Modelowanie zagadnień technicznych

SKRYPT

Siergiej Fialko

Wydział Fizyki, Matematyki i Informatyki
Politechniki Krakowskiej

Kraków 2011

Siergiej Fialko
Modelowanie zagadnień technicznych.

Niniejszy kurs jest poświęcony typowym problemom oraz podstawowym kierunkom ich rozwiązywania, pojawiającym się przed programistami przy opracowywaniu inżyniersko-technicznych systemów komputerowych. W charakterze przykładów rozpatrzone zostały typowe algorytmy algebry liniowej: obliczenia iloczynu skalarnego dwóch wektorów, mnożenie macierzy przez wektor, macierzy przez macierz, rozwiązywanie układów równań liniowych algebraicznych z macierzami gęstymi oraz rzadkimi symetrycznymi przy zastosowaniu metod bezpośrednich. Szczególną uwagą zostały objęte trzy kierunki: osiągnięcie wysokiej wydajności na każdym z procesorów, osobliwości zrównoleglenia obliczeń na wielojądrowych komputerach PC oraz podstawowe sposoby pracy z macierzami rzadkimi.

Kurs jest przeznaczony dla studentów kierunków technicznych, takich jak informatyka stosowana i matematyka obliczeniowa, może też być pożyteczny dla deweloperów oprogramowania, użytkowników programów oraz pragnących głębiej zrozumieć zasady pracy nowoczesnych systemów komputerowych z dziedziny obliczeń technicznych.

Spis Treści

1. Cele i zadnia kursu

Nazwa „modelowanie zagadnień technicznych” obejmuje wiele pojęć o szerokim znaczeniu. W obrębie tego kursu główną uwagę udzielono osobliwościom modelowania komputerowego zagadnień technicznych, czyli aspektom informatycznym, które powstają przy tworzeniu systemów obliczeniowych w zagadnieniach technicznych.

Celem tego kursu jest zapoznanie się z technikami i metodami opracowania algorytmów, często spotykanych przy modelowaniu zagadnień technicznych, przy czym głównie zwrócono uwagę na podstawy tworzenia takich metod na wielordzeniowych komputerach PC.

W postaci typowych przykładów występują klasyczne zagadnienia algebry liniowej: obliczenie iloczynu skalarnego dwóch wektorów, mnożenie macierzy przez wektor, mnożenie macierzy przez macierz, rozwiązywanie układów równań liniowych algebraicznych z macierzami symetrycznymi gęstymi i rzadkimi przy zastosowaniu współczesnych metod bezpośrednich. Problemy, wynikające przy rozważaniu podanych zadań, są często spotykane w innych branżach zagadnień technicznych. Dlatego osiągnięte umiejętności oraz techniki programowania znacznie przekraczają granicy rozważanych tu problemów.

Kurs obejmuje:

- Osiągnięcie wysokiej wydajności typowych algorytmów obliczeniowych zagadnień technicznych przy programowaniu sekwencyjnym oraz wielowątkowym dla komputerów wielordzeniowych (SMP – symmetrical multiprocessing).
- Typowe techniki programowania wielowątkowego dla algorytmów obliczeniowych.
- Algorytmy dla macierzy rzadkich: formaty skompresowane, uporządkowanie w celu zmniejszenia ilości zapełnień przy faktoryzacji, same solwery – metody faktoryzacji.

Kurs zawiera elementy matematyki – algebry liniowej, algebry macierzy, teorii grafów, metod numerycznych jak również wymaga wiedzy z podstaw oprogramowania w języku C, C++, systemów operacyjnych i oprogramowania wielowątkowego.

Przy przedstawianiu twierdzeń matematycznych, położeń i wniosków uwaga jest głównie skupiona nie na ścisłości dowodów, jak jest to przyjęte w kursach, ukierunkowanych na matematyków, ale na algorytmach i właściwościach, niezbędnych dla realizacji komputerowej.

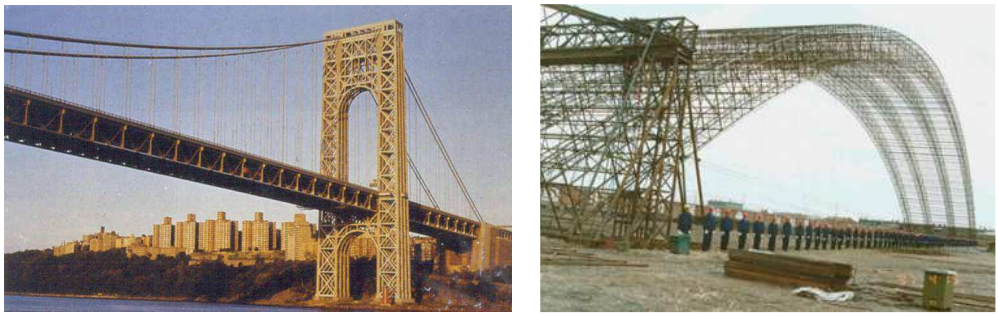
Przy pisaniu kursu autor kierował się doświadczeniem, nabytym w firmach informatycznych RoboBAT (www.robobat.com) i SCAD Soft

(www.scadsoft.com), w których opracowywał oprogramowanie z dziedziny metody elementów skończonych i innych metod obliczeniowych.

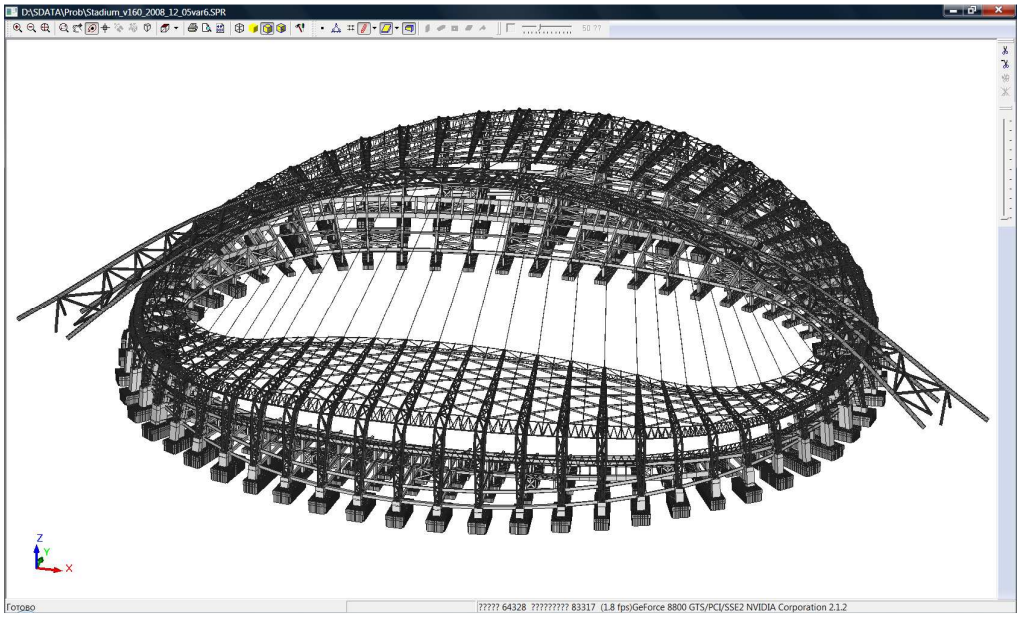
Modelowanie zagadnień technicznych znajduje się na przecięciu metod numerycznych i informatyki. Często zadania techniczne doprowadzają do rozwiązywania zadań numerycznych dużego rozmiaru – powstają układy równań liniowych i nieliniowych algebraicznych, zagadnienia wartości własnych, obliczenia numeryczne całek, całkowanie zagadnień Cauchy itd. Ilość współczesnych modeli różnorodnych obiektów technicznych może wynosić od kilkaset tysięcy do kilka milionów równań. Jest jedna z cech tego typu zadań, przy czym użytkownik najczęściej pragnie rozwiązywać te zadania na komputerach typu PC .

Typowe przykłady takich zadań – widoki i modele różnych konstrukcji – są przedstawione na rys. 1.1 – 1.5.

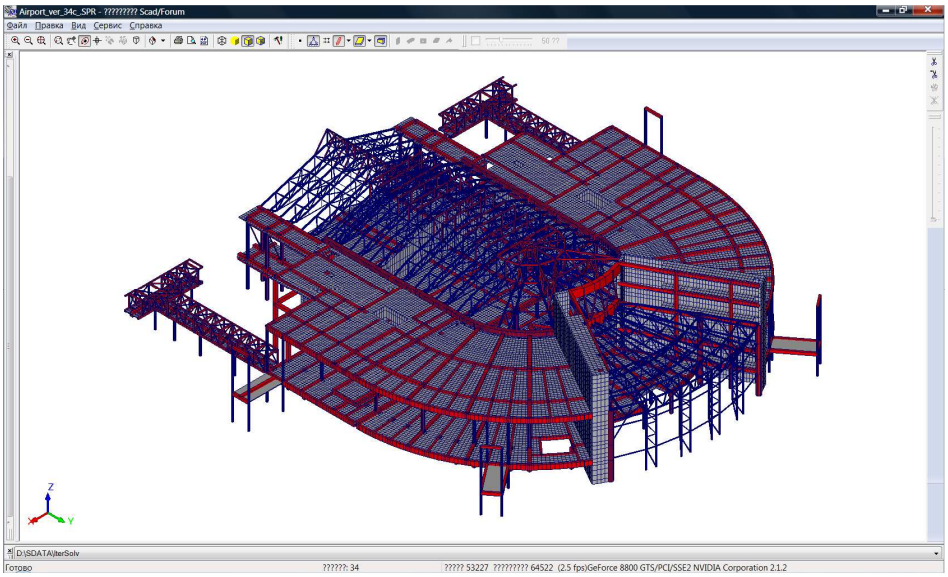
Rys. 1.1 Budynki wielopiętrowe



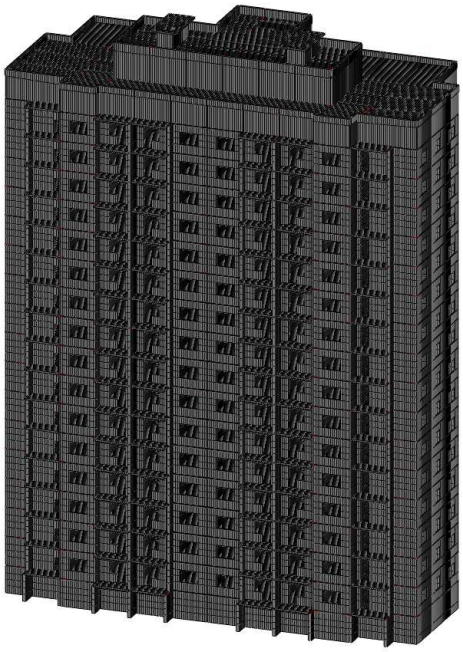
Rys. 1.2 Konstrukcji inżynierskie



Rys. 1.3 Model MES stadionu w Wilnie (62 298 równań nieliniowych algebraicznych)



Rys. 1.4 Model MES terminalu lotniska w Wilnie (315 397 równań nieliniowych algebraicznych)



Rys. 1.5 Model budynku wielopiętrowego z bloków betonowych
(3 198 609 równań liniowych algebraicznych)

Przy opracowaniu oprogramowania, przeznaczonego dla modelowania zagadnień technicznych, zwrócimy uwagę na najważniejsze momenty:

- Zastosowanie niezawodnych i optymalnych metod i algorytmów matematycznych.
- Skuteczna praca ze wszystkimi poziomami hierarchii pamięci – pamięciom główną, pamięciom podręczną, rejestrami procesora.
- Zrównoleglenie operacji na różnych etapach realizacji algorytmów.

1.1 Podstawowe cechy komputerów wielordzeniowych

W tym kursie dla porównania wydajności różnych algorytmów będziemy używać miarę wydajności FLOPS (Floating Point Operations Per Second) – ilość operacji zmiennoprzecinkowych wykonanych za sekundę. Jest to naturalna miara wydajności dla algorytmów, które powstają w zagadnieniach naukowo-technicznych, gdzie dominantowymi są operacji arytmetyczne zmiennoprzecinkowe. Przecież ta miara nie jest przydatna do oceny wydajności innych rodzajów algorytmów – kompilatorów, baz danych, itp. Dla algorytmów obliczeniowych, wykonanych na współczesnych komputerach PC, łatwo stosować MFLOPS и GFLOPS – odpowiednio 10^6 FLOPS oraz 10^9 FLOPS.

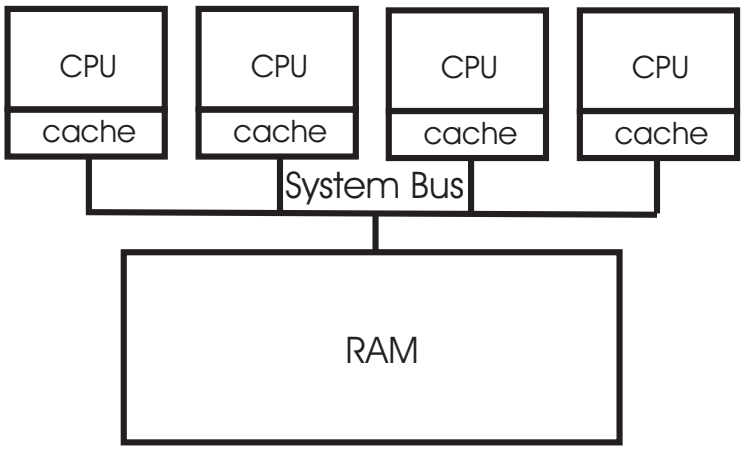
Różne operacje zmiennoprzecinkowe mają różną trwałość. Operacje zmiennoprzecinkowe mogą być szybkie i wolne. Przybliżony stosunek trwałości różnych operacji zmiennoprzecinkowych jest podany w tab. 1.

Tabela 1.

Przybliżony stosunek trwałości operacji zmiennoprzecinkowych

dodawanie, odejmowanie, porównanie, mnożenie	1
dzielenie, obliczenie pierwiastka kwadratowego	4
eksponenta, sinus, ...	8

Będziemy rozważać tylko komputery z pamięcią wspólną. Jest to taka architektura systemów komputerowych równoległych kiedy kilka jednakowych procesorów współpracują z tą samą pamięcią główną (rys. 1.6) . Taką architekturę nazywają UMA – Uniform Memory Access.



Rys. 1.6 Wieloprocesorowe systemy z jednorodnym dostępem do pamięci – UMA (Uniform Memory Access)

Każdy z procesorów (CPU) może mieć swoją własną pamięć podręczną (cache) i łączą się z pamięcią główną (RAM) poprzez magistrale (system bus).

Współczesne komputery PC wielordzeniowe należą do takiej architektury, ponieważ kilka takich samych procesorów (rdzeni) współpracują z jedną wspólną pamięcią główną i są sterowane jednym egzemplarzem systemu operacyjnego. Taka architektura dostała nazwę SMP (symmetrical multiprocessing).

Procesory mają dostęp równoprawny (każdy z procesorów nie ma żadnej przewagi nad innymi) i równoznaczny (czas dostępu do dowolnej części pamięci jest taki sam).

Przy tym związek między procesami i synchronizacja są proste, oprogramowanie mniej skomplikowane, niż w wypadku architektury z pamięcią rozproszoną. Jednak taka architektura ma podstawowe ograniczenia na ilość procesorów – krytyczna jest przepustowość magistrali.

Na dzień dzisiejszy coraz więcej zadań technicznych mogą być rozwiązane na komputerach klasy PC, moc obliczeniowa których ciągłe rośnie. Takie komputery są tanie, i większość użytkowników z małych i średnich biur projektowych preferuje używać właśnie takie komputery i bardzo niechętnie podejmują decyzję o przejściu na klastery, sieci komputerowe i potężne stacji robocze. Doświadczony projektant najczęściej znajdzie sposób zmniejszyć rozmiar zadania obliczeniowego tak, aby to zadanie zmieściło się w komputerze klasy PC.

Komputery klasy PC mają ograniczony rozmiar pamięci głównej i stosunkowo nie wysoką przepustowość magistrali. Ta cecha często wymaga specjalnych podejść przy tworzeniu algorytmów i oprogramowania. Uwaga tego kursu została skupiona głównie na podstawach tworzenia oprogramowania dla opisanej klasy komputerów.

1.2 Potokowe przetwarzanie danych a techniki oprogramowania.

W nowoczesnych mikroprocesorach szeroko stosuje się potokowe przetwarzanie danych. Przy tym cały proces jest podzielony na nieduże części – stopnie przebiegu, każda z których jest realizowana przez poszczególne urządzenia fizyczne. Obróbkę każdej maszynowej komendy można rozłożyć na kilka stopni – etapów. Przy tym dane są przekazywane od poprzedniego etapu do następnego.

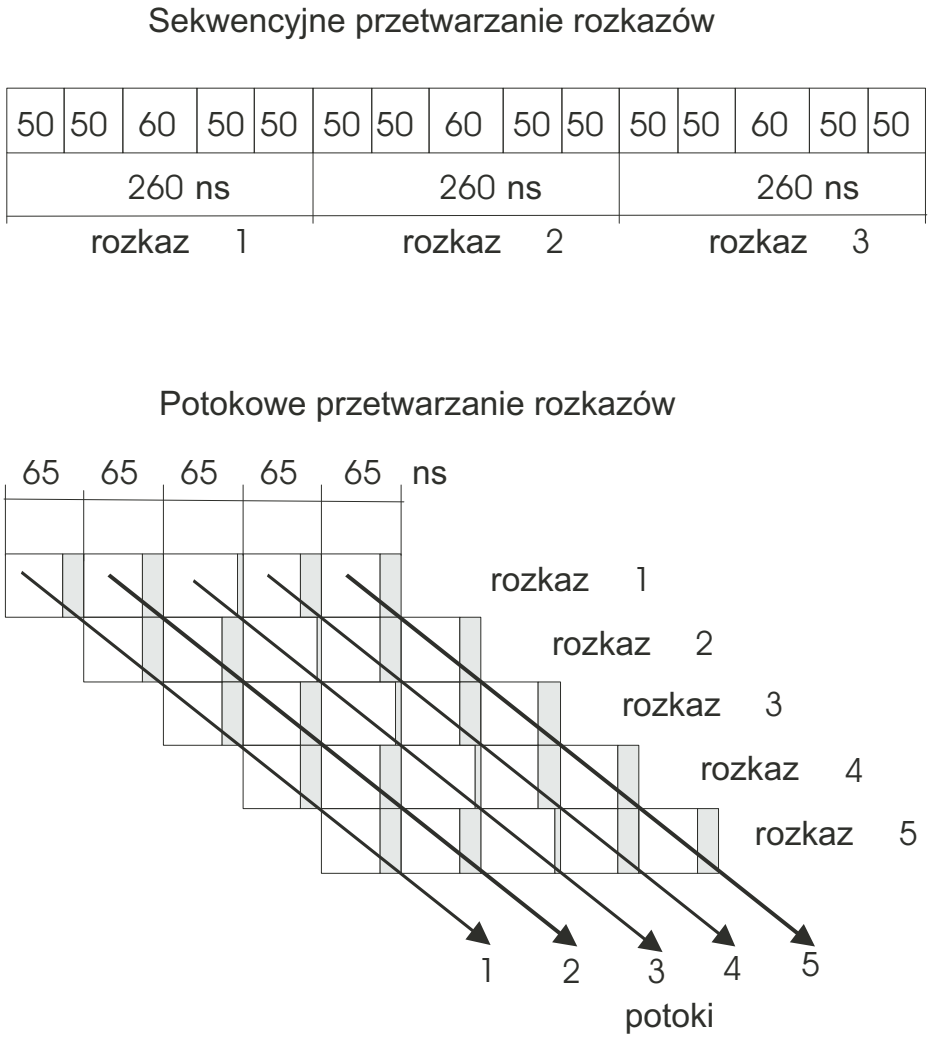
Na przykład [Шагин, 2003], mamy 5 etapów wykonania operacji trwałością 50, 50, 60, 50 i 50 ns odpowiednio (rys. 1.7), a 5 ns – nakład na organizacje przetwarzania potokowego. Średni czas wykonania polecenia przy przetwarzaniu sekwencyjnym wynosi 260 ns, a przy przetwarzani potokowym (t_{sr}^{pot}) – trwałości najdłuższego taktu plus nakład na przetwarzanie potokowe – 60+5 = 65 ns [t_{sr}^{pot} = (maksymalna trwałość etapu)*(ilość etapów)/(ilość potoków) = 65*5/5 = 65 ns]. Przyspieszenie przy przetwarzani potokowym w stosunku do przetwarzania sekwencyjnego wynosi 260/65 = 4 razy.

Struktura polecenia i ilość etapów zależą od rodzaju polecenia. W podanym przykładzie każde polecenie składa się z pięciu etapów. Mogą to być IF – obliczenie adresu polecenia, pobranie polecenia, ID – dekodowanie polecenia, OF – obliczenie adresu operandu, pobranie operandu, EX – wykonanie operacji na operandach, WB – zapis wyniku.

Przy przetwarzaniu potokowym pierwszy potok wykonuje pierwszy etap pierwszego polecenia (IF). Dalej działają dwa potoki. Pierwszy – IF dla drugiego rozkazu, a drugi – ID dla pierwszego polecenia. Dalej pierwszy potok – IF dla trzeciego polecenia, drugi – ID dla drugiego polecenia i trzeci – OF dla pierwszego polecenia. I tak dalej. Każdy potok przetwarza tylko swój etap i jest wykonywany tylko odpowiednią częścią sprzętu. Po wciągnięciu wszystkich potoków w dowolnym momencie czasu jednocześnie działają pięć potoków, w skutek czego i powstaje zrównoleglenie przy potokowym przetwarzani danych.

W odróżnieniu od wielowątkowości takie zrównoleglenie nie wymaga dużej pracy od programisty i dlatego jest bardzo wygodne.

Typowymi fragmentami kodu, gdzie przetwarzanie potokowe jest bardzo skuteczne są długie pętle *for*, *while*, *do – while*, w których są wykonane duże ciągi dokładnie takich samych instrukcji.



Rys. 1.7 Sekwencyjne i potokowe przetwarzanie danych. Czas jałowy dla każdego etapu oraz czas wykorzystany na wyrównanie trwałości etapów jest oznaczony kolorem szarym

W pewnych sytuacjach mogą powstawać zaburzenia potokowego przetwarzania. Rozważmy kilka typowych przykładów.

Przykład 1.

```
for(i=0; i<N; i++)
{
    A[i] = 1.1* A[i];
}
```

Potok, pobierający wartość operandu A[…] (etap OF) na iteracji *i+1* , będzie wstrzymany ponieważ adres operandu nie jest znany dopóki licznik iteracji pętli *i* nie pozostanie inkrementowany. Rozwiązanie:

```
for(i=0; i<N; i+=2)
{
    A[i]   =  1.1*A[i];
    A[i+1] =  1.1*A[i+1];
}
```

Taka technika otrzymała nazwę rozwijania pętli. Instrukcji w pętli są niezależne – dlatego jeden potok procesora jest w stanie pobierać adres operandu drugiej instrukcji nie oczekując na zakończenie wykonania przez inne potoki pierwszej instrukcji, czyli potoki są w stanie działać równoległe. Rozwijanie pętli wspiera potokowe przetwarzanie danych, usuwając zaburzenia. Współczesne kompilatory C/C++ w wersji release przy ustawieniu flagi optymalizacji /O2 (optymalizacja z celem podniesienia wydajności) samodzielnie kilkakrotnie rozwijają pętle. Jest to wykonane dla standardowych fragmentów kodu. Dla kodów skomplikowanych możliwa taka sytuacja że programista ręcznie będzie rozwijał pętle. W szczególności dotyczy to technik SSE2, SSE3 [SSE2].

Przykład 2 [Касперский, 2003].

```
while(next = p[next])
{
    //ciało pętli
    next++;
}
```

Dopóki nie zostanie wykonana ostatnia linia instrukcji pętli, dotąd procesor nie będzie znał zmiennej *next* i nie jest w stanie użyć *prefetch* – pobieranie danych z wyprzedzeniem.

Czas wykonania tego kodu jest wyznaczony latentością układu pamięci – odcinkiem czasu od wystawienia przez procesor zapotrzebowania na chipset do umieszczenia odpowiednich danych w CPU. Rozwiązanie:

```
while(next = p[next++])
{
    //ciało pętli
}
```

Procesor wysyła do chipsetu zapotrzebowanie na ładowanie słowa *p[next]* i natychmiast inkrementuje *next*. Adres następnego słowa już jest znany. Dalej

procesor wysyła kolejne zapotrzebowanie, nie oczekując na operację w pętli. Dopóki procesor liczy, chipset będzie dostarczał nową porcje danych dla kolejnej iteracji pętli.

