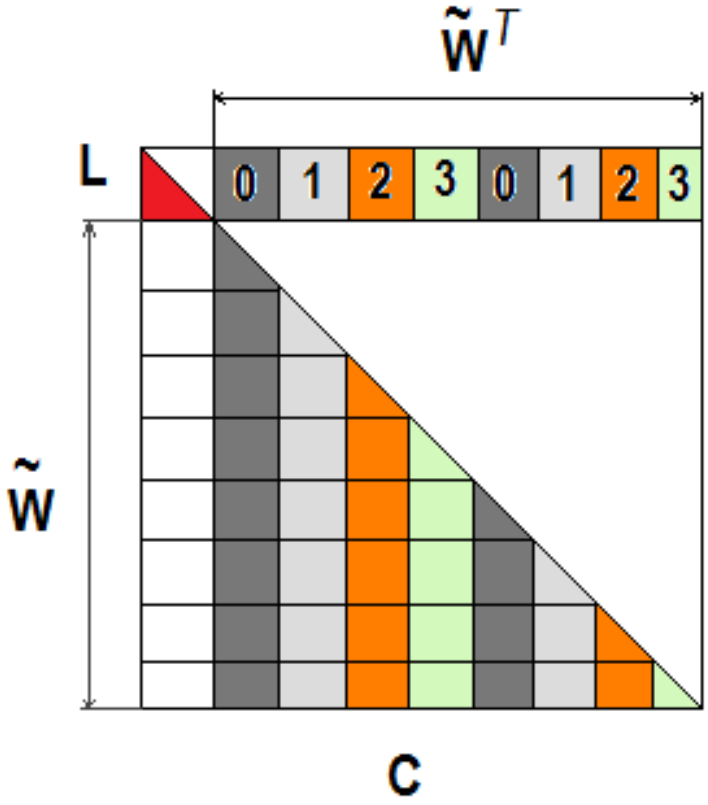
Wydajność faktoryzacji gęstej symetrycznej macierzy blokową metodą Choleskiego. Przy obliczeniu dopełnienia Schura użyto blokowanie rejestrów wektorowych, blokowanie cache L1 i pakowanie danych zgodnie z techniką Intel MKL (Komputer z procesorem CoreQuad)

Platforma	ia32				Intel 64 (x64)			
Liczba procesorów	1	2	3	4	1	2	3	4
MFLOPS	4 459	8 395	11 567	14 299	5 526	10 437	14 559	18 151
$S_p=T_1/T_p$	1	1.88	2.59	3.21	1	1.89	2.63	3.28

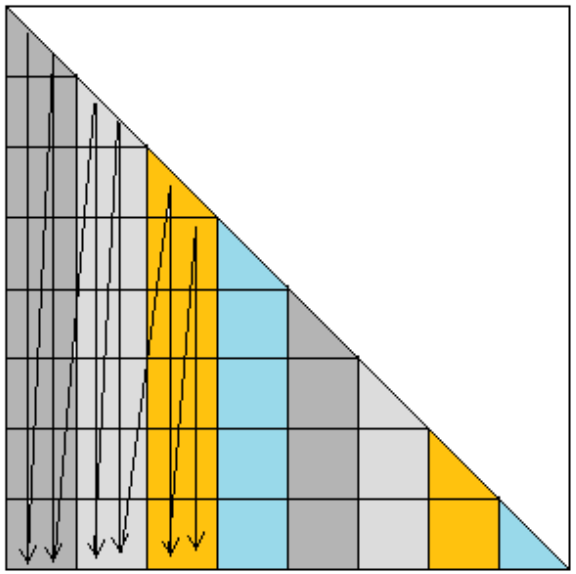
Wydajność faktoryzacji gęstej symetrycznej macierzy blokową metodą Choleskiego. Przy obliczeniu dopełnienia Schura użyto blokowanie tylko cache L1 (Komputer z procesorem CoreQuad)

Platforma	ia32				Intel 64 (x64)			
Liczba procesorów	1	2	3	4	1	2	3	4
MFLOPS	1 575	3 100	4 519	5 231	1 587	3 120	4 599	5 513
$S_p=T_1/T_p$	1	1.97	2.87	3.32	1	1.97	2.90	3.47