Czas ładowania do rejestru N zależnych słów wynosi $T = N\cdot(T_{ch}+T_{mem})$, gdzie T_{ch} – latentność chipseta, T_{mem} – latentność pamięci. Czas ładowania N niezależnych słów wynosi $T = N/C+T_{ch}+T_{mem}$, gdzie C – przepustowość układu pamięci – objętość danych, odczytywanych z pamięci do CPU za jednostkę czasu.

Przykład 3 [Касперский, 2003].

```
for(i=0; i<N; i++)
{
    A[i] = A[i]+B[i];    //line 1
    C[i] = A[i]+1.0;     //line 2
    A[i] = D[i]+1.0;     //line 3
}
```

Linia kodu 3 nie może być wykonana dopóki nie będzie zakończona linia 2. Inaczej A[i] będzie zmodyfikowane wcześniej od jego użycia w linii 2. Rozwiązanie:

```
register double rrr;
for(i=0; i<N; i++)
{
    A[i] = A[i]+B[i];    //line 1
    rrr  = A[i];
    C[i] = rrr +1.0;     //line 2
    A[i] = D[i]+1.0;     //line 3
}
```

Rozważmy test, kod którego jest umieszczony w załączniku 1. Rozwiązujemy zadanie $dot = \mathbf{x}^T\cdot\mathbf{x}$, gdzie $\mathbf{x}$ – wektor o rozmiarze N . Rozmiar N jest automatycznie zmieniony tak żeby być wielokrotnym w stosunku do rozmiaru pamięci podręcznej L1, przedstawionej w ilości słów *double*. Dla komputera, na którym to zadanie było testowane, rozmiar L1 wynosił 32 KB albo 4096 słów typu *double*. Program reprezentuje kilka różnych metod. Pierwsza metoda (rys. 1.8) w pełni odpowiada podejściu klasycznemu. Takie podejścia we współczesnej literaturze otrzymało nazwę metod naiwnych, ponieważ kod komputerowy odpowiada algorytmu matematycznemu. Takie metody nie biorą pod uwagę szczególne cechy struktury pamięci komputera i dlatego często demonstrują bardzo niską wydajność.

```
t_s = GetTickCount();
for(it=0; it<ntimes; it++)
{
      dot = 0.0;
      for(i=0; i<N; i++)
      {
            dot += X[i]*X[i];
      }
}
t_elaps = (double)(GetTickCount()-t_s);
```

Rys. 1.8 Metoda klasyczna (naiwna)

Współczesne komputery bardzo szybko obliczają takie zadania. Dla tego żeby zapewnić poprawność pomiaru czasu obliczeń, trzeba zwiększyć trwałość wykonania tego zadania – powtarzamy *ntimes* razy jego wykonanie. Funkcja platformy Win32 GetTickCount() zwraca czas w *ms* od czasu wystartowania systemu operacyjnego [MSDN], *t_elaps* –czas wykonania tego fragmentu kodu.

Druga i trzecia metody używają czterokrotne i ośmiokrotne rozwijanie pętli (rys. 1.9 – 1.10).

```
for(it=0; it<ntimes; it++)
{
      dot = 0.0;
      for(i=0; i<N; i+=4)
      {
            dot += X[i]*X[i]+X[i+1]*X[i+1]+
                   X[i+2]*X[i+2]+X[i+3]*X[i+3];
      }
}
```

Rys. 1.9 Czterokrotne rozwijanie pętli

```
for(it=0; it<ntimes; it++)
{
      dot = 0.0;
      for(i=0; i<N; i+=8)
      {
            dot += X[i]*X[i]+X[i+1]*X[i+1]+X[i+2]*X[i+2]+
                   X[i+3]*X[i+3]+X[i+4]*X[i+4]+X[i+5]*X[i+5]+
                   X[i+6]*X[i+6]+X[i+7]*X[i+7];
      }
}
```

Rys. 1.10 Ośmiokrotne rozwijanie pętli

Czwarta metoda stosuje ośmiokrotne rozwijanie pętli oraz pobieranie danych z wyprzedzeniem (prefetch – rys. 1.11)

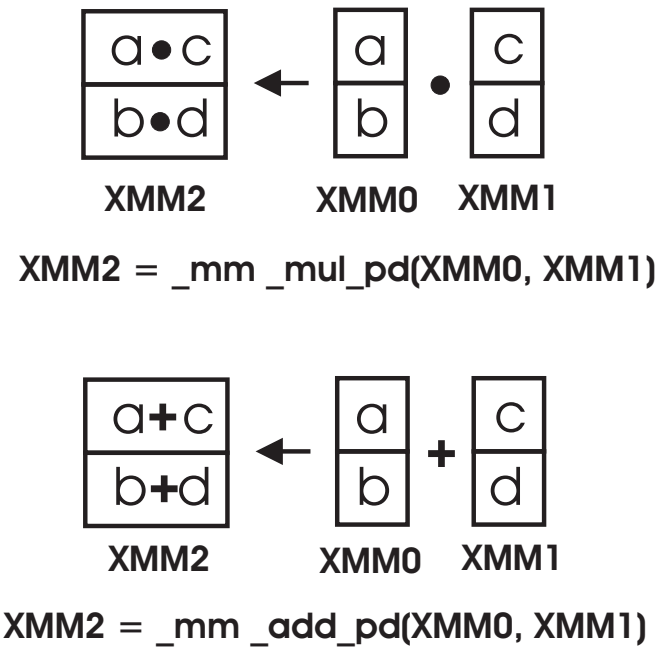
```
for(it=0; it<ntimes; it++)
{
      dot1 = 0.0;
      for(i=0; i<N; i+=8)
      {
          _mm_prefetch((const char *)(&X[i+8]), _MM_HINT_T0);
          dot1 += X[i]*X[i]+X[i+1]*X[i+1]+X[i+2]*X[i+2]+
                  X[i+3]*X[i+3]+X[i+4]*X[i+4]+X[i+5]*X[i+5]+
                  X[i+6]*X[i+6]+X[i+7]*X[i+7];
      }
}
```

Rys. 1.11 Ośmiokrotne rozwijanie pętli oraz wykonanie prefetch'u

W tym kodzie instrukcja *prefetch*(...) jest poleceniem dla układu pamięci o pobieraniu 8 słów *double* , które będą potrzebne przy następnej iteracji pętli, i umieszczeniu ich w pamięci podręcznej procesora. Podczas gdy procesor wykonuje instrukcje iteracji bieżącej, układ pamięci niezależnie przygotowuje dane dla następnej iteracji.

Ostatnia metoda używa 128-bitowe XMM rejestry, jakie mogą zmieścić 2 słowa *double* i za jeden cykl procesora wykonać dwa mnożenia lub dwa dodawania. Dla słów typu *float* (*integer*) w XMM rejestrze można zmieścić 4 słowa i wykonać w jednym cyklu procesora odpowiednio 4 mnożenia lub 4 dodawania. Współczesne procesory komputerów klasy PC mają 8 dostępnych rejestrów XMM na platformie 32-bitowej (ia32) oraz 16 dostępnych XMM rejestrów na platformie 64-bitowej (×64) . Na podstawie techniki SSE2 [SSE2] w językach C/C++ istnieje możliwość programowania na rejestrach XMM, przy czym kod jest bliski do języka assemblera. Schematyczne operacje mnożenia i dodawania, wykonane na rejestrach XMM, są podane na rys. 1.12.

Liczby *a*, *b* oraz *c*, *d* są ładowane do rejestrów XMM0, XMM1, przy czym *a*, *c* znajdują się w górnej części każdego rejestru, natomiast *b*, *d* – w dolnej. Wynik będzie umieszczony w rejestrze XMM2. Odpowiedni fragment kodu jest przedstawiony na rys. 1.13.



Rys. 1.12 Operacje mnożenia i dodawania na rejestrach XMM dla liczb typu *double*

Deklarujemy pięć zmiennych typu *__m128d* o nazwach *c1 – c4, sum*, jakie nadają możliwość stosowania instrukcji SSE2, SSE3. Żeby udostępnić prototypy instrukcji SSE2, SSE3, trzeba dołożyć nagłówek *#include <emmintrin.h>* . Dyrektywa *__declspec(align(16))* oznacza że tablica *res[2]* będzie wyrównana w granicach 16 bajtów. Jest to koniecznie dla instrukcji wyładowania danych z XMM rejestru *_mm_store_pd*. Pierwsza pętle obejmuje ilość powtarzani algorytmu. Przed uruchomieniem drugiej pętli wyzerowujemy rejestr *sum* instrukcją *_mm_setzero_pd* . Wykonujemy prefetch z krokiem 8 słów typu *double* i ładujemy rejestry *c1 – c4* instrukcją *_mm_load_pd* . Każda taka instrukcja ładuje w rejestr XMM dwa sąsiednie elementy tablicy X. Dlatego pamięć dla tablicy X jest alokowana funkcją *_aligned_malloc* przy wyrównaniu na granice 16 bajtów (załacznik 1). Zawartość rejestru *c1* mnożymy przez siebie i wynik umieszczamy do rejestru *c1*. Do rejestru *sum* dodajemy zawartość rejestru *c1*. I tak dalej. Po zakończeniu wykonania pętli wewnętrznej w górnej części rejestru *sum* znajduje się suma kwadratów parzystych elementów tablicy X $res[0] = \sum_{i=0}^{N/2} x_{2i}^2$, a w dolnej –

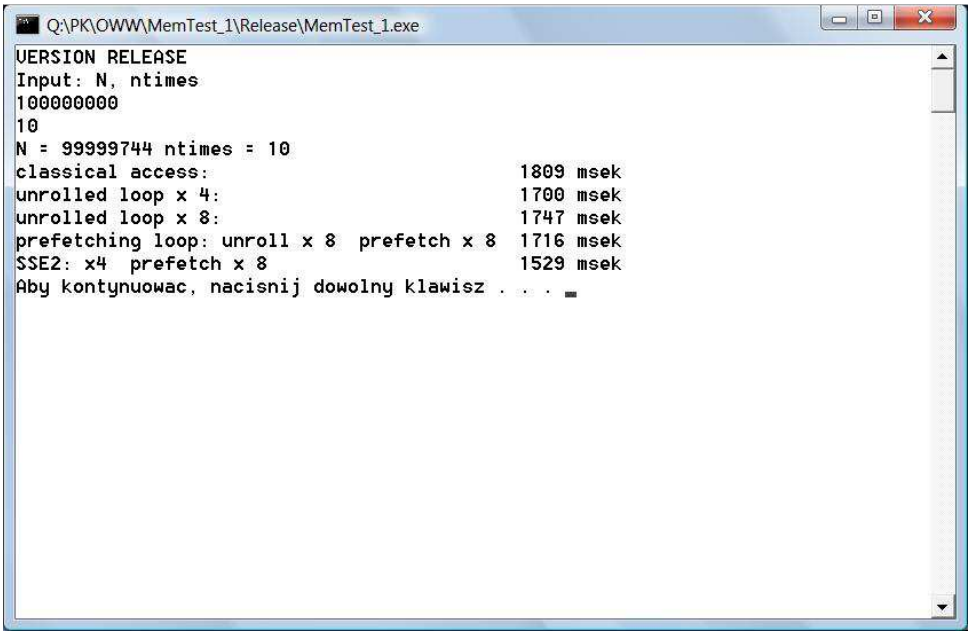
suma kwadratów nieparzystych elementów $res[1] = \sum_{i=1}^{N/2} x_{2i-1}^2$. Wyładowujemy rejestr *sum* do tablicy *res* i dodajemy te elementy.

```
__m128d c1, c2, c3, c4, sum;
__declspec(align(16)) double res[2];

t_s = GetTickCount();
//number of double words in cache line 64B is 8
for(it=0; it<ntimes; it++)
{
      sum = _mm_setzero_pd();

      for(i=0; i<N; i+=8)
      {
         _mm_prefetch((const char *)(&X[i+8]), _MM_HINT_T0);
         c1 = _mm_load_pd(&X[i]);  //load X[i], X[i+1] to c1
         c2 = _mm_load_pd(&X[i+2]);//load X[i+2],X[i+3] to c2
         c3 = _mm_load_pd(&X[i+4]);//load X[i+4],X[i+5] to c3
         c4 = _mm_load_pd(&X[i+6]);//load X[i+6],X[i+7] to c3
         c1 = _mm_mul_pd(c1, c1);  //c1 <- c1*c1
         sum = _mm_add_pd(sum, c1);//sum = sum + c1*c1
         c2 = _mm_mul_pd(c2, c2);  //c2 <- c2*c2
         sum = _mm_add_pd(sum, c2);//sum = sum + c2*c2
         c3 = _mm_mul_pd(c3, c3);  //c3 <- c3*c3
         sum = _mm_add_pd(sum, c3);//sum = sum + c3*c3
         c4 = _mm_mul_pd(c4, c4);  //c4 <- c4*c4
         sum = _mm_add_pd(sum, c4);//sum = sum + c4*c4
      }
      _mm_store_pd (res, sum);      //unload res <- sum
      dot1 = res[0]+res[1];         //dot1 = res[0]+res[1];
}
t_elaps = (double)(GetTickCount()-t_s);
```

Rys. 1.13 Użycie rejestrów XMM dla algorytmu *dot* = $\mathbf{x}^T \cdot \mathbf{x}$

Wyniki są przedstawione na rys. 1.14 .

Testy są wykonane na komputerze z procesorem Intel® Core™2 Quad CPU Q6600 @2.40 GHz, cache L1: 4x32 KB, L2: 4096 KB, RAM DDR2 333.7 MHz, 8 GB, chipset Intel P35/G33/G31, OS Windows Vista™ Business (64-bit), Service Pack 2.



Rys. 1.14 Wyniki testowania zadania *dot* = $\mathbf{x}^T \cdot \mathbf{x}$ na komputerze z procesorem Intel® Core™2 Quad CPU Q6600 @2.40 GHz

1.3 Pobieranie danych z wyprzedzeniem

Pobieranie danych z wyprzedzeniem (prefetch) jest wykonywany z celą ukryć zwłokę (latentność) układu pamięci [Grama A., Gupta A., Karypis G., Kumar V., 2003], [Касперский, 2003]. Typowy schemat bez prefetch'u:

```
for(i=0; i<N; i++)
    dot += X[i]*X[i];
```

Taki program działa tak: przy i = 0 procesor wystawia na chipset zapotrzebowanie na słowo X[0]. Chipset umieszcza to w kolejkę zapotrzebowań. Jak tylko opracowanie tej kolejki dojdzie do naszego zapotrzebowania, podane słowo będzie odczytane z pamięci głównej i za pomocą magistrali przesunięte do pamięci podręcznej L2. Procesor w tym czasie wykonuje puste cykle. Odcinek czasu licząc od momentu wystawienia zapotrzebowania na chipset do umieszczenia danych w pamięci podręcznej jest latentnością układu pamięć – chipset ($T_{ch}+T_{mem}$). Na następnej iteracji i = 1 i wszystko będzie powtórzone dokładnie tak, jak dla poprzedniej iteracji.

Czas wykonania takiego algorytmu : $T = N\cdot(T_{ch}+T_{mem})+2\cdot N\cdot t_{ops} \approx N\cdot(T_{ch}+T_{mem})$, gdzie t_{ops} – czas wykonania jednego mnożenia lub dodawania, $2\cdot N\cdot t_{ops} << N\cdot(T_{ch}+T_{mem})$.

Typowy schemat pętli z prefetch'em:

```
for(i=0; i<N; i+=8)
{
    //inicjujemy ładowanie następnej linii cache danych za
    //adresem &X[i+8] do cache L1. Przy niemożliwości
    //ładowania cache L1 będzie ładowany cache L2.
    //Zakładamy, że rozmiar cache linii dla podanego
    //sprzętu wynosi 64 B - 8 słów double. Ten kod jest
    //zależny od sprzętu.

    _mm_prefetch((const char *)(&X[i+8]), _MM_HINT_T0);
    dot += X[i]*X[i]+X[i+1]*X[i+1] +…+X[i+7]*X[i+7];
}
```

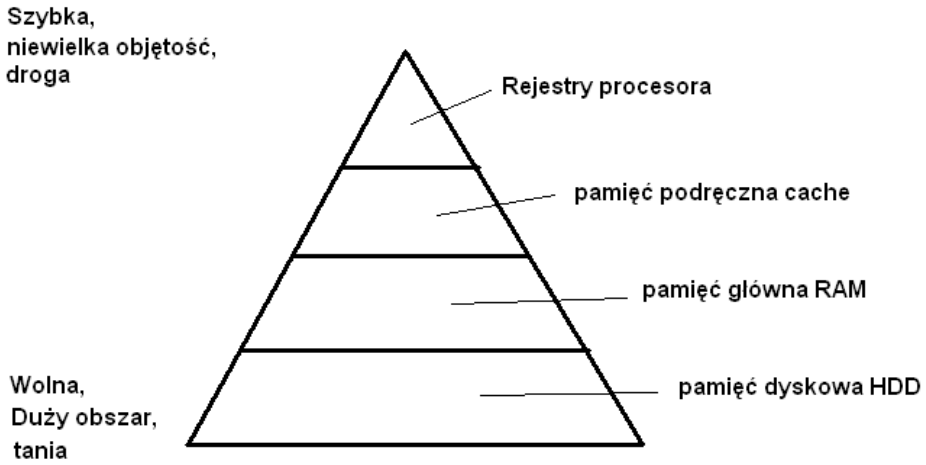
Teraz oczekiwanie procesora w skutek latentności układu pamięci odbywa się tylko przy $i = 0$. Dalej procesor i układ pamięci działają równoległe. Przyczyną hamowania obliczeń dla takiego kodu jest ograniczona przepustowość magistrali, a nie latentność układu pamięci.

Czas wykonania podanego algorytmu wynosi $T = N/C + T_{ch}+T_{mem} + 2\cdot N\cdot t_{ops}/k \approx N/C$, gdzie C – przepustowość układu pamięci, k – przyspieszenie obliczeń w skutek usunięcia zaburzeń potokowego przetwarzania danych przy rozwijaniu pętli.

Rozróżniają prefetch hardwarowy i softwarowy [Prefetch, 2011]. Prefetch hardwarowy dostarcza 64 bajtów (rozmiar cache-linii) za jeden raz. Kiedy prefetch hardwarowy wyznacza dostęp do linii cache *l*, za którą następuje linia cache *l*+1, on zacznie inicjowanie prefetch'u do kolejnej linii cache. Prefetch hardwarowy zawiera szereg ograniczeń. On nie może przecinać granicy stron pamięci, wypełnia tylko linie cache L2, nie może wyznaczyć dostęp do danych, które są pobierane z dużymi skokami. Tę ograniczenia mogą być pokonane poprzez użycie prefetch'u softwarowego, jaki może wypełniać linii cache L1 lub L2 (L3), jeśli poziom cache L1 już jest wypełniony. Może być wykonany tak zwany non-temporal prefetch, który wypełnia cache L1, a wyciśnięcie tych danych jest wykonywane do pamięci głównej, a nie do cache'u niskiego poziomu. O wyborze typu prefetch'u softwarowego decyduje drygi argument instrukcji prefetch.

1.4 Struktura pamięci komputera. Cache konflikty

Najczęściej hierarchie pamięci współczesnego komputera jest przedstawiona w postaci piramidy (rys. 1.15, [Demmel J. W., 1997]).



Rys. 1.15 Hierarchia pamięci współczesnego komputera

Najszybsza pamięć (najdroższa, ma niewielką objętość) jest umieszczona z góry (wierzchołek piramidy), a najwolniejsza (tania, ma stosownie dużą objętość) jest z dołu.

Przykładowe rozmiary i czasy dostępu dla różnych rodzajów pamięci są podane w tabeli 1.2.

Tabela 1.2.

Przykładowe rozmiary i czasy dostępu dla różnych rodzajów pamięci

Typ pamięci	rozmiar	Czas dostępu, ns
rejestry	Kilkaset B	< 1 ns
podręczna (L1)	kilkanaście KB	Kilka ns
podręczna (L2 - L3)	Kilka MB	Kilkanaście ns
główna	Kilkaset MB	sto kilkadziesiąt ns

Gdyby pamięć komputera byłaby tak jednorodna i tak samo szybka, jak rejestry procesora, to znaczna część materiału, przedstawianego w danym opracowaniu, utraciłaby sens. Dodatkowo koszt takiego hipotetycznego komputera byłby wręcz niebotyczny. Oprócz tego, szybka pamięć (rejestry, pamięć podręczna) jest zgromadzona na elementach półprzewodników, które z kolei dość intensywnie wydzielają ciepło. Dlatego problem odprowadzania ciepła jest obecnie jedną z najmocniejszych przeszkód, stojących na drodze ku powiększeniu pojemności szybkiej pamięci.

Z innej strony, istnienie hierarchicznej struktury pamięci, której poziomy znacznie różnią się szybkością dostępu do danych, rodzi cały szereg problemów przy przejściu z jednego poziomu na inny. Pierwsza część naszego opracowania będzie poświęcona właśnie tym problemom.

W ramach danego kursu, za wyjątkiem ostatniego rozdziału, będą rozpatrzone zadania, które mieszczą się tylko w pamięci głównej (RAM) komputera.

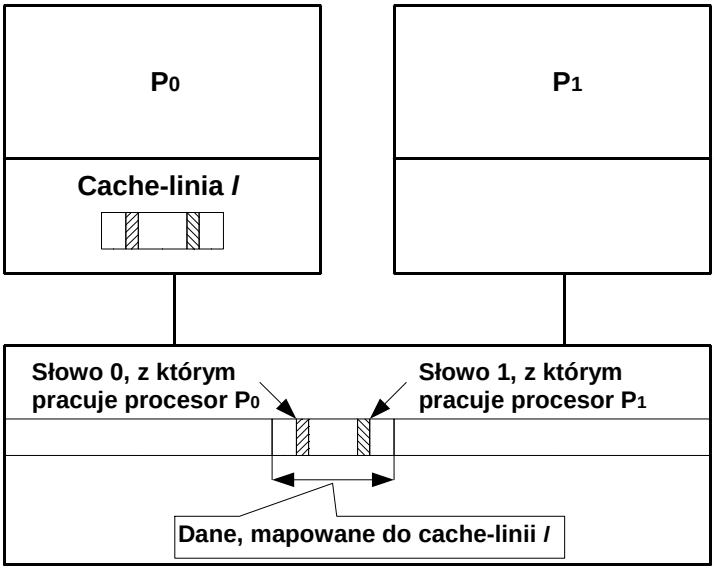
Optymalne wykorzystanie pamięci podręcznej jest bardzo ważne dla wydajności programów. W zależności od typu procesora pamięć podręczna składa się z linii o kilku słów o rozmiarze 32B, 64B, 128B. Dane z pamięci głównej będą umieszczone w pamięci podręcznej przed umieszczeniem w rejestrach procesora. Mówimy o odwzorowaniu bloków pamięci głównej w linie pamięci podręcznej (cache-linie).

Przy czytaniu danych (słowo, bajt) procesor najpierw sprawdza stan swojej pamięci podręcznej. Jeśli te dane są w pamięci podręcznej (nazwiemy tą sytuacje read hit – trafienie), oni będą umieszczone w rejestry procesora. Jeśli tych danych (słowa, bajta) niema w pamięci podręcznej (read miss – chybienie), będzie przeładowana cała linia pamięci podręcznej [Касперский, 2003]. Przy pobraniu tych danych (bajta, słowa) z pamięci głównej będą pobrane nie tylko to słowo, bajt, a i bajty sąsiednie tak, żeby wypełnić całą linie pamięci podręcznej. Dane, które będą umieszczone w cache-linie, muszą być wyrównane po granicach, wielokrotnych 32B, 64B, 128B odpowiednio.

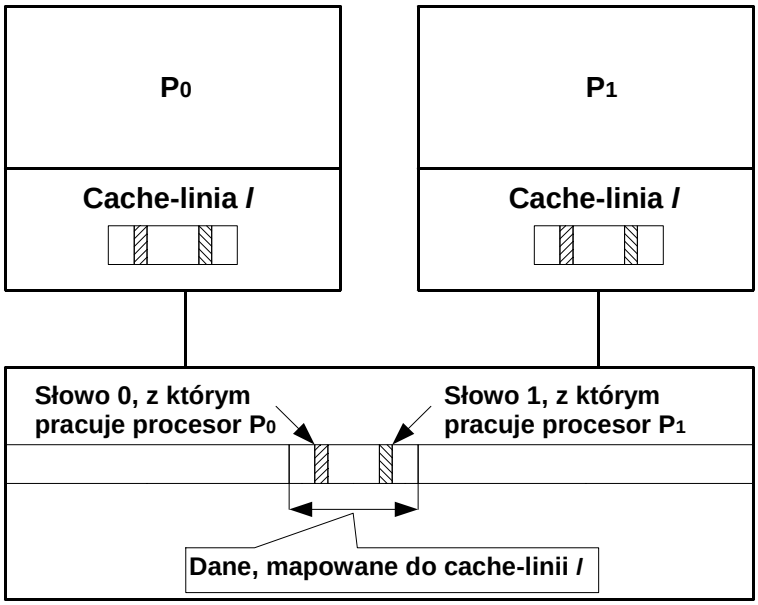
Ideologia cache-linii polega na tym, że zadania najczęściej pracują ze zbiorem sąsiednich słów (bajtów), więc przy odczycie tych słów (bajtów) często wynika stan read hit, i procesor pobiera te dane w rejestry z cache-linii, a nie przez wolny kanał magistral – pamięć główna. Przecież cache-linii istotnie komplikują odnowienie pamięci w środowisku wieloprocesorowym z pamięcią wspólną

Rozważmy taki przykład [Richter, 1997]:

- Procesor P0 odczytuje słowo 0, a również i słowa sąsiedni w swój cache (rys. 1.16).
- Procesor P1 odczytuje słowo 1, a również i słowa sąsiedni w swój cache, przy czym słowo 0 procesora P0 w skutek wyrównania po granicach, wielokrotnych rozmiary cache-linii, znajduje się w cache-linii procesora P1 też (rys. 1.17).

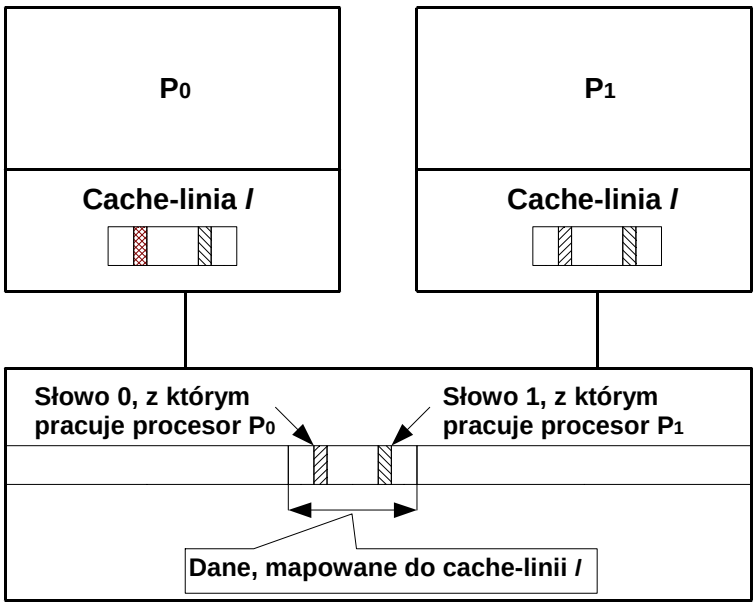


Rys. 1.16 Procesor P0 odczytuje słowo 0, a również i słowa sąsiedni w swój cache



Rys. 1.17 Procesor P1 odczytuje słowo 1 oraz słowa sąsiadujące do swojego cache'u, przy tym słowo 0 procesora P0

- Procesor P0 modyfikuje słowo 0 i umieszcza ten wynik do swojego cache'u. Wartość tego słowa w pamięci głównej na razie pozostaje bez zmian (rys. 1.18). Procesor P1 nic nie wie o tym, że zawartość słowa 1 uległa zmianie. Teraz wartość słowa 0 zależy od tego, cache którego procesora będzie wyładowany w pamięć główną później.



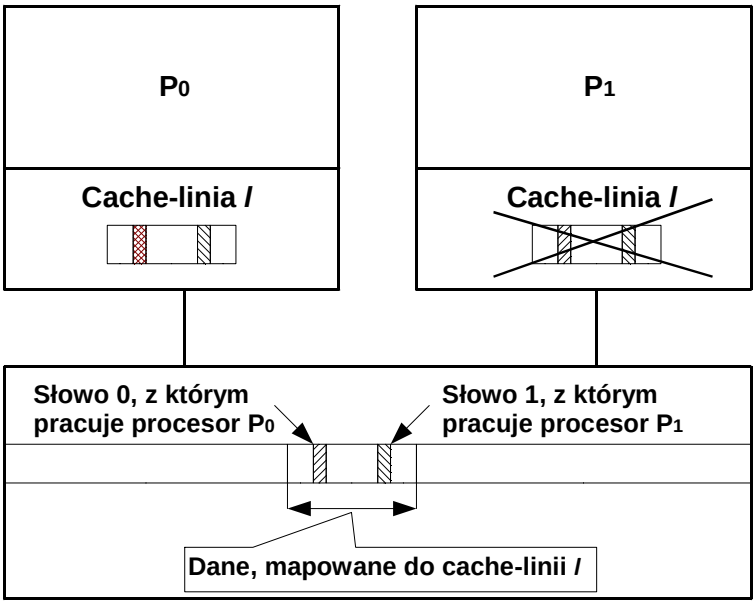
Rys. 1.18 Procesor P0 modyfikuje słowo 0 i umieszcza ten wynik do swojego cache'u. Wartość tego słowa w pamięci głównej na razie pozostaje bez zmian

Taki scenariusz był by prawdziwą katastrofą. Producenci sprzętu komputerowego bardzo dobrze o tym wiedzą, i dlatego w rzeczywistości komputer nigdy tak nie działa.

Działanie komputera wygląda następująco.

- Procesor P0 odczytuje słowo 0, oraz słowa sąsiadujące do swojego cache'u (rys. 1.16).
- Procesor P1 odczytuje słowo 1, także słowa sąsiadujące do swojego cache'u przy czym słowo 0 procesora P0 w skutek wyrównania do granic, wielokrotnych rozmiarów cache-linii, też znajduje się w cache-linii procesora P1 (rys. 1.17).

- Procesor P0 modyfikuje słowo 0 i umieszcza ten wynik w swoim cache'u (rys. 1.18). Wartość tego słowa w pamięci głównej na razie pozostaje bez zmian, ale procesor P1 oraz wszystkie inne procesory, w cache'u, których okazało się słowo 0, dokonuje snooping – śledzenia za stanem słów zmodyfikowanych, i zgłaszają zawartość swojego cache jako unieważnioną (rys. 1.19).

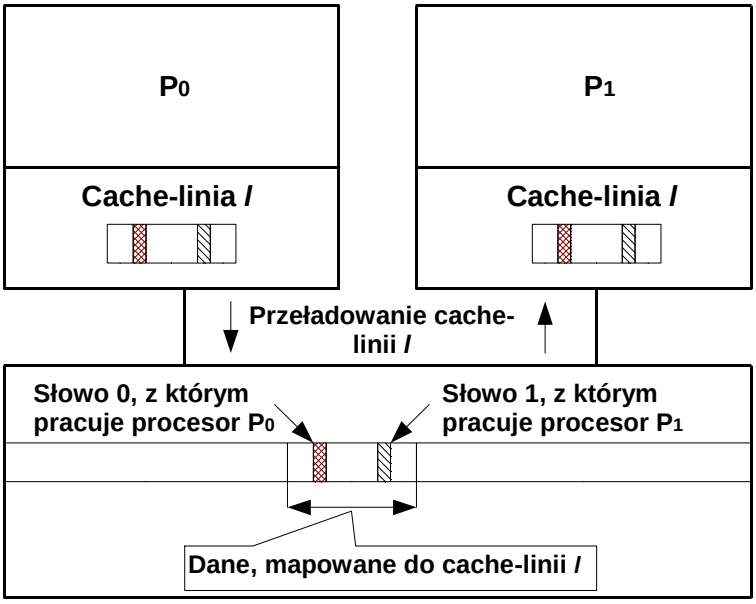


Rys. 1.19 Procesor P0 modyfikuje słowo 0 i umieszcza ten wynik w swoim cache'u. Wartość tego słowa w pamięci głównej na razie pozostaje bez zmian, ale procesor P1 w skutek snooping'u zgłasza zawartość swojego cache jako unieważnioną

- Procesor P0 wyładuje zawartość swojego cache do pamięci głównej, a procesor P1 ponownie załadowuje swoją cache-linie (rys.1.20). Teraz dane w pamięci głównej i w cache-liniach procesorów P0, P1 są spójne (koherentne). Przecież płacą za tą spójność jest nadliczbowe przeładowania cache-linie procesorów P0, P1. Będziemy nazywać taką sytuacje cache-konfliktami.

Okazuje się, że obecność pamięci podręcznej, przeznaczonej dla przyspieszenia obliczeń wskutek zmniejszenia ilości odczytów-zapisów bezpośrednich z pamięci głównej (do pamięci głównej), może stać się przyczyną istotnego zmniejszenia wydajności obliczeń, jeśli dane nie będą wyrównane z uwzględnieniem powyżej

opisanej specyfiki pracy z cache-liniami. Obowiązek, związany z wyrównaniem danych do granic cache-linii, spoczywa na programiście.



Rys. 1.20 Procesor P0 wyładuje zawartość swojego cache do pamięci głównej, a procesor P1 ponownie załadowuje swoją cache-linie. Teraz dane w pamięci głównej i w cache-liniach procesorów P0, P1 są spójne (koherentne).

Dla uniknięcia konfliktów w pamięci podręcznej programiści zwykłe używają następujących zasobów.

- Rozdzielenie danych służących tylko dla odczytu od danych przeznaczonych dla odczytu i zapisu lub tylko do zapisu. Przy danych, które są tylko do odczytu, konflikty w pamięci podręcznej nie wynikają.
- Użycie pamięci lokalnej dla każdego wątku. Przy tworzeniu wątków system operacyjny tworzy osobny stos każdego wątku, przy czym stosy różnych wątków automatycznie będą wyrównane do granic cache-linii.
- Dane rozmieszczone w pamięci głównej są wyrównane tak, żeby różne procesory mieli dostęp do różnych adresów pamięci, rozdzielonej przynajmniej granicą cache-linii.

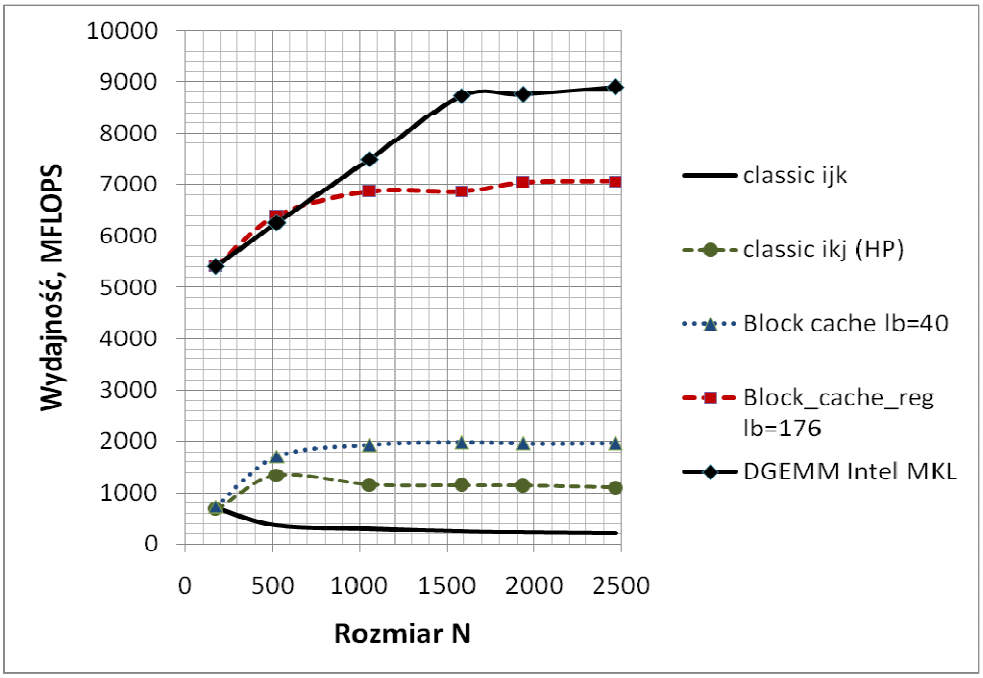
2. Osiągnięcie wysokiej wydajności algorytmów obliczeniowych na jednym procesorze

2.1 Oszacowanie wydajności podstawowych algorytmów algebry liniowej

Będziemy rozważać następujące algorytmy algebry liniowej:

- Mnożenie wektora przez skalar saxpy: $\mathbf{y} = \mathbf{y} + \alpha \cdot \mathbf{x}$.
- Mnożenie macierzy przez wektor: $\mathbf{y} = \mathbf{y} + \mathbf{A} \cdot \mathbf{x}$.
- Mnożenie macierzy przez macierz: $\mathbf{C} = \mathbf{C} + \mathbf{A} \cdot \mathbf{B}$.

Najpierw rozważymy wyniki działalności programu testowego Mulm, opracowanego przez autora, który wykonuje obliczenia według różnych algorytmów mnożenia macierzy przez macierz. Macierzy są kwadratowe o rozmiarze N×N. Wyniki są przedstawione na rys. 2.1.



Rys. 2.1 Wydajność różnych metod mnożenia macierzy przez macierz. Komputer z procesorem Intel® Core™2 Quad CPU Q6600 @2.40 GHz

Przy takim samym rozmiarze zadania *N* każda z prezentowanych metod wykonuje dokładnie tą samą ilość operacji arytmetycznych. Przy tym wydajność tych metod istotnie się różni. Dla dość dużego rozmiaru zadania wydajność każdej metody praktycznie nie zależy od rozmiaru zadania *N*.

Pierwszą metodę – klasyczną, można znaleźć w dowolnym poradniku matematycznym (kolejność umieszczenia pętli – i, j, k [Golub, 1996]). Druga metoda – też klasyczna – ma kolejność pętli i, k, j. Trzecia metoda polega na podziale macierzy na odnośnie niewielkie bloki kwadratowe, przy czym w pamięci podręcznej L1 jednocześnie są umieszczone trzy bloki – po jednym z każdej macierzy. Będziemy nazywać tą metodę block cache, a rozmiar bloku oznaczymy jako l_b. Czwarta metoda pozostała wykonana w technice Intel Math Kernel Library (Intel MKL) [Goto K, Van De Geijn R A (2008)]– biblioteki wysokiej wydajności Intela [Intel MKL, 2011], przy czym autor opracował swoje własne mikrojądro [Fialko S., (2009), CT] – kod niskiego poziomu, napisany na podstawie technik SSE2, SSE3. Będziemy oznaczać tą metodę jako block_cache_reg. I ostatnia metoda – DGEMM (Double General Matrix Multiplication) – jest pobrana z Intel MKL w wersji 10.2.2.025.

Testy są wykonane na komputerze z procesorem Intel® Core™2 Quad CPU Q6600 @2.40 GHz, charakterystyki którego zostały podane wyżej.

Powstaje pytanie: dla czego przy dokładnie takiej samej ilości operacji arytmetycznych, wykonanych każdą metodą dla podanego rozmiaru zadania *N*, wydajność każdej z tych metod jest różna? Jak trzeba tworzyć programy, zęby pracować na wysokim poziomie wydajności?

Odpowiedzi na te pytania został poświęcony dany rozdział.

2.1.1. Podstawowe założenia i modele pamięci

Zaczniemy od teoretycznego oszacowania wydajności podstawowych algorytmów algebry liniowej. W tym celu wprowadzimy następujące założenia, na podstawie, których będziemy wykonywać takie oszacowania.

- Odczyt i zapis (cache ↔ RAM) i operacje arytmetyczne są wykonywane w czasie sekwencyjnym (rezygnujemy z możliwości układu pamięci pracować niezależnie w czasie od procesora centralnego CPU).
- Pamięć podręczna (cache) jest rozdzieloną pomiędzy danymi w najbardziej skuteczny sposób (procesor i system operacyjny są „intelektualne” – dane załadowane w cache tak, żeby zminimalizować ilość transferów cache ↔ RAM).
- Komputer ma tylko rejestry (najbardziej szybką pamięć), jeden poziom cache (szybka pamięć – rezygnujemy z hierarchicznej struktury pamięci podręcznej) i

pamięć główną, dostęp do adresów której jest wykonany z szybkością magistrali [Demmel J. W., 1997].

- Wszystkie dane są umieszczone w pamięci głównej (RAM).

Oznaczymy n – rozmiar zadania, M – rozmiar pamięci podręcznej, przy czym $M << n$. Czas wykonania programu można oszacować:

$$t = f \cdot t_{arith} + m \cdot t_{bus} = f \cdot t_{arith} \cdot \left(1 + \frac{m}{f} \cdot \frac{t_{bus}}{t_{arith}}\right), \tag{2.1}$$

gdzie f, m – ilość operacji arytmetycznych i ilość transferów danych cache ↔ RAM, t_{arith}, t_{bus} – trwanie jednej operacji arytmetycznej i przesłania jednego słowa danych. Zakładamy, że wszystkie operacje arytmetyczne mają jednakowy czas trwania, wartości t_{arith}, t_{bus} nie zależą od algorytmu i kodu programu (zależą tylko od sprzętu), a algorytm dla wykonania danego zadania z punktu widzenia ustawienia matematycznego jest optymalny. Ostatnie założenie oznacza, że dla danego algorytmu ilość operacji arytmetycznych f nie da się zmniejszyć. W taki sposób, pozostaje jedyna możliwość zmniejszenia czasu obliczeń t – zmniejszyć ilość transferów danych m.

Na podstawie przyjętych założeń czas obliczeń t w (2.1) zależy tylko od stosunku m/f. Wprowadźmy wskaźnik wydajności

$$q = \frac{f}{m} \quad . \tag{2.2}$$

Ta charakterystyka będzie służyć dla oszacowania wydajności algorytmów.

Będziemy przechowywać macierze w tablicach jednowymiarowych. Przy tym macierz można umieścić w taką tablicę wiersz po wierszu, albo kolumna po kolumnie. Będziemy umieszczać macierze w jednowymiarowej tablice wiersz po wierszu

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix},$$

gdzie liczby 1, 2, 3, ... oznaczają kolejność umieszczenia elementów macierzy w tablice jednowymiarowej.

Umieszczenie wektorów w tablice jednowymiarowej jest takie

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \rightarrow \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}.$$

Umieszczenie macierzy blokowej wiersz po wierszu jest podane dla przykładu 4×4 i rozmiarów bloku 2×2:

$$\begin{pmatrix} \{a\}_{1,1} & \{a\}_{1,2} \\ \{a\}_{2,1} & \{a\}_{2,2} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & : & a_{13} & a_{14} \\ a_{21} & a_{22} & : & a_{23} & a_{24} \\ . & . & . & . & . \\ a_{31} & a_{32} & : & a_{33} & a_{34} \\ a_{41} & a_{42} & : & a_{43} & a_{44} \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 2 & : & 3 & 4 \\ 5 & 6 & : & 7 & 8 \\ . & . & . & . & . \\ 9 & 10 & : & 11 & 12 \\ 13 & 14 & : & 15 & 16 \end{pmatrix}.$$

2.1.2 Algorytm y = y + α·x

Zaczniemy od prostego algorytmu saxpy mnożenia wektora przez skalar.

```
//odczyt α i umieszczenie w rejestrze
for(i=1; i<=n; i++)
{
    //odczyt yi, xi
    yi = yi + α *xi;
    //zapis yi
}
```

Za jedną iteracje taki algorytm wykonuje dwie operacje arytmetyczne – jedno mnożenie i jedno dodawanie. Ilość iteracji jest równa *n*. Stąd wynika że $f = 2n$.

Zmienna α nie zależy od indeksu iteracji pętli, dlatego najskuteczniej umieścić α w rejestrze i utrzymywać tam przy wykonaniu iteracji żeby uniknąć niepotrzebnych ładowań rejestru przy każdej iteracji. Przy każdej iteracji trzeba jeden raz odczytać y_i i jeden raz – x_i. Po wykonaniu obliczeń zmodyfikowany element y_i powinien być zapisany do pamięci RAM. W trakcie wykonania całego algorytmu trzeba wykonać $2n$ odczytów i n zapisów. Ilość transferów danych wynosi $m = 3n+1$, a wskaźnik wydajności jest równy

$$q = \frac{f}{m} = \frac{2n}{3n+1} = \frac{2}{3+\frac{1}{n}} \approx \frac{2}{3}, \tag{2.3}$$

gdzie dla dużych *n* wartością 1/*n* można pominąć w stosunku do 3 .

W taki sposób, wskaźnik wydajności dla algorytmu saxpy wynosi 2/3 .

2.1.3 Algorytm y = y+A·x

Rozważmy teraz algorytm mnożenia macierzy przez wektor. Diagram algorytmu przedstawimy w postaci

$$\begin{array}{c} \rightarrow j \\ \downarrow \\ i \end{array} \underbrace{\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}}_{\mathbf{A}} \bullet \begin{array}{c} \downarrow \\ j \end{array} \underbrace{\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}}_{\mathbf{x}} \rightarrow \begin{array}{c} \downarrow \\ i \end{array} \underbrace{\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}}_{\mathbf{y}}, \tag{2.4}$$

gdzie liczby 1, 2, 3, ... oznaczają kolejność umieszczenia elementów macierzy oraz wektorów w tablicach jednowymiarowych. Kierunki indeksów są oznaczone strzałkami. Algorytm jest podany niżej.

```
for(i=1; i<=n; i++)
{
    //odczyt yi  i umieszczenie w rejestr
    for(j=1; j<=n; j++)
     {
          //odczyt aij , xj
          yi = yi + aij*xj;
    }
    //zapis yi
}
```

W tym algorytmie dla każdego elementu wektora **y** wyznacza się iloczyn skalarny dwóch wektorów, pierwszy z których jest wierszem macierzy **A**, a drugi – wektorem **x**. Algorytm powinien wykonać 2 operacji za jedną iteracje pętli wewnętrznej, jaka zawiera *n* iteracji i będzie powtórzona *n* razy w pętle zewnętrznej. Oznacza to, że $f = 2n^2$.

Element $\mathbf{y}_i$ nie zależy od indeksu iteracji pętli wewnętrznej i dlatego może być umieszczony w rejestrze przed uruchomieniem pętli wewnętrznej i jest ciągłe utrzymywany tam aż do zakończenia tej pętli. Wyładowanie do pamięci zmodyfikowanej wartości $\mathbf{y}_i$ odbywa się po zakończeniu iteracji pętli *j*. W trakcie przebiegu algorytmu trzeba wykonać *n* odczytów i *n* zapisów elementów wektora **y**.

Elementy macierzy **A** będą odczytane n^2 razy (każdy element jest odczytywany tylko jeden raz) w tej kolejności, w której pozostałe umieszczone w tablice **A**.

Elementy wektora **x** będą odczytane tylko jeden raz przy pierwszym wykonaniu pętli *j*, jeśli rozmiar pamięci podręcznej zezwala na przechowywanie całego wektora **x** . Przy tym będzie zrobione *n* odczytów. Jeśli rozmiar zadania jest na tyle duży, że **x** wektor nie jest w stanie zmieścić się w pamięci podręcznej, to jego elementy będą odczytywane ponownie przy każdym uruchomieniu pętli *j*. W takim przypadku będziemy mieli n^2 odczytów dla elementów wektora **x**.

Ilość transferów danych dla całego algorytmu wynosi $m = n^2+3n$, jeśli $M \geq 2n$ i $m = 2n^2+2n$, jeśli $M < 2n$.

Wskaźnik wydajności jest równy

$$\begin{cases} q = \dfrac{f}{m} = \dfrac{2n^2}{n^2+3n} = \dfrac{2}{1+\dfrac{3}{n}} \approx 2, & M \geq 2n \\ q = \dfrac{f}{m} = \dfrac{2n^2}{2n^2+2n} = \dfrac{2}{2+\dfrac{2}{n}} \approx 1, & M < 2n \end{cases} \tag{2.5}$$

Wydajność drugiego algorytmu okazuje się wyższa od pierwszego zagadnienia, nie patrząc na to, że tak w pierwszym zagadnieniu, jak i w drugim wskaźnik wydajności nie zależy od rozmiaru pamięci podręcznej. Oznacza to, że te algorytmy są skazane na wykonanie z prędkością wolnej magistrali, a nie szybkiego procesora [Demmel J. W., 1997]. Zwiększenie częstotliwości zegara procesora albo objętości pamięci podręcznej nie powoduje istotnego przyspieszenia obliczeń.