Podział danych pomiędzy wątkami



Umieszczenie danych



Krok blokowej faktoryzacji:

$$\mathbf{A} = \mathbf{L} \cdot \mathbf{L}^T \rightarrow \mathbf{L}$$

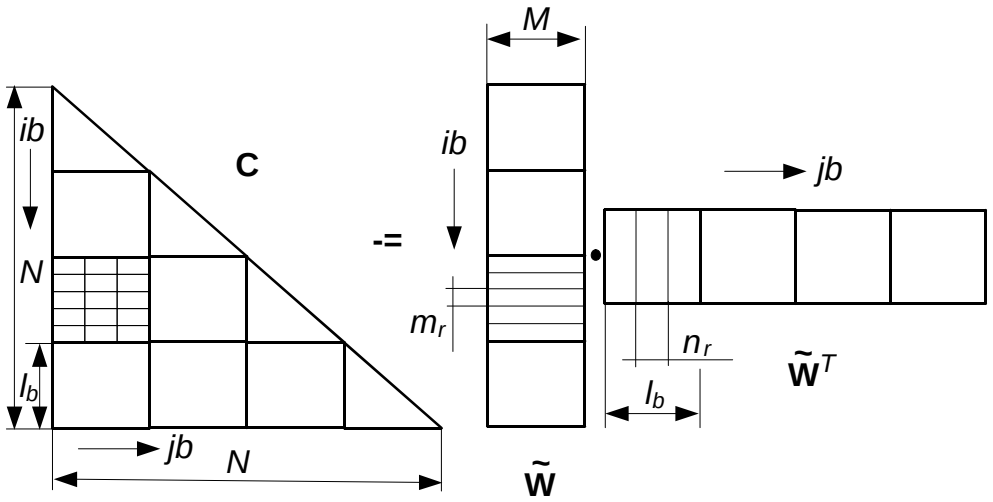
$$\mathbf{L} \cdot \tilde{\mathbf{W}}^T = \mathbf{W}^T \rightarrow \tilde{\mathbf{W}}$$

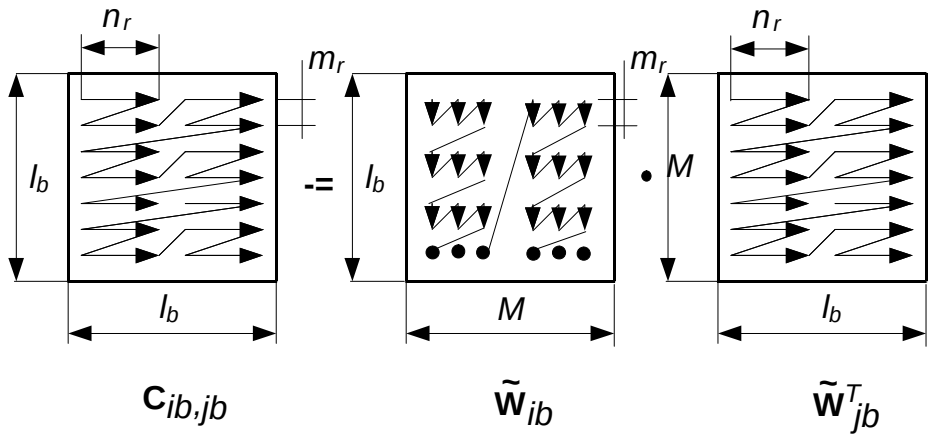
$$\tilde{\mathbf{C}} = \mathbf{C} - \tilde{\mathbf{W}} \cdot \tilde{\mathbf{W}}^T$$

Algorytm obliczenia dopełnienia Schura na poziomie bloków

$$
\begin{aligned}
&// \ Pack \quad \tilde{\mathbf{W}} \\
&\#\ pragma \quad omp \quad parallel \quad for \quad private \ (ib,\ jb)\ scheduler \quad (dynamic \quad) \\
&for\ (\ jb\ =\ 0;\ jb\ <\ Nb\ ;\ jb\ +\ +) \\
&\{ \\
&\quad // \ Pack \qquad \tilde{\mathbf{W}}^{T}_{jb} \\
&\quad for\ (ib\ =\ jb\ ;ib\ <\ Nb\ ;ib\ +\ +) \\
&\quad \{ \\
&\qquad \tilde{\mathbf{C}}_{ib\ ,\ jb}\ =\ \tilde{\mathbf{C}}_{ib\ ,\ jb}\ -\ \tilde{\mathbf{W}}_{ib}\ \cdot\ \tilde{\mathbf{W}}^{T}_{jb} \\
&\quad \} \\
&\}
\end{aligned}
$$