Rozważmy przykład. Załóżmy, że trwałość transferu jednego słowa wynosi ~40 cykli procesora. Dla algorytmu $q = 1$ oznacza, że na jedną operację arytmetyczną przypada jeden transfer danych. Jeden cykl procesor zużyje na operacje arytmetyczną. Pozostałe 39 cykli, natomiast są puste, ponieważ procesor będzie oczekiwał na dostarczenie danych.

Tu nie zostały uwzględnione *prefetch* i ten fakt, że ładowanie danych do pamięci podręcznej jest wykonywane nie słowo po słowie, a grupami słów, umieszczonych w cache-linie. Jednakże, to nie zmienia sytuacji w całości.

2.1.4 Algorytm C = C+A·B

2.1.4.1 Algorytm klasyczny (naiwny)

Istnieje kilka modyfikacji algorytmu mnożenia macierzy przez macierz [Golub, Van Loan, 1996]. Rozważmy algorytm przy umieszczeniu kolejności pętli *i, j, k*. Taki algorytm jest najczęściej podawany w poradnikach matematycznych, a jego diagram jest przedstawiony w postaci

$$\underbrace{\begin{array}{c}\rightarrow j\\ \downarrow i\end{array}\begin{pmatrix}1&2&3\\4&5&6\\7&8&9\end{pmatrix}}_{\mathbf{C}} += \underbrace{\begin{array}{c}\rightarrow k\\ \downarrow i\end{array}\begin{pmatrix}1&2&3\\4&5&6\\7&8&9\end{pmatrix}}_{\mathbf{A}} \bullet \underbrace{\begin{array}{c}\rightarrow j\\ \downarrow k\end{array}\begin{pmatrix}1&2&3\\4&5&6\\7&8&9\end{pmatrix}}_{\mathbf{B}}, \tag{2.6}$$

gdzie liczby 1, 2, 3, ... oznaczają kolejność umieszczenia elementów macierzy w tablice jednowymiarowej. Kierunki indeksów są oznaczone strzałkami. Algorytm jest zrealizowany dokładnie tak, jak jest podany w poradnikach

$$c_{ij} += \sum_{k=1}^{n} a_{ik} b_{kj}, \quad i, j \in [1, n] \ , \tag{2.7}$$

i dlatego dostał nazwę algorytmu naiwnego. Algorytm ten jest podany niżej.

```
for(i=1; i<=n; i++)
{
    for(j=1; j<=n; j++)
    {
          //odczyt cij  i umieszczenie w rejestrze
          for(k=1; k<=n; k++)
          {
                //odczyt aik , akj
                cij = cij + aik*bkj ;
          }
          //zapis cij
    }
}
```

Pętla wewnętrzna algorytmu przedstawia sobą obliczenie iloczynu skalarnego dwóch wektorów, pierwszy z których jest wierszem *i* macierzy **A**, a drugi – kolumną *j* macierzy **B**.

Dla macierzy kwadratowych algorytm wykonuje $2n^3$ operacji arytmetycznych – $f = 2n^3$.

Elementy macierzy **C** nie zależą od indeksu *k*, dlatego element c_{ij} jest ładowany do rejestru przed uruchomieniem pętli wewnętrznej i pozostaje tam dotąd, dokąd pętla *j* nie skończy się. Wyładowanie z rejestru do pamięci odbywa się po zakończeniu pętli. Takim czynem dla elementów macierzy **C** mamy n^2 odczytów i n^2 zapisów.

Elementy macierzy **B** zależą od dwóch ostatnich indeksów – przy każdym uruchomieniu pętli *k* elementy macierzy **B** powinny być odczytane ponownie – mamy n^3 odczytów.

Elementy macierzy **A** będą odczytane n^2 razy, jeśli objętość pamięci podręcznej jest wystarczająca dla przechowywania całego wiersza i macierzy **A**. W przeciwnym wypadku przy każdym uruchomieniu pętli k elementy macierzy **A** powinny być odczytane ponownie, czyli ilość odczytów będzie wynosić n^3.

Ilość transferów danych RAM – cache – RAM wynosi $m = n^3+3n^2$, jeśli $M \geq 2n$, albo $m = 2n^3+2n^2$, jeśli $M < 2n$. Stąd otrzymujemy wskaźnik wydajności:

$$\begin{cases} q = \dfrac{f}{m} = \dfrac{2n^3}{n^3+3n^2} = \dfrac{2}{1+\dfrac{3}{n}} \approx 2, & M \geq 2n \\ q = \dfrac{f}{m} = \dfrac{2n^3}{2n^3+2n^2} = \dfrac{1}{1+\dfrac{1}{n}} \approx 1, & M < 2n \end{cases} . \tag{2.8}$$

Ten algorytm wykazał najmniejszą wydajność (rys. 2.1), ponieważ elementy macierzy **B** w pętli wewnętrznej są pobierane skokowo – w tablicy jednowymiarowej elementy są umieszczone wiersz po wierszu, a algorytm odczytuje ich kolumna po kolumnie. Powoduje to w przypadku dużych macierzy bardzo nie skuteczny sposób działania układu pamięci i, jako skutek, drastyczne zwolnienie obliczeń.

2.1.4.2 Usunięcie skoków danych przy odczytywaniu elementów macierzy B

Przestawimy pomiędzy sobą pętli o indeksach j, k . Zmodyfikowany algorytm ma wygląd

```
for(i=1; i<=n; i++)
{
    for(k=1; k<=n; k++)
    {
          //odczyt aik  i umieszczenie w rejestrze
          for(j=1; j<=n; j++)
          {
                //odczyt cij , bkj
                cij = cij + aik*bkj ;
                //zapis cij
          }

    }
}
```

W pętle wewnętrznej algorytm wykonuje mnożenie wektora przez skalar, ponieważ element macierzy a_{ik} nie zależy od indeksu pętli wewnętrznej i dlatego

może być umieszczony w rejestrze przed rozpoczęciem tej pętli i ciągłe tam pozostawać do jej zakończenia. W samej pętle wiersz k macierzy **B** jest mnożony poprzez skalar a_{ik}, w skutek czego dostajemy poprawiony wiersz i macierzy **C** . Algorytm wykonuje $2n^3$ operacji arytmetycznych – $f = 2n^3$.

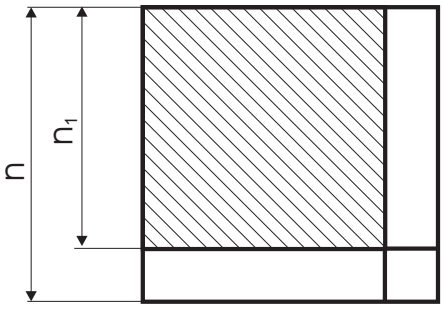
Ilość transferów danych RAM – cache – RAM wynosi $m = n^3+3n^2$, jeśli $M \geq 2n$, albo $m = 3n^3+n^2$, jeśli $M < 2n$. Uzasadnienie tych wyników pozostawiamy czytelnikowi jako ćwiczenie. Stąd dostajemy wskaźnik wydajności:

$$\begin{cases} q = \dfrac{f}{m} = \dfrac{2n^3}{n^3+3n^2} = \dfrac{2}{1+\dfrac{3}{n}} \approx 2, & M \geq 2n \\ q = \dfrac{f}{m} = \dfrac{2n^3}{3n^3+n^2} = \dfrac{2}{3+\dfrac{1}{n}} \approx \dfrac{2}{3}, & M < 2n \end{cases} . \qquad (2.9)$$

W użytym przez nas modelu teoretycznym nie było brane pod uwagę istotne zwolnienie pracy algorytmów przy skokowym pobieraniu danych z pamięci głównej. Dlatego wskaźnik wydajności dla macierzy dużych ($M < 2n$) okazuje się być nawet gorszym, niż w przypadku poprzednim, chociaż w rzeczywistości algorytm *i*, *k*, *j* działa znacznie szybszej (rys. 2.1), ponieważ skoki w danych przy odczycie elementów macierzy **B** pozostają usunięte.

2.1.4.3 Podział macierzy na bloki – blokowanie pamięci podręcznej

Podzielimy każdą z macierzy **A**, **B**, **C** na bloki kwadratowe tak żeby w pamięci podręcznej można by było umieścić po jednym bloku z każdej macierzy. Będziemy zakładać, że rozmiar zadania n jest wielokrotny rozmiarowi bloku l_b. Takie założenie nie obniża ogólności wniosków, ponieważ naszym celem jest wyjaśnienie, jakie czynniki wpływają na podniesienie wydajności. W algorytmach rzeczywistych, przeznaczonych dla praktycznego zastosowania, programiści stosują tak zwane podkładanie – znajdują największy rozmiar zadania $n_1 < n$ taki że $n_1 \% l_b = 0$ (reszta od dzielenia n_1 przez l_b jest równa zeru, czyli n_1 jest wielokrotny do l_b). Przy tym główna część zadania będzie wykonana na podstawie algorytmu blokowego, a niewielka pozostała część – algorytmu naiwnego (rys. 2.2).



Rys. 2.2 Podkładanie: znajdujemy taki największy rozmiar $n_1 < n$ że n_1 jest wielokrotny rozmiarowi bloku l_b. Dla zakreskowanej części macierzy stosujemy metodę blokową, a dla pozostałej – metodę naiwną

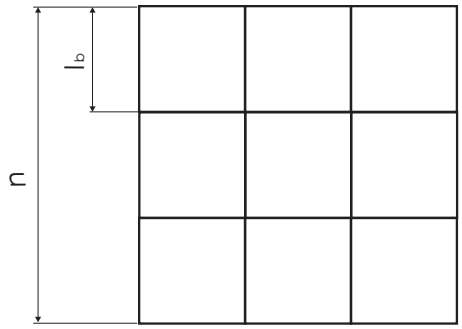
Podział macierzy na bloki dla przykładu o rozmiarze 4×4 oraz kierunki indeksów blokowych *ib*, *jb*, *kb* są przedstawione na diagramie

$$\underset{\mathbf{C}}{\underbrace{\begin{array}{c}\rightarrow jb\\ \downarrow\\ ib\end{array}\begin{pmatrix} 1 & 2 & : & 3 & 4 \\ \underbrace{5}_{\mathbf{C}_{11}} & \underbrace{6}_{\mathbf{C}_{11}} & : & \underbrace{7}_{\mathbf{C}_{12}} & \underbrace{8}_{\mathbf{C}_{12}} \\ \cdots\cdots & \cdots\cdots & : & \cdots\cdots & \cdots\cdots \\ 9 & 10 & : & 11 & 12 \\ \underbrace{13}_{\mathbf{C}_{21}} & \underbrace{14}_{\mathbf{C}_{21}} & : & \underbrace{15}_{\mathbf{C}_{22}} & \underbrace{16}_{\mathbf{C}_{22}} \end{pmatrix}}} = \underset{\mathbf{A}}{\underbrace{\begin{array}{c}\rightarrow kb\\ \downarrow\\ ib\end{array}\begin{pmatrix} 1 & 2 & : & 3 & 4 \\ \underbrace{5}_{\mathbf{A}_{11}} & \underbrace{6}_{\mathbf{A}_{11}} & : & \underbrace{7}_{\mathbf{A}_{12}} & \underbrace{8}_{\mathbf{A}_{12}} \\ \cdots\cdots & \cdots\cdots & : & \cdots\cdots & \cdots\cdots \\ 9 & 10 & : & 11 & 12 \\ \underbrace{13}_{\mathbf{A}_{21}} & \underbrace{14}_{\mathbf{A}_{21}} & : & \underbrace{15}_{\mathbf{A}_{22}} & \underbrace{16}_{\mathbf{A}_{22}} \end{pmatrix}}} \bullet \underset{\mathbf{B}}{\underbrace{\begin{array}{c}\rightarrow jb\\ \downarrow\\ kb\end{array}\begin{pmatrix} 1 & 2 & : & 3 & 4 \\ \underbrace{5}_{\mathbf{B}_{11}} & \underbrace{6}_{\mathbf{B}_{11}} & : & \underbrace{7}_{\mathbf{B}_{12}} & \underbrace{8}_{\mathbf{B}_{12}} \\ \cdots\cdots & \cdots\cdots & : & \cdots\cdots & \cdots\cdots \\ 9 & 10 & : & 11 & 12 \\ \underbrace{13}_{\mathbf{B}_{21}} & \underbrace{14}_{\mathbf{B}_{21}} & : & \underbrace{15}_{\mathbf{B}_{22}} & \underbrace{16}_{\mathbf{B}_{22}} \end{pmatrix}}} .$$

Dla macierzy blokowej stosujemy działania nad blokami

$$\mathbf{C}_{ib,\,jb} = \mathbf{C}_{ib,\,jb} + \sum_{kb=1}^{Nb} \mathbf{A}_{ib,kb} \cdot \mathbf{B}_{kb,\,jb}, \quad ib,\, jb \in [1, Nb] \tag{2.10}$$

Pozostawiamy czytelnikowi udowodnienie, że algorytm blokowy doprowadzi do dokładnie takich samych wyników jak i algorytm zwykły. Tu N_b – ilość bloków wzdłuż jednego boku macierzy (rys. 2.3). Dla podanego przykładu $N_b = 3$.



Rys. 2.3 Podział macierzy na bloki o rozmiarze l_b

Algorytm metody blokowej wygląda tak:

```
for(ib=1; ib<=Nb; ib++)
{
    for(jb=1; jb<= Nb; jb++)
    {
          //odczyt  bloku Cib,jb i umieszczenie w cache
          for(kb=1; kb<= Nb; kb++)
          {
                //odczyt bloków Aib,kb , Bkb,jb
                Cib,jb = Cib,jb + Aib,kb·Bkb,jb ;
          }
          //zapis bloku Cib,jb
    }
}
```

Przy mnożeniu bloków w pętli wewnętrznej jest używany algorytm naiwny. Ponieważ bloki zostały umieszczone w pamięci podręcznej nie występuje drastyczne obniżenie wydajności w skutek skoków przy pobieraniu elementów bloku $B_{kb,jb}$.

Ilość operacji arytmetycznych wykonanych przy jednej iteracji pętli wewnętrznej, wynosi $2l_b^3$. Ilość iteracji przy wykonaniu całego zadania jest równa N_b^3, dlatego algorytm wykonuje $f = 2l_b^3 N_b^3 = 2n^3$ operacji arytmetycznych, ponieważ $n = l_b N_b$ (rys. 2.3). Jest to dokładnie tyle, ile wykonują algorytmy naiwne.

Indeksy bloku $C_{ib,jb}$ nie zależą od indeksu iteracji pętli wewnętrznej *kb*, stąd wynika że blok $C_{ib,jb}$ będzie odczytany do pamięci podręcznej przed uruchomieniem pętli wewnętrznej, a zapisany do pamięci głównej – po

zakończeniu pętli *kb* [Demmel J. W., 1997]. Ilość transferów danych dla bloków macierzy **C** przy wykonaniu całego zadania wynosi

$$2N_b^2 l_b^2 = 2N_b^2 \underbrace{\left(\frac{n}{N_b}\right)^2}_{l_b^2} = 2n^2 \ .$$

Rozmiar bloku l_b jest dość duży, dlatego każda zmiana indeksów algorytmu powoduje ponowne ładowanie bloków macierzy **A**, **B** do pamięci podręcznej. To oznacza, że ilość odczytów dla tych macierzy razem wynosi

$$2N_b^3 l_b^2 = 2N_b^3 \underbrace{\left(\frac{n}{N_b}\right)^2}_{l_b^2} = 2N_b n^2 \ .$$

Ilość transferów danych dla algorytmu blokowego wynosi

$$m = 2N_b n^2 + 2n^2 = 2n^2\left(N_b + 1\right) \approx 2n^2 N_b \ ,$$

a wskaźnik wydajności –

$$q = \frac{f}{m} = \frac{2n^3}{2n^2 N_b} = \frac{n}{N_b} = l_b$$

Im większy rozmiar bloku, tym wyższa jest wydajność algorytmu. Przy czym rozmiar bloku jest ograniczony warunkiem, że trzy bloki powinny być umieszczone w pamięci podręcznej $3l_b^2 \le M$. Stąd wynika, że optymalny rozmiar bloku jest równy

$$l_b = \sqrt{\frac{M}{3}} \ ,$$

a wydajność algorytmu blokowego –

$$q = \sqrt{\frac{M}{3}} \qquad (2.11)$$

Dla algorytmu blokowego wydajność rośnie przy zwiększeniu rozmiaru pamięci podręcznej.

Wróćmy do przykładu z rozdziału 2.1.3. Teraz nasz procesor podczas każdego transferu danych powinien wykonać $f = qm\big|_{m=1} = \sqrt{\frac{M}{3}}$ operacji arytmetycznych. Przy wystarczająco dużym rozmiarze M ilość pustych cykli będzie dążyć do zera.

Wydajność algorytmów algebry liniowej przyjęto odnosić do jednego z poziomów BLAS (Basis Linear Algebra Subroutines – biblioteka wysokiej wydajności typowych procedur algebry liniowej) zgodnie z rzędem operacji arytmetycznych $O(n^k)$. BLAS 1 – $O(n^1)$, BLAS 2 – $O(n^2)$, BLAS 3 – $O(n^3)$ – tab. 2.1. Jedną z najlepszych realizacji BLAS zawiera [Intel MKL].

Tabela 2.1.

Klasyfikacja algorytmów algebry liniowej według BLAS

Algorytm	$q = \frac{f}{m}$	Poziom BLAS
$\mathbf{y} = \mathbf{y} + \alpha \cdot \mathbf{x}$	2/3	1 – O(n)
$\mathbf{y} = \mathbf{y} + \mathbf{A} \cdot \mathbf{x}$	2 ($M \geq 2n$) 1 ($M < 2n$)	2 – O(n^2)
$\mathbf{C} = \mathbf{C} + \mathbf{A} \cdot \mathbf{B}$ Classic i, j, k	2 ($M \geq 2n$) 1 ($M < 2n$)	3 – O(n^3)
$\mathbf{C} = \mathbf{C} + \mathbf{A} \cdot \mathbf{B}$ Classic i, k, j	2 ($M \geq 2n$) 2/3 ($M < 2n$)	3 – O(n^3)
$\mathbf{C} = \mathbf{C} + \mathbf{A} \cdot \mathbf{B}$ Metoda blokowa	$q = \sqrt{\frac{M}{3}}$	3 – O(n^3)

Algorytmy poziomów 1, 2 BLAS pracują na niskim poziomie wydajności ($q \leq 2$) z prędkością wolnego układu pamięci. Przyczyną tego jest to, że oni zawierają ilość danych takiego samego rzędu, jak i ilość operacji arytmetycznych. Przy tym nie ma możliwości wielokrotnego wykorzystania szybkiej pamięci podręcznej.

Dla algorytmów poziomu 3 BLAS ilość operacji arytmetycznych jest większego rzędu w stosunku do ilości danych. Takie algorytmy mają przynajmniej teoretyczną możliwość wielokrotnego użycia danych, które zostały jeden raz

umieszczone do pamięci podręcznej. Algorytmy naiwne nie potrafią tego zrealizować, i wydajność okazała się dla nich nie większa niż dla algorytmów poziomu 1, 2 BLAS. Dla algorytmu blokowego podniesienie wydajności okazało się możliwe w skutek tego, że dane, jeden raz umieszczone w pamięć podręczną, były wielokrotnie odczytywane z szybkiej pamięci podręcznej, a nie z wolnej pamięci głównej. Ta technika dostała nazwę cache reuse, podstawą ku temu posłużył podział macierzy na bloki.

2.1.4.4 Blokowanie rejestrów

Dotąd rozważaliśmy transfer danych pomiędzy pamięcią główną a pamięcią podręczną. Teraz zajmiemy się poziomem pamięć podręczna – rejestry – pamięć podręczna.

Okazuje się, że zastosowanie techniki blokowej zmniejsza ilość ładowania rejestrów i realizuje zasadę, że dane, które zostały umieszczone jeden raz w rejestrze, powinny być wielokrotnie użyte (register's reuse). Tak samo, jak i w przypadku blokowania pamięci podręcznej, stosowanie techniki blokowej na poziomie rejestrów procesora (register's blocking) doprowadza do podniesienia wydajności algorytmu.

Rozważmy algorytm $C_{ib,jb} = C_{ib,jb} + A_{ib,kb} \cdot B_{kb,jb}$, gdzie $A_{ib,kb}$, $B_{kb,jb}$, $C_{ib,jb}$ są to bloki macierzy, umieszczone w pamięci podręcznej.

Najpierw zastosujemy do mnożenia tych bloków algorytm naiwny *i*, *k*, *j*.

```
for(i=1; i<=lb; ++i)
{
    for(j=1; j<=lb; ++j)
    {
          r = 0;
          for(k=1; k<=lb; ++k)
          {
                r = r + aik·bkj ;
          }
          cij = r;
    }
}
```

Tu *r* – zmienna klasy pamięci *register*, która ma być przechowywana w rejestrze podczas wykonania całej pętli wewnętrznej, a_{ik}, b_{ik} – elementy bloków $A_{ib,kb}$, $B_{kb,jb}$ odpowiednio. Po zakończeniu pętli wewnętrznej wartość zmiennej *r* będzie przypisana do c_{ij} – elementowi bloku $C_{ib,jb}$. W pętli wewnętrznej liczy się iloczyn skalarny, przy czym pierwszym wektorem jest wiersz *i* bloku $A_{ib,kb}$, a drugim – kolumna *j* bloku $B_{kb,jb}$.

Policzymy ilość operacji arytmetycznych i ilość ładowań rejestrów. Za jedną iteracje pętli *k* jest wykonywane 2 operacje arytmetyczne (jedno mnożenie i jedno

dodawanie) i 2 ładowania danych do rejestrów (elementy a_{ik}, b_{ik}). Przy wykonaniu całego zadania instrukcja kodu w pętli wewnętrznej będzie powtórzona n^3 razy, czyli będzie wykonana $2n^3$ operacji arytmetycznych oraz $2n^3$ ładowań do rejestrów. Wskaźnik wydajności wynosi

$$q = \frac{2n^3}{2n^3} = \frac{2}{2} = 1$$

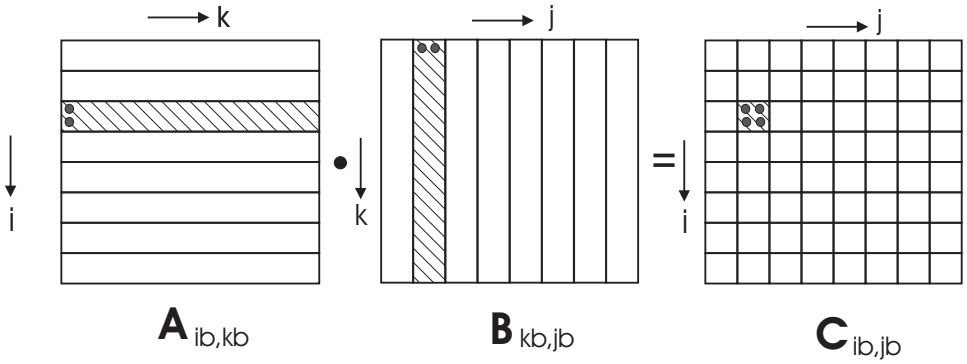
W taki sposób można policzyć tylko wskaźnik wydajności dla jednej iteracji pętli wewnętrznej. Dla algorytmu naiwnego q = 1, a dla jego wykonania są niezbędne 4 rejestry typu *double* – jeden rejestr – dla *r*, dwa rejestry – dla elementów a_{ik}, b_{ik} i jeden rejestr – pomocniczy, dla umieszczenia wyniku pośredniego przy wykonaniu mnożenia, ponieważ na poziome asemblera mnożenie jest operacją binarną, przy czym wynik będzie umieszczony do adresu pierwszego operandu. Żeby zachować wartość pierwszego operandu bez zmian, trzeba przed mnożeniem skopiować jego do rejestru pomocniczego, w którym po wykonaniu operacji mnożenia okaże się wynik.

Rozważmy teraz algorytm mnożenia bloków macierzy przy blokowaniu rejestrów w schemacie 2×2.

```
for(i=0; i<lb; i+=2)
{
    for(j=0; j<lb; j+=2)
    {
       //r0, r1, r2, r3 ← 0 – wyzerowujemy rejestry, w
       //których będziemy przechowywali elementy macierzy C
          for(k=0; k<lb; ++k)
          {
                r0 = r0 + a_ik · b_kj;
                r1 = r1 + a_i+1,k · b_kj;
                r2 = r2 + a_i,k · b_k,j+1;
                r3 = r3 + a_i+1,k · b_k,j+1;
          }
          c_ij     += r0;
          c_i+1,j  += r1;
          c_i,j+1  += r2;
          c_i+1,j+1 += r3;

    }
}
```

Pętla *i* jest inkrementowana ze skokiem 2. To oznacza, że macierzy $\mathbf{A}_{ib,kb}$ oraz $\mathbf{C}_{ib,jb}$ będą podzielone na poziome pasma o szerokości 2 (rys. 2.4). Pętla *j* też jest inkrementowana ze skokiem 2. Macierzy $\mathbf{B}_{kb,jb}$ i $\mathbf{C}_{ib,jb}$ będą podzielone na pionowe

pasma o szerokości 2. Macierz $\mathbf{C}_{ib,jb}$ więc będzie podzielona na klatki o rozmiarze 2×2 .



Rys. 2.4 Graficzna interpretacja algorytmu blokowania rejestrów 2×2

Żeby otrzymać wynik poprawny, trzeba rozwinąć pętlę wewnętrzną o wartości indeksów i+1, j+1. Oznacza to, że w pętli wewnętrznej zamiast obliczenia iloczynu skalarnego przy wykonaniu operacji skalarno-wektorowych, jak to było robione w algorytmie naiwnym, liczymy iloczyn macierzy zakreskowanych na rys. 2.4, przy czym otrzymujemy wartości elementów macierzy $\mathbf{C}_{ib,jb}$ w klatce 2×2. Są to elementy c_{ij} , $c_{i+1,\,j}$, $c_{i,j+1}$, $c_{i+1,j+1}$. Elementy te nie zależą od indeksu iteracji pętli wewnętrznej, dlatego są przechowywane w rejestrach $r0$ – $r3$ przy wykonaniu iteracji pętli wewnętrznej. Wyładowanie danych z tych rejestrów do pamięci odbywa się po zakończeniu pętli k . Za jedną iteracje pętli wewnętrznej trzeba wykonać 4 mnożenia i 4 dodawania (razem – 8 operacji arytmetycznych) oraz 4 ładowania elementów macierzy a_{ik} , $a_{i+1,k}$, b_{kj} , $b_{k,j+1}$ do rejestrów. W algorytmie elementy te zostały podkreślone. W taki sposób, blokowanie rejestrów 2×2 doprowadziło do tego, że w pętli wewnętrznej każdy element jest jeden raz ładowany i dwa razy używany w wyrażeniach arytmetycznych. Wskaźnik wydajności wynosi q = 8/4 = 2. Jest to w dwa razy więcej niż dla algorytmu naiwnego.

Dla wykonania tego algorytmu będą potrzebne 8 rejestrów: 4 rejestry dla elementów macierzy $\mathbf{C}_{ib,jb}$, 2 – dla elementów macierzy $\mathbf{A}_{ib,kb}$ i 1 – dla elementów macierzy $\mathbf{B}_{kb,jb}$. Jeszcze jeden rejestr potrzebujemy dla przechowywania wyników operacji mnożenia. Taki algorytm może być uruchomiony na platformie 32-bitowej, ponieważ są dostępne 8 rejestrów dla liczb typu *double*. Na platformie 64-bitowej mamy dostępnych 16 takich rejestrów. Dlatego możemy zastosować rozmiar bloku 3×3. Potrzebuje to 9 rejestrów dla elementów macierzy $\mathbf{C}_{ib,jb}$, 3 rejestry dla elementów macierzy $\mathbf{A}_{ib,kb}$, 1 rejestr dla elementów macierzy $\mathbf{B}_{kb,jb}$ i 1

rejestr pomocniczy – razem 14 rejestrów. W jednej iteracji pętli wewnętrznej będą wykonane 18 operacji arytmetycznych i 6 ładowań do rejestrów. Wskaźnik wydajności jest równy q = 18/6 = 3. Przedstawiam czytelnikowi możliwość samodzielnego stworzenia algorytmu blokowania rejestrów 3×3 i udowodnienia tego, co wyżej zostało przedstawione.

Dla rozmiaru zadania N = 1 000 w tabeli 2.2 jest podana wydajność dla algorytmu naiwnego (*i*, *j*, *k*), algorytmu blokowania rejestrów 2×2 i algorytmu blokowania pamięci podręcznej l_b = 64 oraz blokowania rejestrów 2×2 – procedura *dgemm* BLAS 1989 [Dongarra, Mayes, di Brozolo , 1989].

Obliczenia były wykonane na komputerze z procesorem Intel® Core™2 Quad CPU Q6600 @2.40 GHz. Program *dgemm* BLAS 1989 został pobrany z Internetu i przepisany z Fortranu na C (załącznik 2).

Tabela 2.2

Porównanie wydajności (MFLOPS) algorytmów **C = C+A·B**

Algorytm naiwny ijk	300
Algorytm blokowania rejestrów 2×2	2 600
Algorytm blokowania cache (l_b=64) oraz rejestrów 2×2 (BLAS 1989, J. Dongarra)	3 121

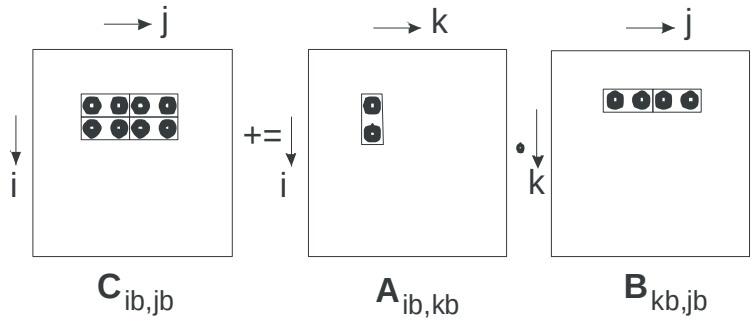
Wszystkie programy zostały skompilowane kompilatorem Intel C/C++ 10.1.021 przy opcjach /O3 /QaxT /QxT /Qunroll:10 /Qparallel.

2.1.4.5 Wektoryzowanie obliczeń a użycie rejestrów XMM. Techniki oprogramowania SSE2, SSE3 .

Następnym etapem podniesienia wydajności jest zastosowanie elementów wektoryzowania obliczeń. Procesory Pentium IV, Pentium Dual-Core, Intel® Core™2 Quad, Intel Core i5, Intel Core i7 i wiele innych zawierają 16 128-bitowych rejestrów XMM, każdy z których umieszcza dwa słowa typu *double*. Takie rejestry nadają możliwość w jednym cyklu procesora wykonać dwa mnożenia lub dwa dodawania. W porównaniu ze zwykłymi rejestrami typu *double*, które mogą wykonywać tylko jedną operacje (mnożenie lub dodawanie) za jeden cykl procesora, użycie rejestrów XMM teoretycznie zezwala podnieść wydajność algorytmów obliczeniowych, jeśli nie powstaje przy tym zatrzymanie transferu danych RAM – cache – RAM, cache – rejestry – cache. Jest to element architektury SIMD (single instruction stream – multiple data stream). Jeszcze kilka lat temu zastosowanie rejestrów XMM było możliwe tylko w języku assemblera. Teraz technologie SSE2, SSE3 (Streaming SIMD Extensions SSE2, SSE3) dają możliwość programowania z użyciem rejestrów XMM na poziomie języka C/C++ [SSE2].

Dla podniesienia wydajności jest stosowane blokowanie rejestrów XMM. Istnieją różne schematy blokowania. Dla aplikacji 32-bitowych (platforma ia32) dostępnych jest tylko 8 rejestrów XMM. Stąd i blokowanie odbywa się w

schemacie albo 2×4, albo 4×2. Na rys. 2.5 jest przedstawiony schemat blokowania 2×4 rejestrów XMM, używany dla procesorów Pentium IV architektury Prescott и Opteron [Goto K, Van De Geijn R A, 2008].



Rys. 2.5 Schemat blokowania 2×4 rejestrów XMM

Przy jednej iteracji pętli wewnętrznej mamy 16 operacji arytmetycznych oraz 6 ładowań danych do rejestrów. Wskaźnik wydajności q = 16/6 = 2.67.

Fragment kodu, napisanego w technice SSE2, jest przedstawiony poniżej.

```
#include <emmintrin.h>
#include <intrin.h>
__m128d c1, c2, c3, c4, w, w1, wt, wt1;//są potrzebne 8
                                       //rejestrów XMM
const int mr = 2;
const int nr = 4;
for(i=0; i<lbv; i+=mr)
{
   for(j=0; j<lbh; j+=nr)
   {
      pWt = WT+M*j;//wskazuje do Bkb,jb ;  M – ilość wierszy
                   //w bloku Bkb,jb
      pW = W+i*M;  //wskazuje do Aib,kb  ;  M – ilość kolumn
                  //w bloku Aib,kb
      //wyzerowujemy rejestry c1 – c4 dla elementów bloku
      // Cib,kb
      c1 = _mm_setzero_pd();
      c2 = _mm_setzero_pd();
      c3 = _mm_setzero_pd();
      c4 = _mm_setzero_pd();

      for(k=0; k<M; k++)
      {
```

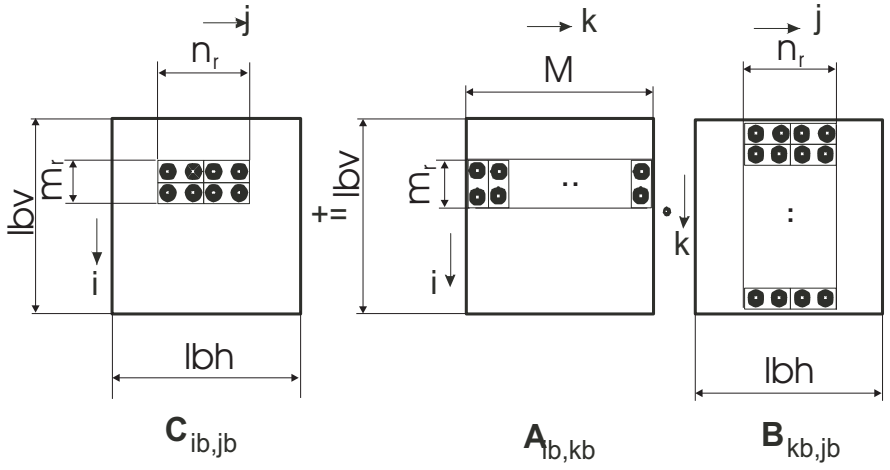
```
        w   = _mm_load1_pd(pW);     //load w <- a_ik, a_ik
        wt  = _mm_load_pd(pWt);     //load wt <- b_kj, b_k,j+1
        wt1 = _mm_load_pd(pWt+2);//load wt1 <- b_k,j+2, b_k,j+3
        w1  = _mm_mul_pd(w, wt);//{a_ikb_kj, a_ikb_k,j+1}: w1<-w*wt
        c1 = _mm_add_pd(c1, w1);//{c_ij, c_i,j+1} <- {c_ij, c_i,j+1}+
                                //{a_ikb_kj, a_ikb_k,j+1}

        w1 = _mm_mul_pd(w, wt1);//{a_ikb_k,j+2, a_ikb_k,j+3}:
                                //w1<-w*wt1
        c2 = _mm_add_pd(c2, w1);//{c_i,j+2, c_i,j+3} <-
                        //{c_i,j+2, c_i,j+3}+{a_ikb_k,j+2, a_ikb_k,j+3}
        w  = _mm_load1_pd(pW+1);//load w <- a_i+1,k, a_i+1,k
        w1 = _mm_mul_pd(w, wt);//{a_i+1,kb_kj, a_i+1,kb_k,j+1}:
                               //w1<-w*wt
        c3 = _mm_add_pd(c3, w1);//{c_i+1,j, c_i+1,j+1} <-
                        //{c_i+1,j, c_i+1,j+1}+{a_i+1,kb_kj, a_i+1,kb_k,j+1}
        w1 = _mm_mul_pd(w, wt1);//{a_i+1,kb_kj, a_i+1,kb_k,j+1}:
                                //w1<-w*wt
        c4 = _mm_add_pd(c4, w1);//{c_i+1,j+2, c_i+1,j+3}
                  //<- {c_i+1,j+2, c_i+1,j+3}+{a_i+1,kb_k,j+2, a_i+1,kb_k,j+3}
        pW += mr;
        pWt+= nr;
      }

     pos_CW = mr*j;
     //wyładujemy z rejestrów do bloku Cib,kb
     _mm_store_pd(CW+pos_CW, c1);
     _mm_store_pd(CW+pos_CW+2, c2);
     _mm_store_pd(CW+pos_CW+4, c3);
     _mm_store_pd(CW+pos_CW+6, c4);

    }//j loop
}//i loop
```

Dane dla tablic $\mathbf{B}_{kb,jb}$, $\mathbf{C}_{ib,jb}$ mają być wyrównane do granic 16 bajtów ponieważ są one załadowane przy pomocy instrukcji *_mm_load_pd()* oraz wyładowane instrukcją *_mm_store_pd()*. Dlatego są używane funkcje [MSDN]:

```
ptr_A = (double *)_aligned_malloc(number_of_bytes, 16),
_aligned_free(ptr_A).
```

Tu *ptr_A* – bufor pamięci, który alokuje się dynamicznie z wyrównaniem na granicę 16 bajtów, *lbv*, *lbh* – ilość wierszy i ilość kolumn bloku macierzy $\mathbf{C}_{ib,jb}$ oraz ilość wierszy w bloku $\mathbf{A}_{ib,kb}$ i ilość kolumn w bloku macierzy $\mathbf{B}_{kb,jb}$, M – ilość kolumn bloku $\mathbf{A}_{ib,kb,}$ oraz ilość wierszy bloku $\mathbf{B}_{kb,jb}$. Indeksy iteracji *i*, *j* są inkrementowane ze skokami *mr*, *nr* odpowiednio. Powoduje to podział bloku $\mathbf{C}_{ib,jb}$

na klatki o rozmiarze *mr* × *nr*, bloku $\mathbf{A}_{ib,kb}$ – na pasma poziome o szerokości *mr*, a bloku $\mathbf{B}_{kb,jb}$ – na pasma pionowe o szerokości *nr* (rys. 2.6).



Rys. 2.6 W pętle wewnętrznej obliczamy iloczyn podmacierzy wydzielonych

Dla wspierania potokowego przetwarzania danych pętla wewnętrzna pozostaje rozwinięta K_1 razy. Ilość używanych rejestrów XMM nie zależy od K_1. W zależności od typu procesora K_1 może osiągnąć 90 – 120 razy [Yotov K., Roeder T., Pingali K., Gunnels J., Gustavson F., 2007]. Napisana w taki sposób pętla wewnętrzna dostała nazwę mikrojądra (microkernel).

2.1.4.6 Podział macierzy na bloki, pakowanie danych według Intel Math Kernel Library, mikrojądro.

Dlatego, żeby zminimalizować "read miss" (cache-chybienie przy odczycie), dane mają być umieszczone w pamięci głównej w kolejności ich pobierania. Będziemy to nazywać kolejnością optymalną.

Jednak pierwotne umieszczenie danych w każdej z macierzy i w blokach $\mathbf{C}_{ib,jb}$, $\mathbf{A}_{ib,kb}$, $\mathbf{B}_{kb,jb}$ nie odpowiada takiej kolejności. Z tego wynika konieczność przepakowania danych.

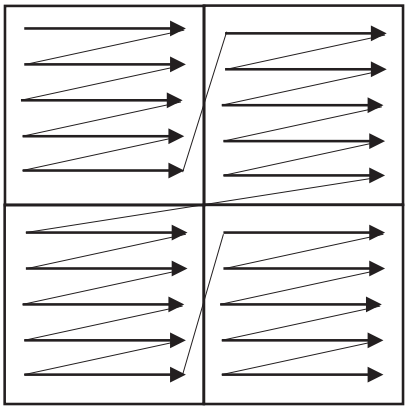
Istnieje kilka różnych sposobów przepakowania.

Algorytm przepakowania procedury DGEMM BLAS z 1989 roku dla macierzy, umieszczonych w pamięci głównej kolumna po kolumnie, wygląda tak (załącznik 2):

```
for(ib=0; ib<Nb; ++ib)
{
```

```
    for(kb=0; kb<Nb; ++kb)
    {
          //Pack  Aib,kb
          for(jb=0; jb<Nb; ++jb)
          {
                Cib,jb+=Aib,kb*Bkb,jb
          }
    }
}
```

Pakowanie elementów bloku $\mathbf{A}_{ib,kb}$ odbywa się w taki sposób, że po wykonaniu całego zadania algorytm faktycznie pracuje z macierzą, elementy której umieszczone są w pamięci tak, jak to jest pokazane na rys. 2.7 . Takie pakowanie jest optymalne ponieważ w trakcie wykonania całego zadania ilość transferów danych nie przekracza ilości elementów macierzy **A**. Po pakowaniu elementy bloku $\mathbf{A}_{ib,kb}$ są umieszczone w pamięci dokładnie w takiej kolejności, w jakiej będą pobierane w pętli wewnętrznej. Przy tym będzie potrzebna jedna dodatkowa tablica dla przechowywania przepakowanego bloku o rozmiarze $l_b \times l_b$.



Rys. 2.7 Pakowanie elementów bloku $\mathbf{A}_{ib,kb}$

Algorytm mnożenia bloków $\mathbf{C}_{ib,jb}+=\mathbf{A}_{ib,kb}*\mathbf{B}_{kb,jb}$ używa blokowanie rejestrów double 2×2:

```
for(j=1; j<lb; j+=2)
{
    for(i=0; i<lb, i+=2)
    {
          // wyzeruj rejestry dla bloku Cib,jb :
          t11 = t12 = t21 = t22 = 0;
```

```
            for(k=0; k<lb; ++k)
            {
                  t11 = t11 + a_ik·b_kj;
                  t12 = t12 + a_i+1,k·b_kj;
                  t21 = t21 + a_ik·b_k,j+1;
                  t22 = t22 + a_i+1,k·b_k,j+1;
            }
            c_ij     = t11;
            c_i+1,j  = t12;
            c_i,j+1  = t21;
            c_i+1,j+1 = t22;
      }
}
```

Teoretycznie optymalny rozmiar bloku można dostać z warunku, że trzy bloki $\mathbf{A}_{ib,kb}$, $\mathbf{B}_{kb,jb}$, $\mathbf{C}_{ib,kb}$ powinny się mieścić w pamięci podręcznej. Jeśli brać pod uwagę tylko rozmiar cache L1, to

$$3 \cdot l_b^2 = L_1 \rightarrow \quad l_b = \sqrt{\frac{L_1}{3}},$$

skąd przy L1=32K = 32*1024 = 32 768 bajtów albo 32 768/8 = 4 096 słów double dostajemy l_b = 36.95 → 36 – 40. Testy pokazują że optymalny rozmiar bloku wychodzi większy, ponieważ w podanym oszacowaniu nie była wzięta pod uwagę obecność pamięci podręcznej L2 (L3).

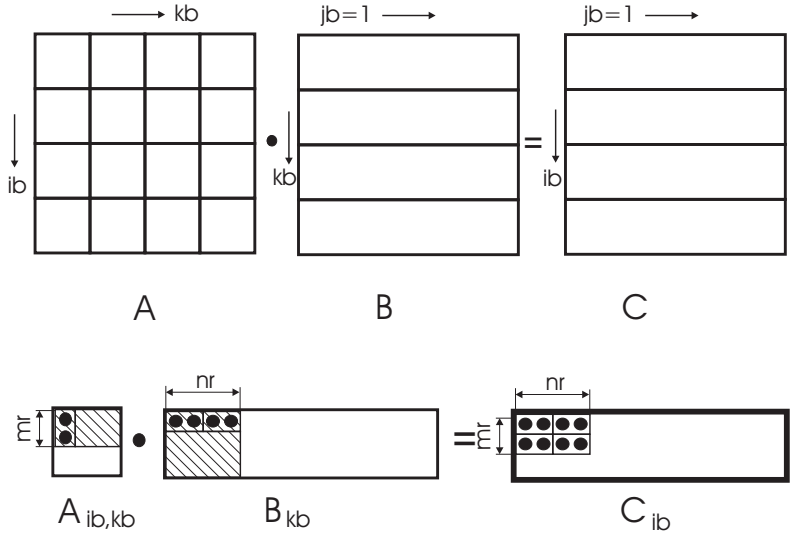
W procedurze DGEMM z biblioteki Intel MKL jest użyty inny podział na bloki (rys. 2.8) [Goto K, Van De Geijn R A, 2008].

Algorytm na poziome mnożenia bloków wygląda następująco

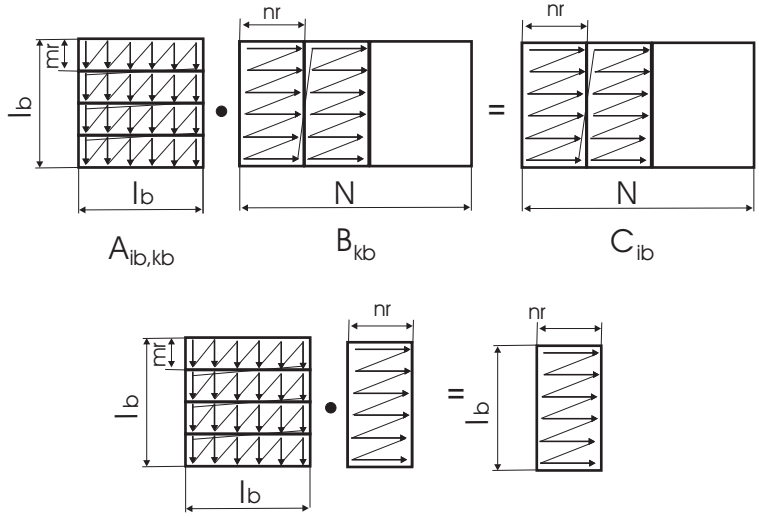
```
for(kb=0; kb<Nb; ++kb)
{
    //Przepakowanie bloku B_kb
    for(ib=0; ib<Nb; ++ib)
    {
          // Przepakowanie bloku A_ib,kb
          C_ib+=A_ib,kb*B_kb
    }
}
```

Tu macierz **A** jest podzielona na bloki kwadratowe o rozmiarze $l_b \times l_b$. Zakładamy, że rozmiar zadania jest wielokrotny do rozmiaru bloku. W przeciwnym wypadku trzeba użyć podkładania (rozdział 2.1.4.3). Macierzy **B** i **C** są podzielone na pasma pionowe o szerokości l_b. Przepakowanie bloków macierzy **A** i **B** jest optymalne (patrz rozdział 2.1.4.5). W pętli wewnętrznej wykonujemy

mnożenie bloków macierzy $\mathbf{C}_{ib}$+=$\mathbf{A}_{ib,kb}$*$\mathbf{B}_{kb}$ przy blokowaniu 2×4 rejestrów XMM (algorytm został podany wyżej). Dla minimalizowania ilości chybień w pamięci podręcznej jest wykonane przepakowanie bloków macierzy (rys. 2.9) według przyjętego schematu blokowania rejestrów XMM.



Rys. 2.8 Podział na bloki według Intel MKL (na górze) oraz podmacierze używane w pętli wewnętrznej przy blokowaniu rejestrów XMM 2×4



Rys. 2.9 Pakowanie danych przy blokowaniu rejestrów XMM m_r×n_r. Z dołu są podmacierze, które powinny być umieszczone w buforze TLB

Z punktu widzenia wydajności obliczeń przy wykonaniu pętli wewnętrznej w cache L1 mają być umieszczone bloki danych, zakreskowane na rys. 2.8 (dół), oraz klatka macierzy $\mathbf{C}_{ib}$ o rozmiarze $m_r \times n_r$. Stąd wynika:

$$l_b(m_r + n_r) + m_r \cdot n_r = L_1 \quad \rightarrow \quad l_b = \frac{L_1 - m_r \cdot n_r}{m_r + n_r} \tag{2.12}$$

Z innej strony rozmiar danych, umieszczonych w cache L2, nie powinien przekroczyć rozmiaru TLB – Translation Lookaside Buffer. To jest cache CPU używany przez narzędzie sterowania pamięcią dla przyspieszenia translacji adresów wirtualnych w trakcie mapowania adresów wirtualnych do pamięci fizycznej. Wiele procesorów w tym procesory x86, używają tego urządzenia. Jeśli obszar danych przekracza rozmiar TLB, dostęp do tych danych idzie znacznie wolniej. Jeśli chybienia przy odczycie w pamięci podręcznej można ukryć za pomocą prefetch'u, to chybień w buforze TLB ukryć nie da się.