Pakowanie danych dla skutecznego użycia rejestrów wektorowych

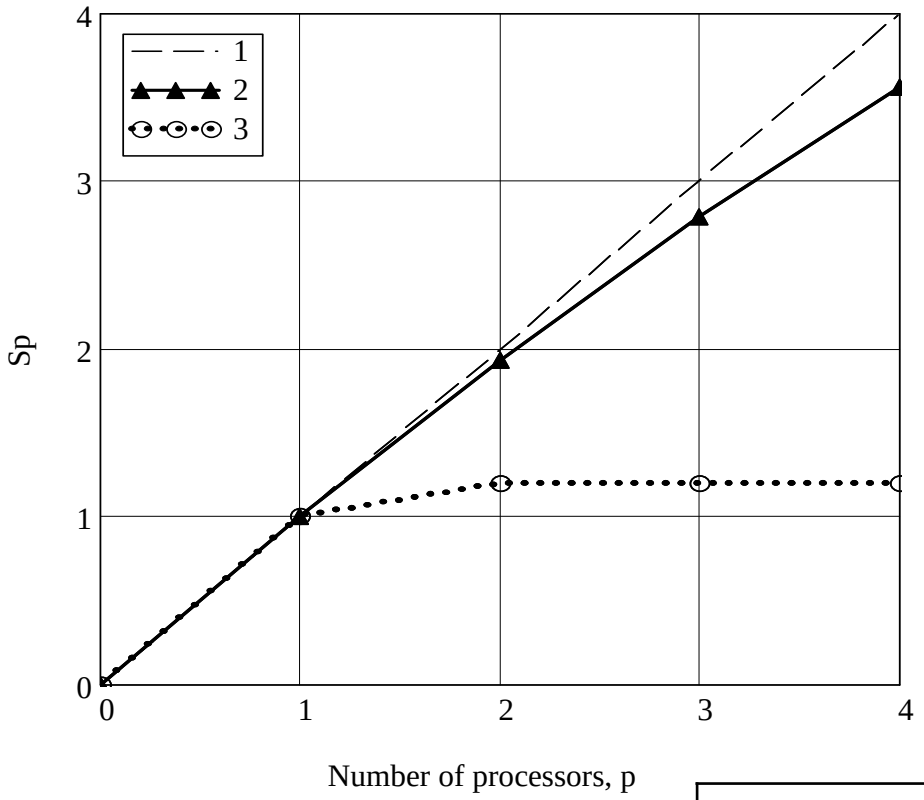




- **Jeśli Wasz komputer zawiera dwa procesory, to nasz program będzie liczył w dwa razy szybciej…**

Sformułujemy to inaczej: czy może speed up występować jako wyczerpująca miara jakości systemu obliczeniowego równoległego?

Przykład: obliczenie iloczynu skalarnego dwóch wektorów: dot = X*Y, rozmiar zadania N = 10 000 000 . Zastosujemy dla rozwiązania Intel Threading Building Blocks, przy czym jeden raz użyjemy dla wektorów typ danych *concurrent_vector*, wprowadzony przez Intel, a drugi raz – typ double.



1 – przyspieszenie idealne
2 – typ concurrent_vector
3 – typ double

Czas wykonania, ms

Ilość procesorów	concurrent _vector	double
1	7 062	1 453
2	3 656	1 218
3	2 531	1 219
4	1 984	1 219

- **Chocza speed up zadania przy użyci typu *concurrent_vector* jest wspaniały, wydajność dla typu *double* na jednym procesorze jest większa od wydajności dla typu *concurrent_vector* na 4 procesorach.**

- **Z tego wynika: speed up zadania nie może być przyjęty jako główna miara jakości systemu obliczeniowego.**