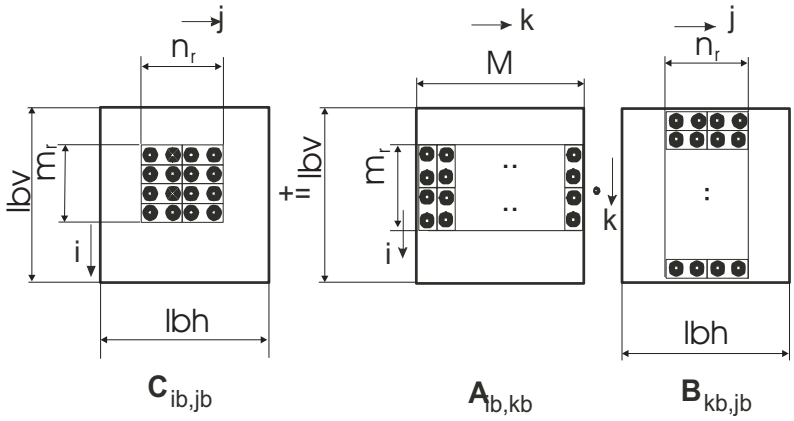
Drugi warunek – nie przekroczenie rozmiaru bufora TLB przy wykonaniu pętli wewnętrznej (dane, oznaczone na poprzednim rys 2.9, dół):

$$l_b^2 + 2n_r l_b = TLB \quad \rightarrow \quad l_b = \sqrt{n_r^2 + TLB} - n_r \tag{2.13}$$

Z tych dwóch warunków trzeba wybrać ten, który doprowadzi do mniejszego rozmiaru l_b. Na przykład: $m_r = 2$, $n_r = 4$, TLB = 256 K (Pentium IV, Pentium Dual-Core), L1 = 32 K. To oznacza, że L1 = 4096 słów double, TLB = 32 768 słów *double*. Ze wzoru (2.12) otrzymujemy $l_b = (4\,096\text{-}2*4)/(2+4) = 681$ oraz z (2.13) wyznaczamy $l_b = \sqrt{[22+32\,768]} - 2 = 179$. Stąd optymalna wartość $l_b = 176$, ponieważ l_b musi być wielokrotna do $n_r = 4$.

Technologie EM64t (system operacyjny 64 bitowy) dopuszczają obecność 16 128-bitowych rejestrów XMM. Więc blokowanie rejestrów oraz pakowanie danych powinny odpowiadać schematowi 4×4 ($m_r = n_r = 4$).

Dla takiego algorytmu są potrzebne 8 rejestrów dla elementów macierzy $\mathbf{C}_{ib}$, 2 – dla elementów macierzy $\mathbf{A}_{ib,kb}$, 1 rejestr – dla elementów macierzy $\mathbf{B}_{kb}$ i 1 rejestr pomocniczy dla przechowywania wyników operacji mnożenia. Na każdej iteracji pętli wewnętrznej mamy 8 ładowań do rejestrów i 32 operacji arytmetyczne, czyli q = 32/8 = 4.



Rys. 2.10 Schemat blokowania 4×4 rejestrów XMM

2.2 Rozwiązywanie układów równań liniowych algebraicznych dla macierzy rzadkich symetrycznych

2.2.1 Metody Gaussa i Choleskiego

Porównamy czas obliczeń programu testowego Gauss, tworzonego przez autora i przeznaczonego dla rozwiązywania układów równań liniowych algebraicznych z macierzą symetryczną gęstą metodami bezpośrednimi. Program wykonuje kilka różnych realizacji metod Gaussa i Choleskiego. Rozmiar zadania wynosi 2 400 równań. Wyniki są w tabeli 2.3.

W pierwszych dwóch wierszach jest przedstawiona metoda Gaussa dla macierzy symetrycznej przy umieszczeniu elementów macierzy w tablice jednowymiarowej wiersz po wierszu i kolumna po kolumnie. Następne dwa wiersze tabeli reprezentują metodę Choleskiego przy umieszczeniu elementów macierzy wiersz po wierszu i kolumna po kolumnie. W ostatnich wierszach są umieszczone wyniki dla blokowej metody Choleskiego przy różnych rozmiarach bloku.

Na rys 2.11 jest przedstawiona wydajność blokowej metody Choleskiego w zależności od rozmiaru bloku. Tamże są wyniki dla metody Gaussa i metody Choleskiego (umieszczone wierz po wierszu). Te metody nie są blokowe, dlatego wyniki nie zależą od rozmiaru bloku l_b.

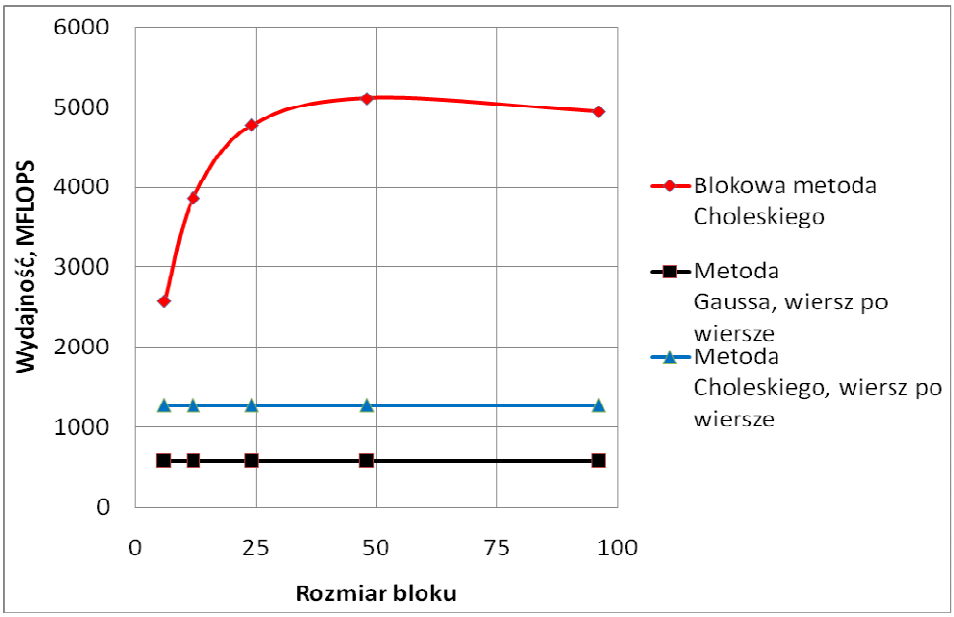
Wydajność metody blokowej przy $l_b \rightarrow 0$ dąży do wydajności metody Gaussa. Maksymalna wydajność dla metody Choleskiego blokowej w dość szerokich granicach wartości l_b bardzo mało zależy od rozmiaru bloku.

Wszystkie metody wykonują tą samą ilość operacji arytmetycznych, jednak demonstrują zupełnie różne wydajności. Powstaje pytanie – dlaczego? Wyjaśnimy to w danym rozdziale.

Tabela 2.3

Porównywanie wydajności rozwiązania układu równań liniowych algebraicznych o rozmiarze 2 400 z macierzą gęstą symetryczną różnymi metodami.

Metoda	FLOP	Czas, s	Wydajność, MFlops
Gauss (LU), wiersz po wiersze	$O(n^3/3)$	7.98	577
Gauss (LU), kolumna po kolumnie		8.71	529
Cholesky (LL^T), wiersz po wiersze		3.63	1 270
Cholesky (LL^T), kolumna po kolumnie		4.22	1 090
Block Cholesky (block 6)		1.79	2 573
Block Cholesky (block 12)		1.2	3 856
Block Cholesky (block 24)		0.97	4 770
Block Cholesky (block 48)		0.9	5 109
Block Cholesky (block 96)		0.93	4 944



Rys. 2.11 Wydajność via rozmiar bloku

Metody bezpośrednie są to takie metody, które doprowadzają do dokładnego rozwiązania (w arytmetyce dokładnej) po wykonaniu skończonej ilości operacji. Taka cecha odróżnia je od metod iteracyjnych, które doprowadzają do rozwiązania przybliżonego, przy czym ilość operacji dla metod iteracyjnych nie jest przewidywana.

Podstawą dla metod bezpośrednich jest faktoryzacja, czyli rozkład macierzy źródłowej w iloczyn macierzy o strukturze specjalnej: dolne trójkątne, górne trójkątne oraz diagonalne.

Załóżmy, że trzeba rozwiązać układ równań liniowych algebraicznych z macierzą dolną trójkątną o wymiarze 3×3:

$$\begin{pmatrix} \lambda_{11} & 0 & 0 \\ \lambda_{21} & \lambda_{22} & 0 \\ \lambda_{31} & \lambda_{32} & \lambda_{33} \end{pmatrix} \cdot \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} . \tag{2.14}$$

Mnożymy pierwszy wiersz macierzy przez wektor **y** : $\lambda_{11} y_1 = b_1 \rightarrow y_1 = b_1 / \lambda_{11}$. Mnożymy drugi wiersz macierzy przez wektor **y** : $\lambda_{21} y_1 + \lambda_{22} y_2 = b_2$. W tym wyrażeniu y_1 już jest znany, dla tego otrzymaliśmy jedno równanie z jedną niewiadomą – $y_2 = (b_2 - \lambda_{21} y_1) / \lambda_{22}$. Mnożymy trzeci wierz macierzy przez wektor **y** : $\lambda_{31} y_1 + \lambda_{32} y_2 + \lambda_{33} y_3 = b_3$. Znów mamy jedno równanie i jedną niewiadomą y_3 ponieważ y_1, y_2 są wiadome z poprzednich kroków – $y_3 = (b_3 - \lambda_{31} y_1 - \lambda_{32} y_2) / \lambda_{33}$. Można udowodnić że dla zadania o dowolnym rozmiarze N $y_k = \left(b_k - \sum_{s=1}^{k-1} \lambda_{k,s} y_s \right) / \lambda_{k,k}$, $k = 1, 2, \ldots, N$.

W podobny sposób rozwiązujemy układ równań liniowych algebraicznych z macierzą górną trójkątną

$$\begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} . \tag{2.15}$$

Zaczynamy od ostatniego równania: $u_{33} x_3 = y_3 \rightarrow x_3$. Przechodzimy dalej do przedostatniego równania $u_{22} x_2 + u_{23} x_3 = y_2 \rightarrow x_2$. Dla pierwszego równania

mamy $u_{11}x_1 + u_{12}x_2 + u_{13}x_3 = y_1 \to x_1$. Dla zadania z N niewiadomymi algorytm jest taki: $x_k = \left(y_k - \sum_{s=k+1}^{n} u_{k,s}x_s \right) / u_{k,k}, \quad k = N, N-1, \ldots, 1$.

Dla macierzy diagonalnej każde równanie zawiera tylko jedną niewiadomą. Dla tego dzielimy każdy z elementów wektora prawej strony przez odpowiedni element diagonalny i otrzymujemy rozwiązanie. Ta procedura otrzymała nazwę skalowania diagonalnego.

Układ równań liniowych algebraicznych

$$\mathbf{Ax} = \mathbf{b} \ , \tag{2.16}$$

rozwiązujemy w kilka etapów. Najpierw wykonujemy faktoryzację macierzy

$$\mathbf{A} = \mathbf{LU} \ , \tag{2.17}$$

gdzie **L** – macierz dolna trójkątna, **U** – górna trójkątna. Podstawiamy w (2.16):

$$\mathbf{L}\underbrace{\mathbf{Ux}}_{\mathbf{y}} = \mathbf{b} . \tag{2.18}$$

Wykonujemy podstawienie bezpośrednie:

$$\mathbf{Ly} = \mathbf{b} \to \mathbf{y} \ . \tag{2.19}$$

Wykonujemy podstawienie wsteczne:

$$\mathbf{Ux} = \mathbf{y} \to \mathbf{x} \ . \tag{2.20}$$

W (2.19) rozwiązujemy układ równań liniowych algebraicznych z macierzą dolną trójkątną, a w (2.20) – z macierzą górną trójkątną.

W taki sposób, rozkład macierzy na iloczyn macierzy dolnej trójkątnej i górnej trójkątnej doprowadza zadanie (2.17) do rozwiązania zadań (2.19) i (2.20) .

Ograniczymy się rozważaniem macierzy symetrycznych. Dla faktoryzacji macierzy symetrycznych i dodatnio określonych najczęściej stosowana jest metoda Choleskiego

$$\mathbf{A} = \mathbf{LL}^T \ , \tag{2.21}$$

a podstawienia bezpośrednie i wsteczne przyjmują postaci

$$\mathbf{Ly} = \mathbf{b} \to \mathbf{y} \ , \tag{2.22}$$

$$\mathbf{L}^T\mathbf{x} = \mathbf{y} \rightarrow \mathbf{x} \ . \tag{2.23}$$

Dla metody Gaussa będziemy przechowywać w pamięci głównej część górną trójkąta

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ & 6 & 7 & 8 & 9 \\ & & 10 & 11 & 12 \\ & & & 13 & 14 \\ & & & & 15 \end{pmatrix},$$

a dla metody Choleskiego – dolną trójkąta

$$\begin{pmatrix} 1 & & & & \\ 2 & 3 & & & \\ 4 & 5 & 6 & & \\ 7 & 8 & 9 & 10 & \\ 11 & 12 & 13 & 14 & 15 \end{pmatrix}$$

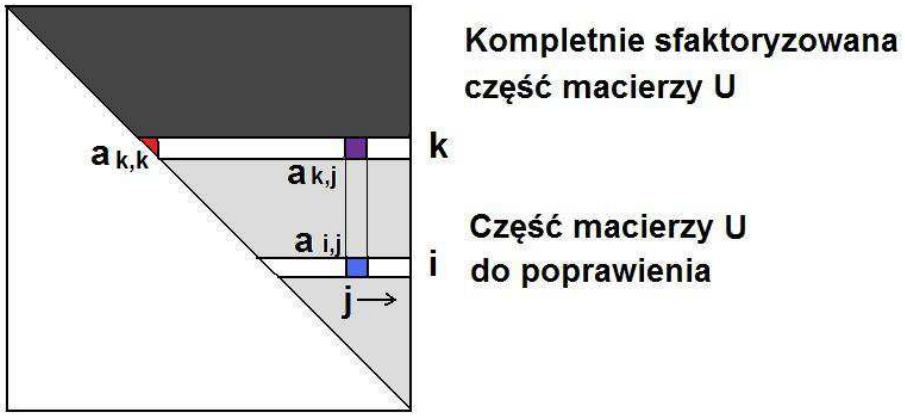
Tu poprzez liczby 1, 2, ... oznaczone są kolejne numery umieszczenia elementów macierzy w tablice jednowymiarowej.

Istnieje kilka realizacji metod Gaussa i Choleskiego [Golub, Van Loan, 1996] . Rozpatrzmy algorytm metody Gaussa, używany dla pierwszego zadania w teście wymienionym powyżej (rys. 2.12).

$k = 1, 2, \ldots, n$
  $piv1 = 1.0 / a_{k,k}$
  $i = k+1, \ldots, n$
    $piv = a_{i,k} \cdot piv1;$ // $\lambda_{i,k} = a_{i,k} / a_{kk}$
    $j = i, \ldots, n$
      $a_{i,j} = a_{i,j} - \lambda_{i,k} \cdot a_{j,k}$
    end j loop
  end i loop
end k loop

Rys. 2.12 Algorytm faktoryzacji macierzy gęstej symetrycznej metodą Gaussa

Załóżmy, że wykonano *k* - 1 pierwszych kroków algorytmu (rys. 2.13). Górna część macierzy – wierszy od 1 do *k* - 1 są sfaktoryzowane. Wykonujemy eliminację kolumny *k* . Wiersz *k* pozostaje bez zmian, a wierszę od *k*+1 do *n* ulegają poprawieniu (pętle po indeksie *i*). Dla wiersza *i* poprawiamy elementy w pętli *j* .



Rys. 2.13 Algorytm faktoryzacji macierzy gęstej symetrycznej metodą Gaussa

W pętli wewnętrznej tego algorytmu (indeks *j*) wykonuje się algorytm *saxpy* (rozdział 2.1.2), ponieważ λ_{ik} nie zależy od indeksu iteracji *j* . Czyli w pętli wewnętrznej jest wykonany algorytm $a_{ij} = a_{ij} + \alpha \cdot a_{kj}$, gdzie α jest stałą, która nie zależy od indeksu *j* . Dla jednej iteracji pętli wewnętrznej są wykonane jedno mnożenie i jedno dodawanie. Przy tym trzeba wykonać dwa odczyty elementów a_{ij}, a_{kj} i jeden zapis elementu a_{ij} . Jeśli wiersz *k* jest umieszczony w pamięci podręcznej, to elementy a_{kj} będą pobierane z pamięci podręcznej. Dla dość dużej macierzy elementy a_{ij} z wierszu *i* za każdym razem trzeba odczytywać z pamięci głównej do pamięci podręcznej, a po modyfikacji – zapisywać do pamięci głównej. Na każde dwie operacji arytmetyczne trzeba wykonać jeden odczyt z pamięci podręcznej, jeden odczyt z pamięci głównej i jeden zapis do pamięci głównej. Jest to podstawowa przyczyna powolnego działania tego algorytmu.

Dla zrealizowanego w teście algorytmu Choleskiego (rys. 2.14, 2.15) przebieg faktoryzacji jest wykonany wierz po wierszu (pętla po *i*) .

$l_{11} = \sqrt{a_{11}}$

$i = 2, \ldots, n$ *// loop over rows*

$j = 1, 2, \ldots, i-1$ *// loop for update of elements in row i*

$S = 0.0;$

$k = 1, 2, \ldots, j-1$

$S = S + a_{i,k} \cdot a_{j,k}$

end k loop

$l_{i,j} = \left(a_{i,j} - S\right) / a_{j,j}; \quad // l_{i,j} = \left(a_{i,j} - \sum_{k=1}^{j-1} a_{i,k} \cdot a_{j,k} \right) \Big/ a_{j,j}$

end j loop

$S = 0.0;$

$k = 1, 2, \ldots, i-1$

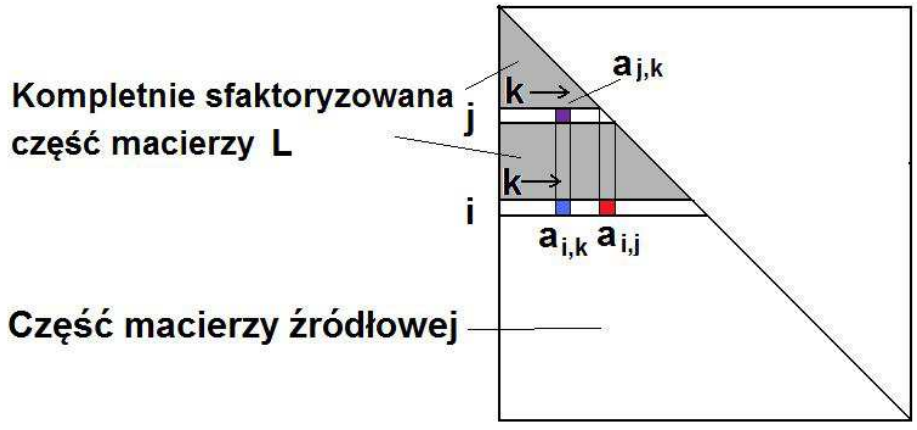
$S = S + a_{i,k} \cdot a_{i,k};$

end k loop

$l_{i,i} = \sqrt{a_{i,i} - S}; \quad // l_{i,i} = \sqrt{a_{i,i} - \sum_{k=1}^{i-1} a_{i,k}^2}$

end i loop

Rys. 2.14 Algorytm faktoryzacji macierzy gęstej symetrycznej metodą Choleskiego

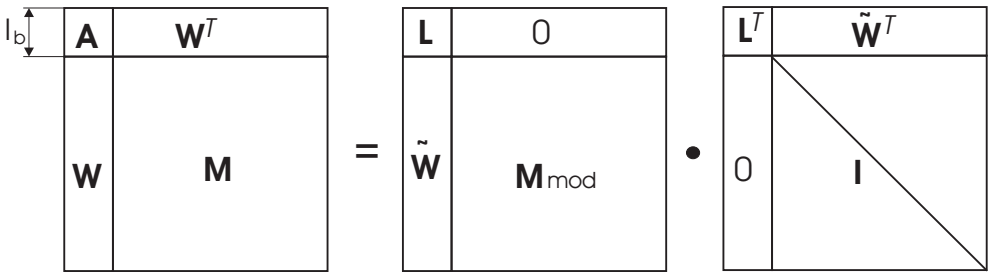


Rys. 2.15 Algorytm faktoryzacji macierzy gęstej symetrycznej metodą Choleskiego

Dla obliczanego elementu l_{ij} bieżącego wiersza *i* jest potrzebny wiersz *j* dla obliczenia sumy $\sum_{k=1}^{j-1} a_{i,k} \cdot a_{j,k}$, która jest liczona w pętli wewnętrznej po indeksie k. Jest to obliczenie iloczynu skalarnego dwóch wektorów, elementy których są pobierane z wierszy *i*, *j* : $S = S + a_{i,k} \cdot a_{j,k}$. W trakcie jednej iteracji pętli wewnętrznej są wykonane dwie operacji arytmetyczne (jedno mnożenie i jedno dodawanie). Jeśli wiersz *i* jest umieszczony w pamięci podręcznej, to na każde dwie operacji arytmetyczne trzeba odczytać element $a_{i,k}$ z pamięci podręcznej i element $a_{j,k}$ z pamięci głównej. Zmienna S jest utrzymywana w rejestrze, dla tego zapis do pamięci głównej nie jest potrzebny. Stąd ten algorytm jest szybszy od algorytmu Gaussa.

2.2.2. Blokowa metoda Choleskiego

Na rys. 2.16 przedstawiony jest jeden krok blokowej faktoryzacji Choleskiego, która polega na podziale macierzy na bloki. Diagonalny blok **A** macierzy źródłowej ma wymiar $l_b \times l_b$, który jest dobierany na podstawie obliczeniowego eksperymentu. Zwykłe optymalny rozmiar l_b leży w granicach 40 – 90 .



Rys. 2.16 Blokowa metoda Choleskiego. Podział macierzy na bloki.

Z prawej strony znajduje się macierz źródłowa **Z**, a z lewej – iloczyn dwóch macierzy, które powstają: **Z** = **B**·**C**, gdzie **B** jest pierwszą macierzą z prawej strony, a **C** – drugą.

Na pierwszym etapie mnożymy pierwszy wierz blokowy macierzy **B** przez pierwszą kolumnę blokową macierzy **C**:

$$\mathbf{A} = \mathbf{L} \cdot \mathbf{L}^T \quad . \tag{2.24}$$

Oznacza to, że blok **A** powinien być sfaktoryzowany, a jako wynik powstaje dolna trójkątna macierz **L**.

Na drugim etapie mnożymy drugi wiersz blokowy macierzy **B** przez pierwszą kolumnę blokową macierzy **C***:*

$$\mathbf{W} = \tilde{\mathbf{W}} \cdot \mathbf{L}^T + \mathbf{M}_{\text{mod}} \cdot 0 \quad \rightarrow \quad \tilde{\mathbf{W}} \cdot \mathbf{L}^T = \mathbf{W} .$$

Transponujemy ostatnie wyrażenie

$$\left(\tilde{\mathbf{W}} \cdot \mathbf{L}^T\right)^T = \left(\mathbf{L}^T\right)^T \cdot \tilde{\mathbf{W}}^T = \mathbf{L} \cdot \tilde{\mathbf{W}}^T = \mathbf{W}^T ,$$

skąd ostatecznie otrzymujemy

$$\mathbf{L} \cdot \tilde{\mathbf{W}}^T = \mathbf{W}^T \quad \rightarrow \quad \tilde{\mathbf{W}} . \tag{2.25}$$

Tu trzeba rozwiązać układ równań liniowych algebraicznych z macierzą dolną trójkątną **L** , jaka była wyznaczona w pierwszym etapie, z paczką prawych stron $\mathbf{W}^T$. Dostajemy paczkę wektorów rozwiązania $\tilde{\mathbf{W}}$.

W trzecim etapie mnożymy drugi wiersz blokowy macierzy **B** przez drugą kolumnę blokową macierzy **C**:

$$\mathbf{M} = \tilde{\mathbf{W}} \cdot \tilde{\mathbf{W}}^T + \mathbf{M}_{\text{mod}} \cdot \mathbf{I} \quad ,$$

skąd znajdujemy dolny blok diagonalny

$$\mathbf{M}_{\text{mod}} = \mathbf{M} - \tilde{\mathbf{W}} \cdot \tilde{\mathbf{W}}^T \quad . \tag{2.26}$$

Po wykonaniu podanego kroku faktoryzacji stosujemy takie same podejście dla macierzy $\mathbf{M}_{\text{mod}}$ – dzielimy je na bloki i powtarzamy te etapy od początku. I tak dalej. W końcu na miejsce macierzy **B** powstaje dolna trójkątna macierz ***L***, a na miejsce **B** – macierz $\boldsymbol{L}^T$.

Faktoryzacja bloku diagonalnego **A** (2.24) odbywa się na poziomie wysokiej wydajności, ponieważ rozmiar l_b jest dość mały i macierz w całości będzie umieszczona w pamięci podręcznej.

Dla rozwiązania układu równań liniowych algebraicznych z paczką prawych stron (2.25) można też stworzyć algorytm wysokiej wydajności. Istotne jest to, że są wykonywane operacje macierzowe, a nie skalarno-wektorowe, jak to się odbywa przy jednej prawej stronie.

Wyrażenie (2.26) otrzymało nazwę obliczenia dopełnienia Schura. Właśnie na ten etap przypada ponad 99% obliczeń dla macierzy o dużym rozmiarze.

Wyrażenie (2.26) zawiera mnożenie macierzy przez macierz. Dla osiągnięcia maksymalnej wydajności stosujemy procedurę DGEMM albo DSYRK z biblioteki Intel MKL.

Jeśli przyjąć, że rozmiar bloku $l_b = 1$, to metoda blokowa będzie doprowadzona do metody klasycznej looking right (rozdział 5.5) , która faktoryzuje macierz kolumna po kolumnie. Są tu wykonane operacje skalarno-wektorowe, i dla tego wydajność takiego algorytmu znajduje się na poziomie metody Gaussa (rozdział 2.2.1) .

3. Elementy oprogramowania równoległego w architekturze SMP na platformie Windows NT

3.1 Tworzenie procesów i wątków. Sekcje krytyczne, zdarzenia i sygnały.

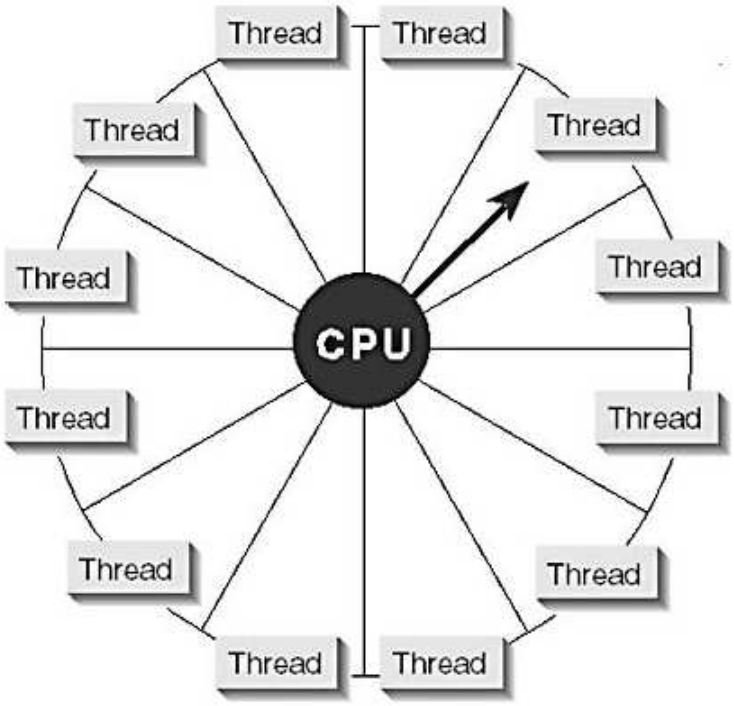
Podany w tym rozdziale materiał opisuje osobliwości tworzenia, sterowania, kasowania oraz synchronizacji procesów i wątków dla systemu operacyjnego Windows platformy NT based (Windows XP, Windows 2003, Windows Vista, Windows 7 itd.). Głównymi źródłami literackimi dla jego napisania posłużyły prace [Richter J. (1997)] oraz [MSDN (2011)].

Proces jest obiektem jądra systemu operacyjnego, który reprezentuje egzemplarz programu uruchomionego. Proces składa się z obiektu jądra, który służy dla kierowania procesem przez system operacyjny, przestrzeni adresowej, która obejmuje kod i dane wszystkich modułów *exe*, *dll* oraz również stosów wątków oraz stos (heap) pamięci, alokowanej dynamicznie.

Proces samodzielnie nie wykonuje nic. Żeby proces zaczął działać koniecznie jest uruchomienie wątku (wątków). Właśnie wątki niosą odpowiedzialność za wykonanie instrukcji kodu, zawartego w przestrzeni adresowej procesu. Każdy wątek ma swój własny stos i w trakcie wykonania instrukcji kodu rozdziela rejestry CPU.

Każdy proces ma przynajmniej jeden wątek, który wykonuje kod w przestrzeni adresowej. Jeśli zamknąć wszystkie wątki procesu, system operacyjny (OS) automatycznie zniszczy ten proces oraz również jego przestrzeń adresową.

Dla wszystkich istniejących wątków system operacyjny planuje pewne odcinki czasu, w skutek czego CPU udziela każdemu wątkowi kwanty czasu. Dla komputera jedno-procesorowego powstaje szeregowanie cykliczne (rys. 3.1 – jest pobrany z [Richter J. (1997)]).



Rys. 3.1 Szeregowanie cykliczne.

Dla komputerów wieloprocesorowych algorytm zrównoważenia obciążenia procesorów jest bardziej skomplikowany.

Kiedy tworzy się proces, system operacyjny tworzy pierwotny wątek. Ten wątek może stworzyć wielu potomnych wątków (child threads), które mogą tworzyć swoje własne potomne wątki.

Tworzenie procesu odbywa się przy wywołaniu funkcji *CreateProcess*:

```
BOOL CreateProcess(
LPCTSTR lpApplicationName,                  // name of exec. module
LPTSTR lpCommandLine,                       // command line string
LPSECURITY_ATTRIBUTES lpProcessAttributes,  // SD
LPSECURITY_ATTRIBUTES lpThreadAttributes,   // SD
BOOL bInheritHandles,                       // handle inheritance option
DWORD dwCreationFlags,                      // creation flags
LPVOID lpEnvironment,                       // new environment block
LPCTSTR lpCurrentDirectory,                 // current directory name
LPSTARTUPINFO lpStartupInfo,                // startup information
LPPROCESS_INFORMATION lpProcessInformation  // process information
);
```

Tu SD oznacza Security Descriptor. Szczegółowy opis tej funkcji znajduje się w [MSDN (2011)]. Najważniejszymi składowymi są *lpCommandLine*, wiersz tekstowy, który zawiera ścieżkę oraz nazwę *exe* module, *lpStartupInfo* – wskaźnik do struktury STARTUPINFO oraz *lpProcessInformation* – wskaźnik do struktury PROCESS_INFORMATION, który zwraca ta funkcja. Poniżej umieszczony jest typowy przykład tworzenia procesu.

```
void CCalcExtern::StartProc()
/*============================================================
  Startujemy nowy proces i zawieszamy bieżący proces dopóki
tworzony proces nie skończy się
  ============================================================*/
 {
      PROCESS_INFORMATION pi;
      STARTUPINFO StartupInfo;
      ZeroMemory(&StartupInfo, sizeof(StartupInfo));
      StartupInfo.cb = sizeof(StartupInfo);
      DWORD dwExitCode;
      LPTSTR lpCommandLine =
                  "Q:\\win32app\\Disp\\Debug\\Disp.exe";
      BOOL fSuccess = CreateProcess(NULL, lpCommandLine,
      NULL, NULL, FALSE, NORMAL_PRIORITY_CLASS, NULL, NULL,
      &StartupInfo, &pi);

      if(!fSuccess)
      {
            AfxMessageBox("StartProc is failed");
            throw EXCEPT_CALC;
      }

      //zamykamy handle wątku tworzonego procesu
      CloseHandle(pi.hThread);

      //zatrzymujemy wykonanie kodu dopóki tworzony proces
      //nie skończy się
      WaitForSingleObject(pi.hProcess, INFINITE);
      //Niszczymy handle tworzonego procesu
      GetExitCodeProcess(pi.hProcess, &dwExitCode);
      CloseHandle(pi.hProcess);
}
```

Handle jest tu typem zmiennej, przeznaczonej dla odróżnienia jednego obiektu OS od drugiego. Za pomocą *handle* można sterować obiektami.

Wywołanie funkcji *CreateProcess* powoduje:

1. Tworzenie obiektu jądra procesu przez system operacyjny i ustawienie licznika obiektu jądra na 1. Jądro procesu – to struktura danych, która służy do kierowania procesem i zawiera statystyczną informację o procesie.

2. Tworzenie wirtualnej przestrzeni adresowej, ładowanie kodu i danych dla *exe* pliku oraz wszystkich *dll* do przestrzeni adresowej procesu.

3. OS tworzy obiekt jądra wątku i ustawia licznik obiektu jądra wątku na 1. Będzie to pierwotny wątek procesu. Jądro wątku jest strukturą danych, która służy do kierowania wątkiem i zawiera statystyczną informacje o nim.

4. Pierwotny wątek zaczyna wykonanie kodu. Jeśli przy tworzeniu procesu występuje sukces, funkcja *CreateProcess* zwraca TRUE.

Istnieje kilka sposobów zakończenia procesu.

1. Funkcja, z której pierwotny wątek zaczyna wykonanie kodu, dochodzi do instrukcji *return*. Jest to najlepszy sposób zakończenia wątku, ponieważ:

- Każdy obiekt C++, tworzony przez ten wątek, będzie zniszczony poprawnie, używając własnego destruktora.
- OS poprawnie zwolni stos wątku.
- OS ustawi kod zakończenia procesu na wartość, którą zwraca funkcja wejściowa wątku pierwotnego.
- Os zmniejszy licznik obiektu jądra procesu o 1.

2. Jeden z wątków procesu wywołuje funkcję *ExitProcess* (w miarę możliwości unikamy tego).

- Część kodu po wywołaniu *ExitProcess* nigdy nie będzie wykonana.
- Wszystkie wątki tego procesu będą przerwane. Chociaż dokumentacja SDK głosi, że wszystkie resursy wątków i procesu (stosy wątków, stos pamięci (heap) oraz pliki) będą zwolnione poprawnie, dla programów C++ może to spowodować, że destruktory klas nie zadziałają. Na przykład [Richter J. (1997)]:

```
#include <windows.h>
#include <stdio.h>
class CSomeObj
{
    public:
    CSomeObj()  { printf("Constructor\r\n"); }
    ~CSomeObj() { printf("Destructor\r\n"); }
};
```

```
CSomeObj g_GlobalObj;
void main ()
{
    CSomeObj LocalObj;
    ExitProcess(0);   // This shouldn't be here
    // At the end of this function, the compiler
    // automatically added  the code necessary
    //to call LocalObj's destructor. ExitProcess prevents
    //it from executing.
}
```

Wydruk na monitorze wygląda tak:

Constructor
Constructor

Potwierdza to, że destruktory obiektów globalnego oraz lokalnego nie zadziałały. Jeśli ExitProcess() usunąć z kodu, wydruk będzie taki:

Constructor
Constructor
Destructor
Destructor

Funkcja *ExitProcess* jest przeznaczona dla kasowania procesu poprzez swoje własne wątki.

3. Jeden z wątków procesu wywołuje funkcję *TerminateProcess* (w miarę możliwości unikamy tego), która jest przeznaczona dla kasowania procesu dowolnym wątkiem (nie tylko własnym). Służy ona dla przymusowego (awaryjnego) zakończenia procesu, kiedy proces w skutek jakichś zdarzeń (błędów) gubi sterowanie i nie odpowiada na notyfikacje.
4. Wszystkie wątki procesu zostały zakończone w skutek wywołania funkcji *ExitThread*(..), *TerminateThread* , system operacyjny niszczy proces. To się zdarza rzadko.

Przy zakończeniu procesu:

- Każdy wątek procesu będzie zakończony.
- Wszystkie obiekty użytkownika, jak również obiekty GDI (graphic device interface), będą zwolnione, z kolei obiekty jądra zostaną zamknięte (zwolnienie pamięci, stosu, zamknięcie plików).
- Obiekt jądra „proces” przechodzi w stan „signaled”.

- Licznik obiektu jądra zmniejsza się o 1.

Przejdźmy do oglądania wątków. Aplikacje, które mają tylko jeden wątek, są aplikacjami jednowątkowymi. Przypuśćmy, że mamy aplikację jednowątkową – okno dialogowe z przyciskiem START. Po naciśnięciu na ten przycisk będą uruchomione jakieś obliczenia, przy czym działają one na tym samym wątku. W takim przypadku wszystkie kontrolki dialogu są zablokowane, dopóki obliczenia nie zakończą się – żaden przycisk dialogowy nie da się nacisnąć. Nie ma możliwości przerwania obliczeń (na przykład, w celu poprawienia danych wejściowych i ponownego uruchomienia). Pozostaje tylko jedna możliwość – przerwać wykonanie całego zadania poprzez menedżer zadań.

Dla poprawienia tej wady trzeba stworzyć aplikację wielowątkową w taki sposób, żeby przy naciśnięciu przycisku START wątek pierwotny stworzył nowy wątek, zadaniem którego będzie wykonywanie obliczeń. Teraz wątek pierwotny obsługuje okienko dialogu, a wątek potomny jedynie liczy. Wątek pierwotny jest wątkiem interfejsowym, ponieważ tworzy się go jako aplikację GDI. Z tego powodu on „widzi" handle okna, a wątek potomny może wysyłać do niego komunikaty. Wysyłanie komunikatów jest możliwe tylko do okna aplikacji, ponieważ ma ono *handle*, który występuje podobnie jak adres, do którego można wysłać komunikat.

W odróżnieniu od wątków interfejsowych wątki robocze nie mają okna aplikacji oraz nie mają *handle*. Dla tego wątek roboczy może wysłać komunikat wątkowi interfejsowemu, ale wątek interfejsowy do wątku roboczego już nie.

Najczęściej takie dwu wątkowe aplikacje są tworzone tak, że wątek potomny, który odpowiada za obliczenia, jest wątkiem roboczym. Czas od czasu wątek roboczy wysyła komunikaty wątkowi interfejsowemu, informując go o stanie wykonania zadania obliczeniowego, a wątek interfejsowy wyświetla na kontrolkach dialogu tą informacje. Wątek interfejsowy jest właścicielem *handle* wątku roboczego, ponieważ sam go stworzył. Przy pomocy tego *handle* wątek interfejsowy może sterować przebiegiem wątku roboczego: zawiesić wątek roboczy przy naciśnięciu przycisku „PAUSE", obudzić wątek roboczy po zawieszeniu, przykleić wątek roboczy (oraz siebie) do procesora fizycznego, zmienić priorytet wątku roboczego, skasować go przy naciśnięciu na przycisk „BREAK" itd.

Aby dialog szybko reagował na polecenia użytkownika, wątkowi obliczeniowemu należy obniżyć priorytet o jeden punkt w stosunku do wątku interfejsowego.

Wielowątkowość – jest to niezbędna składowa obliczeń równoległych dla architektury SMP.

Wątek się składa z:

- Obiektu jądra, będącego strukturą danych mającej za zadanie kierowanie wątkiem i przechowywanie danych statystycznych.

- Stosu wątku, który wspiera parametry wszystkich funkcji, zmiennych lokalnych wątku i kodu zwracanego wątkiem.

Wątek wyznacza kolejność wykonania instrukcji kodu, wykonuje te instrukcje w obszarze przestrzeni adresowej procesu, jak też operuje danymi, umieszczonymi w przestrzeni adresowej procesu. Jeśli mamy kilka wątków tego samego procesu, oznacza to, że te wątki dzielą pomiędzy sobą przestrzeń adresową. Dwa albo więcej wątków mogą wykonywać ten sam kod i operować tymi samymi danymi. Wątki mogą rozdzielać *handles* jądra, dla tego, że tabele *handles* istnieją dla każdego procesu, ale nie dla każdego wątku.

Tworzenie wątku odbywa się przy wywołaniu funkcji *CreateThread* [MSDN, (2011)]:

```
HANDLE CreateThread(
LPSECURITY_ATTRIBUTES lpThreadAttributes,     // SD
SIZE_T dwStackSize,                           // initial stack size
LPTHREAD_START_ROUTINE lpStartAddress,        // thread function
LPVOID lpParameter,                           // thread argument
DWORD dwCreationFlags,                        // creation option
LPDWORD lpThreadId                            // thread identifier
);
```

Najważniejszymi argumentami tu są *dwStackSize* – rozmiar stosu wątku, *lpStartAddress* – wskaźnik do funkcji wątku, *lpParameter* – argument, przekazywany do funkcji wątku, *dwCreationFlags* – flaga, która ustawia, w jakim trybie będzie tworzony wątek. Jeśli *dwCreationFlags = 0* – wątek od razu zacznie wykonywać instrukcje funkcji wątku. Jeżeli *dwCreationFlags = CREATE_SUSPENDED* – wątek będzie tworzony w trybie zawieszonym. Taki wątek trzeba później obudzić przez wywołanie funkcji *ResumeThread* . W przypadku powodzenia funkcja *CreateThread* zwraca *handle* do tworzonego wątku, który służy do sterowania wątków, w przeciwnym wypadku – *NULL*.

Każdy wątek zaczyna wykonanie instrukcji kodu funkcji wątku, wskaźnik do której jest przekazywany argumentem *lpStartAddress* :

```
DWORD WINAPI ThreadFunc(PVOID pvParam)
{
    DWORD rtwResult = 0;
    //Coś wykonuje
    return(dwResult);
}
```

Tu *DWORD* jest synonimem *unsigned long*, *PVOID* – synonimem *void* * . OS Windows wprowadza swoje własne nazwy podstawowych typów danych. Makro *WINAPI* oznacza *Windows Application Program Interface* – ustawia odpowiednie

konwencje wywoływania funkcji [MSDN, (2011)]. *ThreadFunc* jest nazwą funkcji wątku, podana przez programistę. Jeśli tworzy się kilka wątków, przy czym każdy z nich ma wykonywać te same instrukcje kodu, wszystkie te wątki będą używali tej samej funkcji wątku. Przy wykonaniu osobnych instrukcji kodu każdy wątek powinien mieć swoją własną funkcję wątku, przy czym nazwy tych funkcji wątków powinny być unikalne. Wraz z zakończeniem wykonania instrukcji funkcji wątku, działalnie wątku zakończy się też . Funkcja wątku przyjmuje tylko jeden parametr – wskaźnik typu *void* do danych, które muszą być przekazane wątkowi. Jeśli trzeba przekazać więcej danych, należy stworzyć strukturę danych i umieścić te dane, jako składowe struktury.

Przy tworzeniu wątku system operacyjny tworzy obiekt jądra "wątek", przydziela pamięć dla stosu wątku w przestrzeni adresowej procesu. Nowy wątek ma dostęp do wszystkich *handles* obiektu jądra, pamięci procesu oraz stosów innych wątków tego samego procesu. Dla tego w zadaniach wielowątkowych komunikacja pomiędzy wątkami jednego procesu jest stosunkowo prosta.

Podobnie do funkcji *main*, *WinMain* funkcja wątku zwraca kod zakończenia. Funkcja wątku jak najbardziej powinna używać swoich własnych parametrów i zmiennych lokalnych. Do zmiennych statycznych i globalnych jednocześnie mogą próbować uzyskać dostęp kilka wątków. Wtedy dla zabezpieczenia poprawnej wartości tej zmiennej trzeba użyć synchronizacji. Powoduje to komplikację kodu, spadek wydajności obliczeń, zmniejszenie niezawodności programu.

Zmienne lokalne i parametry wątku pozostają umieszczone w jego stosie – lokalne dane różnych wątków są umieszczone w różnych adresach pamięci. Dostęp do tych danych z innego wątku tego samego procesu jest możliwy tylko przez programistę przy napisaniu odpowiedniego kodu. System operacyjny dba automatycznie o wyrównanie obszarów stosów wątkowych do granic cache-linii procesorów. Z tego wynika, że dla danych, które służą do zapisu i są zadeklarowane jako zmienne lokalne wątku, konflikty w pamięci podręcznej nie powstają.

Tablicy o dużym rozmiarze nie da się umieścić w stosie wątku. Najprawdopodobniej będą one alokowane dynamicznie i umieszczone w stosie (heap'ie) procesu.

Przykład tworzenia wątków jest podany w programie *MultThread* (załącznik 3). Struktura *THREAD_DATA* jest przeznaczona dla przekazywania danych wątkom potomnym. W pętli

```
for(iThread=0; iThread<nThreads; iThread++)
```

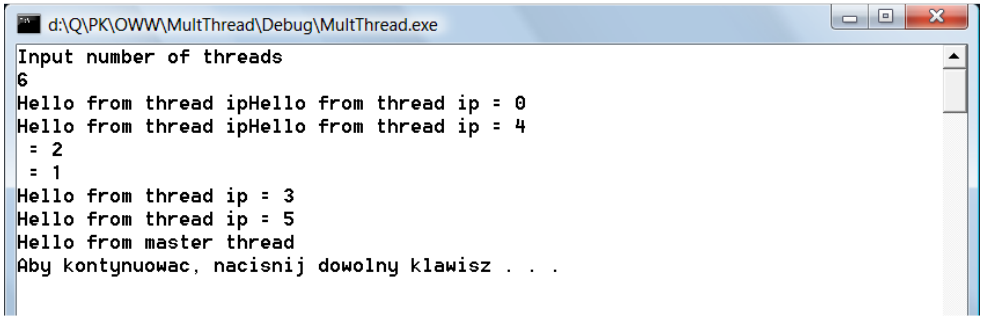
wykonujemy wypełnienie elementu tablicy obiektów typu *THREAD_DATA* o nazwie *tDat*. *tDat[0]* jest przeznaczone dla wątku potomnego o numerze *ip = 0*, *tDat[1]* – dla wątku o numerze *ip = 1* itd.

Po utworzeniu wątków potomnych wątek pierwotny powinien zostać zawieszony. Wywołujemy dla tego funkcje *WaitForMultipleObjects* [MSDN, (2011)].

Wątek potomny po jego utworzeniu (patrz funkcje wątku *ThreadFunc*) pobiera swój numer *ip* i wyprowadza w strumień *stdout* informacje

```
printf("Hello from thread ip");
Sleep(0);
printf(" = %d\n", ip);
```

Ze względu na to, iż kilka wątków w tym samym czasie mogą wyprowadzać swoje dane w ten sam strumień *stdout*, może powstać niespójność danych (rys. 3.2).



Rys. 3.2 Niepoprawny wynik aplikacji wielowątkowej *MultThread* z powodu braku synchronizacji pomiędzy wątkami

Załóżmy, że jeden wątek zaczyna wyprowadzenie danych i wykonuje instrukcję

```
printf("Hello from thread ip");
```

Po wykonaniu tej instrukcji wątek może zostać odłączony od procesora w skutek szeregowania cyklicznego. Drugi wątek zaczyna wyprowadzenie swoich danych i wykonuje tą samą instrukcję. Na monitorze powstaje:

```
Hello from thread ipHello from thread ip
```

Teraz drugi wątek może skończyć wykonanie swojego kodu przed tym, jak zostanie odłączony od procesora. Możliwe jest, że znów pierwszy wątek będzie podłączony do procesora i zakończy wyprowadzanie swoich danych, a także możliwe jest, że trzeci wątek opanuje wspólny resurs *stdout*. Zachowanie programu staje nieprzewidywanym. Na komputerach jednoprocesorowych taki błąd nie występuje przy każdym uruchomieniu. Powstaje wrażenie, że program działa

poprawnie. Dla tego, żeby sprowokować niepoprawne działanie, wstawiamy pomiędzy dwoma instrukcjami *printf* instrukcję *Sleep(0)* , która powoduje zatrzymanie wykonania wątku o ilość milisekund, podanych jako argument faktyczny. Przy tym wątek będzie odłączony od procesora przez *0* milisekund. Zdarza się, że od razu ten sam wątek znów zostaje podłączony do tego samego procesora, lecz nie zawsze tak się dzieję.

Jeśli program jest napisany poprawnie, to żadna prowokacja nie powinna doprowadzić do niestabilnego i niepoprawnego działania.

Dla poprawienia kodu trzeba umieścić wspólny resurs – strumień *stdout* – w sekcją krytyczną. Jest to fragment kodu, dostępny tylko jednemu wątkowi. Wejście do sekcji krytycznej jest kontrolowane przez instrukcję

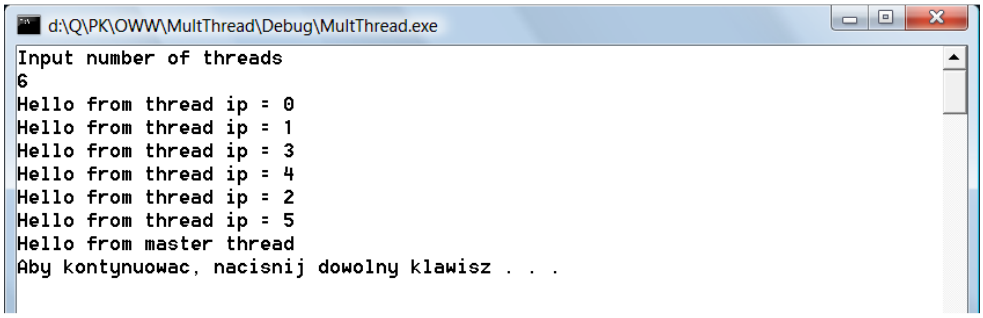
```
EnterCriticalSection(&cs);
```

a wyjście –

```
LeaveCriticalSection(&cs);
```

Jeśli resurs *stdout* jest wolny, to zajmuje go dowolny wątek. Żaden wątek z pozostałych nie może wejść do sekcji krytycznej dopóki wątek, który zajmuje tą sekcję, nie zwolni ją. Tylko wtedy inny wątek może zająć tą samą sekcję krytyczną.

Wyniki poprawionego kodu są podane na rys. 3.3 .



Rys. 3.3 Poprawny wynik aplikacji wielowątkowej *MultThread* – dostęp do wspólnego resursu *stdout* jest chroniony sekcją krytyczną

Wykonanie wątku można zakończyć na kilka sposobów.

1. W chwili, gdy funkcja wątku skończy się – watek samo likwiduje się. Jest to najbardziej pożądany sposób.
   - Wszystkie obiekty C++ będą poprawnie zniszczone przez wywołanie destruktorów.

- OS poprawnie zwalnia pamięć, zajmowaną stosem wątku.
- OS ustawia kod zakończenia wątku, który zwraca funkcja wątku.
- OS zmniejsza licznik wątku o 1.

2. Wątek sam siebie zniszczy, wywołując funkcje *ExitThread* [MSDN, (2011)]. Wszystkie resursy wątku będą poprawnie zwolnione, jednak destruktory klas C++ nie zostaną wywołane. Dla tego należy stworzyć funkcję, która zwalnia pamięć dynamicznie alokowaną wątkiem, zamyka pliki itd. Najpierw wywołujemy tę funkcję, a później – *ExitThread*.
3. Wątek pozostaje zniszczony innym wątkiem tego samego lub innego procesu przez wywołanie funkcji *TerminateThread* [MSDN, (2011)]. Ten sposób jest używany, kiedy wątek zawiesił się i nie odpowiada na żadną notyfikacje. Wywołanie *TerminateThread* nie komunikuje wątkowi to, że zostanie on zniszczony, więc wątek sam nie jest w stanie zwolnić resursy. *TerminateThread* jest używamy również, jeśli na żądanie użytkownika działanie wątku roboczego (obliczeniowego) trzeba przerwać z wątku interfejsowego. Wtedy dla poprawnego zamykania każdego obiektu pamięć, alokowana dynamicznie, musi być zwolniona przez wywołanie funkcji specjalnej – dealokatora. Tak samo trzeba zrobić i przy zamykaniu plików. Procedura niszczenia wątku roboczego wygląda tak: zawieszamy wątek roboczy poprzez wywołanie funkcji *SuspendThread* , wywołujemy z wątku interfejsowego dealocator wątku roboczego i tylko po tym niszczymy go, wywołując *TerminateThread* .
4. Proces, który jest właścicielem wątku, kończy swoje działanie.

Przy zakończeniu wątku:

- Wszystkie *handles* obiektów użytkownika, który należą do wątku, pozostają zwolnione przez OS. Większość obiektów należą do procesu i będą zwolnione przy jego zakończeniu.
- Kod zakończenia wątku pozostaje zmieniony z STILL_ACTIVE na kod, przekazany w funkcje *ExitThread* lub *TerminateThread*.
- Obiekt jądra „wątek” przechodzi w stan wolny – wątek podaje sygnał o zakończeniu.
- Licznik obiektu jądra „wątek” zmniejsza się o 1.

Każdy wątek ma swój własny zestaw rejestrów procesora – kontekst wątku. Kontekst odbija stan rejestrów procesora w momencie ostatniego wykonania wątku i zapisuje go w strukturze CONTEXT, która znajduje się w obiekcie jądra „wątek”. Najważniejsza informacja w kontekście wątku – to stan wskaźnika rozkazów IP i

wskaźnika stosu SP, które wskazują na odpowiednie adresy pamięci przestrzeni adresowej procesu.

Zawieszenie i obudzenie wątku jest wykonane przez wywołanie funkcji

```
DWORD SuspendThread(HANDLE hThread);
DWORD ResumeThread(HANDLE hThread);
```

Wątek może zawiesić sam siebie, z kolei ponowić jego może tylko inny wątek. *SuspendThread* może zawiesić wykonanie programu. Na przykład, wątek zaczął alokować pamięć dynamicznie, zablokował dostęp do stosu procesu (heap'u), i w tym momencie pozostał zawieszony. Heap zablokowany – żaden inny wątek nie jest w stanie wydzielić pamięć dynamicznie, dopóki nie będzie ponowiony wątek, który „utrzymuje" heap. Podobna sytuacja otrzymała nazwę *dead lock* .

Funkcja *int GetThreadPriority(HANDLE hThread)* zwraca bieżący priorytet wątkowi. Zmienić priorytet wątku można przy pomocy funkcji *BOOL SetThreadPriority(HANDLE hThread, int nPriority)* . Os wspiera takie priorytety wątków:

THREAD_PRIORITY_IDLE,
THREAD_PRIORITY_LOWEST,
THREAD_PRIORITY_BELOW_NORMAL,
THREAD_PRIORITY_NORMAL,
THREAD_PRIORITY_ABOVE_NORMAL,
THREAD_PRIORITY_HIGHEST,
THREAD_PRIORITY_TIME_CRITICAL .

Wątek z większym priorytetem wywłaszcza wątki z mniejszym priorytetem. Dopóki wątek z większym priorytetem nie jest w stanie oczekiwania, CPU nie będzie przydzielać czas wątkom z mniejszym priorytetem. Wątki OS mają największy priorytet.

3.2. Wielowątkowość, prawo Amdahl'a, zrównoważenie procesorów, ziarnistość.

Rozważmy przykład. Przeglądanie stron internetowych przez przeglądarkę zajmuje dość sporo czasu. Istnieją jednak pewne sposoby przyspieszenia [Grama A., Gupta A., Karypis G., Kumar V., 2003] takie jak:

- Przewidujemy, które strony będą następne, i przygotujemy zadanie na pobieranie z wyprzedzeniem. Jest to podobne do techniki prefetch'u .

- Uruchamiamy kilka przeglądarek jednocześnie, i dopóki jedna strona ładuje się, pracujemy z inną stroną. Jest to technika, podobna do wielowątkowości (multithreading).

Wielowątkowość jest używana dla ukrycia zwłoki (hiding of latency). Na 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 zapisie / odczycie na /z dysk(u) dużego pliku – asynchroniczne wejście – wyjście [MSDN, (2011)].

Wielowątkowość może być zrealizowana na podstawie bibliotek wielowątkowych (Win 32 platforma – SDK, Posix Thread library), na podstawie dyrektyw kompilatora (OpenMP) oraz przy użyciu TBB – Threading Building Blocks [TBB].

Oprogramowanie na podstawie bibliotek wielowątkowych wymaga od programisty ręcznego tworzenia wątków, rzutowania na nich zadania równoległe, wprowadzania synchronizacji na dolnym poziomie, jawnego wyznaczania klasy pamięci dla danych (stos wątku, pamięć alokowana dynamicznie – heap procesu, TLS – Thread Local Storage [MSDN, 2011], pamięć dla zmiennych statycznych i globalnych itd.), przyklejania wątków do fizycznych procesorów, zawieszania, budzenia i niszczenia wątków, zmieniania ich priorytetów i wiele innych czynności. Taki sposób oprogramowania daje możliwość tworzenia elastycznych, wysoko wydajnych kodów.

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) daje możliwość unikania komplikacji programu poprzez ukrycie wielu szczegółów, które są niezbędne przy użyciu 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 jest modelem wielowątkowym, który nadaje możliwość zrealizować połączenie widłowe (fork-joint) pomiędzy wątkami typu master-worker. To oznacza, że zakres stosowania OpenMP jest ograniczony w stosunku do jawnego oprogramowania wielowątkowego.

Threading Building Blocks [TBB] jest narzędziem dla zrównoleglenia na podstawie wielowątkowości programów, napisanych w C++. Zalety i wady są podobne do OpenMP.

Będziemy rozróżniać takie typy zrównoleglenia: *zrównoleglenie funkcjonalne* (zrównoleglenie zadań) oraz *zrównoleglenie danych*. Przy zrównolegleniu zadań różne wątki wykonują różne instrukcje kodu (algorytmy, zadania), a przy zrównolegleniu danych – ten sam zestaw instrukcji (fragment kodu). Typowym przykładem zrównoleglenia zadań jest procesor tekstowy. Takie zadania, jak sprawdzenie pisowni, czas od czasu zapis lub 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. Drugi typowym przykładem zrównoleglenia zadań jest zadanie dwu wątkowe, przy czym jeden z wątków obsługuje interfejs, a drugi – roboczy (obliczeniowy).

Przy zrównolegleniu danych ten sam algorytm (zestaw instrukcji) jest wykonany dla różnych danych. Dalej w obrębie tego kursu będziemy zajmować się zadaniami zrównoleglenia danych.

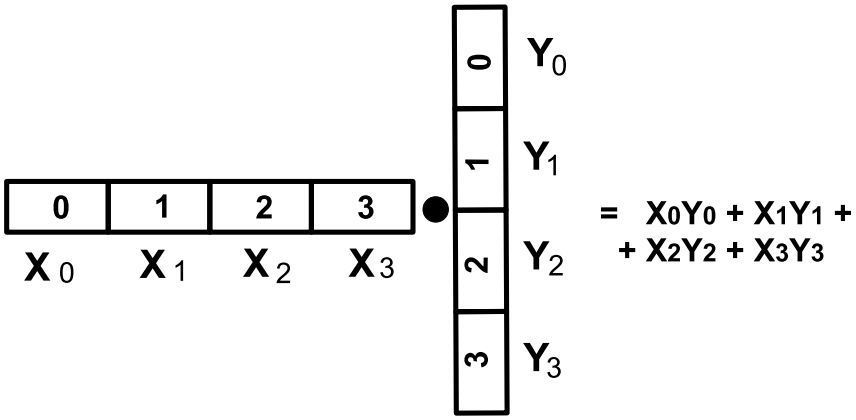
W algorytmie obliczenia iloczynu skalarnego dwóch wektorów przy równomiernym podziale danych pomiędzy wątkami równomierne obciążenie pomiędzy procesorami będzie zabezpieczone. To oznacza, że wszystkie wątki zaczną i skończą swoją prace mniej więcej jednocześnie. Przykład podziału danych na cztery wątki jest przedstawiony na rys. 3.4 .

Obliczenie iloczynów $\mathbf{x}_0^T\mathbf{y}_0, \mathbf{x}_1^T\mathbf{y}_1, \ \ldots\ , \mathbf{x}_3^T\mathbf{y}_3$ nie zawiera zależności pomiędzy danymi odpowiednich części tablic **x**, **y** i dla tego można je wykonać równoległe. Sumowanie wyników obliczenia częściowych iloczynów skalarnych powinno być wykonane w trybie sekwencyjnym. Matematyczne sformalizowanie problemu przedstawiono poniżej:

$$dot = \mathbf{x}^T\mathbf{y} = \sum_{ip=0}^{np-1} \left(\sum_{i=\frac{N}{np}ip+1}^{\frac{N}{np}(ip+1)} x_i y_i \right) . \qquad (3.1)$$

Nie zmniejszając ogólności metody zakładamy, że *N* jest wielokrotną ilością wątków *np*. W przeciwnym wypadku trzeba minimalnie zwiększyć rozmiar tablic **x**, **y** tak, żeby nowy rozmiar tablicy był wielokrotny do *np* , a powstałe w skutek takiego rozszerzenia elementy tablic były zerowe. Takie podejście daje możliwość stosowania (3.1), nie komplikując algorytmu. Wzór (3.1) doprowadza do wyniku poprawnego:

$$dot = \mathbf{x}^T \mathbf{y} = \sum_{ip=0}^{np-1} \left(\sum_{i=\frac{N}{np}ip+1}^{\frac{N}{np}(ip+1)} x_i y_i \right) = \sum_{i=1}^{N} x_i y_i \quad . \tag{3.2}$$

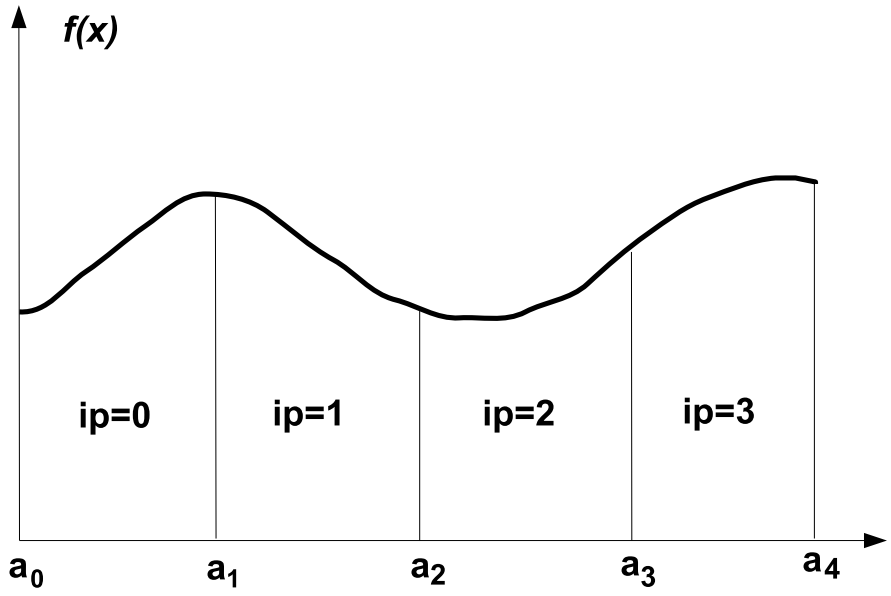


Rys. 3.4 Podział danych pomiędzy wątkami dla algorytmu $dot = \mathbf{x}^T \cdot \mathbf{y}$

Drugi przykład zadania zrównoleglenia danych – to obliczenie numeryczne całki metodą trapezów. Matematyczną podstawą dla zrównoleglenia jest to, że

$$\int_a^b f(x)dx = \sum_{ip=0}^{np-1} \left(\int_{a_{ip}}^{a_{ip+1}} f(x)dx \right) \quad . \tag{3.3}$$

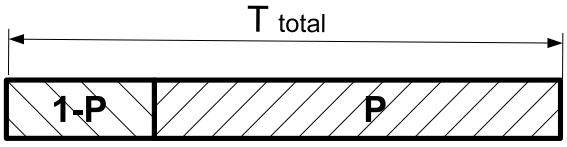
Tu $a_0 = a$, $a_{\text{np}} = b$. Teraz każdą całkę $\int_{a_{ip}}^{a_{ip+1}} f(x)dx$ można policzyć niezależnie od innych i to daje możliwość liczyć każdą całkę na osobnym wątku (rys. 3.5).



Rys. 3.5 Podział danych pomiędzy wątkami dla algorytmu obliczenia całki metodą trapezów

Zadania, które polegają na zrównolegleniu danych, można rozwiązywać na podstawie OpenMP oraz również na podstawie bibliotek wielowątkowych.

Dla oszacowania możliwości przyspieszenia przy zwiększeniu ilości procesorów stosuje się *prawo Amdahl'a* [Amdahl, Gene, 1967]. Jest ono podstawą teoretyczną dla wyznaczenia górnej granicy wydajności algorytmów równoległych przy zrównolegleniu danych. Zakładamy, że w podanym algorytmie część operacji nad danymi pozostała wykonana w trybie sekwencyjnym, a część – w trybie równoległym (rys. 3.6).



Rys. 3.6 Podział operacji, wykonanych algorytmem, na części równoległą (P) oraz sekwencyjną (1-P)

Tu P – cześć pracy obliczeniowej, która jest wykonana w trybie równoległym, przy czym $0 \le P \le 1$. Wtedy część pracy obliczeniowej, wykonanej w trybie sekwencyjnym, wynosi 1 - P . Prawo Amdalh'a formułuje się następująco:

$$T_{parall} = \left[(1-P) + \frac{P}{n_p} \right] T_{total} + O_{np} \ , \tag{3.4}$$

gdzie T_{total} – czas wykonania algorytmu w trybie sekwencyjnym, T_{parall} – czas wykonania algorytmu przy zastosowaniu zrównoleglenia, n_p – ilość procesorów, O_{np} – wydatki na zrównoleglenie.

Jeśli pominąć wydatki na zrównoleglenie O_{np}, to

$$S_p = \frac{T_{total}}{T_{parall}} = \frac{1}{(1-P) + \frac{P}{n_p}} \ , \tag{3.5}$$

gdzie S_p – przyspieszenie przy zwiększeniu ilości procesorów, albo speed up. Prawo Amdalh'a ustawia górną teoretyczną granicę przyspieszenia dla danego algorytmu przy podanej ilości procesorów. Jeśli przyjąć, że $n_p \to \infty$, to otrzymamy największe możliwe przyspieszenie dla danego algorytmu:

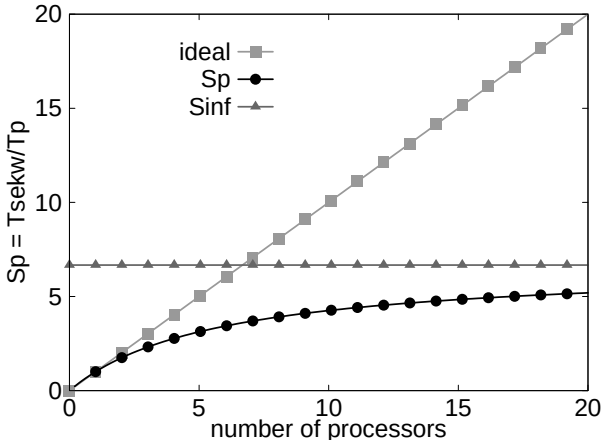
$$S_{\inf} = \frac{T_{total}}{T_{Parallel}} = \frac{1}{(1-P)} \tag{3.6}$$

Rozważmy przykład, w którym dola pracy obliczeniowej w trybie sekwencyjnym w algorytmie wynosi 15%. Oznacza to, że P = 0.85 (rys. 3.7).

Tu T_{sekw} – czas wykonania zadania algorytmem sekwencyjnym, T_p – algorytmem równoległym na n_p procesorach. Często zamiast T_{sekw} jest używane T_1 – czas wykonania zadania algorytmem równoległym na jednym procesorze.

Dla podanego przykładu górna teoretyczna granica przyspieszenia wynosi 6.67. Oznacza to, że nie istnieje takiego sprzętu, na którym by udało się osiągnąć speed up, większy od tej wartości. Z innej strony, na realnym komputerze przyspieszenie może być tylko mniejsze od tego, jakie otrzymujemy na podstawie prawa Amdalh'a, ponieważ prawo Amdalh'a ignoruje osobliwości sprzętu i oszacowuje możliwość przyspieszenia podanego algorytmu wychodząc tylko z samego algorytmu. Jeśli na podstawie prawa Amdalh'a podany algorytm praktycznie nie nadaje się do przyspieszenia, to żaden superkomputer jego nie przyspieszy w skutek zrównoleglenia. Jeśli prawo Amdalh'a mówi nam, że podany algorytm można wspaniałe przyspieszyć, to nie oznacza, że na dowolnym komputerze osiągniemy istotne przyspieszenie. Typowym przykładem jest algorytm obliczenia

iloczynu skalarnego dwóch wektorów. Dla dużych tablic **x**, **y** , które nie mieszczą się w pamięci podręcznej, przyspieszenie algorytmu przy zwiększeniu ilości procesorów na komputerach klasy PC jest słabe (rozdział 3.4), chociaż na podstawie prawa Amdalh'a graniczny speed up jest bardzo wysoki ponieważ część operacji sekwencyjnych jest wielokrotnie mniejsza od części operacji równoległych. Przyczyną takiej rozbieżności pomiędzy oszacowaniem teoretycznym a wynikami eksperymentu numerycznego jest słaba przepustowość układu pamięci.



Rys. 3.7 Prawo Amdalh'a (P = 0.85)

Rozważmy podstawowe etapy projektowania algorytmów równoległych dla zadań zrównoleglenia danych.

Zrównoważenie obciążenia wątków (loading balance) jest bardzo ważne dla osiągnięcia wysokiej wydajności. Głównym celem jest tu minimalizacja czasu *idle* (czasu jałowego) dla każdego wątku.

Przyczyny *idle* są różne – pobieranie danych z pamięci, operacji wejścia/wyjścia, oczekiwanie na jakieś zdarzenie, lock przy synchronizacji, itd. Trzeba zminimalizować czas *idle* dla każdego wątku. Techniki przyspieszenia – to asynchroniczne wejście/wyjście (I/O), pobieranie danych z pamięci z wyprzedzeniem (*prefetch*), uporządkowanie pobierania danych dla ułatwienia używania pamięci podręcznej (usunięcie skoków, podział danych na bloki), wspieranie potokowego przetwarzania danych (rozwijanie pętli) itd.

Dla zabezpieczenia poprawności danych globalnych koniecznie jest użycie synchronizacji. Synchronizacja może istotnie zmniejszyć wydajność kodu i jego zdolność do przyspieszenia przy zwiększeniu ilości procesorów (speed up). Dla tego algorytmy równoległe trzeba projektować tak, żeby zminimalizować ilość zdarzeń komunikacyjnych. W związku z tym jak najbardziej używamy danych

lokalnych wątku umieszczonych w jego stosie. Przy konieczności dokonania zapisu w dużą tablicę, której nie można umieścić w stosie wątku, warto użyć techniki Thread Local Storage (TLS – rozdział 3.3) przy dynamicznym alokowaniu pamięci.

Jeśli jest konieczne zastosowanie synchronizacji, sensowne jest użycie sekcji krytycznych, ponieważ sekcje krytyczne działają w trybie użytkownika (user mode) i dla tego są znacznie szybsze od mutex'ów, które działają w trybie jądra systemowego (kernel mode) [Richter, 1997]. Kiedy wątek próbuje wstąpić w obszar, kontrolowany przez sekcję krytyczną, która jest zajęta innym wątkiem, OS przeprowadza jego z trybu użytkownika do trybu jądra – wątek przechodzi w stan oczekiwania. Procesor na to zużywa ~1000 taktów – to dość wysoka cena. Na komputerach wieloprocesorowych często się zdarza, że resurs blokowany sekcją krytyczną, pozostanie zwolniony wcześniej, niż wątek będzie wprowadzony w tryb jądra i inny procesor odnowi wykonanie tego wątku. Wiele taktów procesora zużyto na wprowadzenie wątku w tryb jądra i odwrotnie – w tryb użytkownika. Dla tego i wymyślono spin-blokowanie – jest to skuteczne, jeśli wspólny resurs pozostaje blokowany w ciągu krótkiego odcinku czasu, który jest rzędu przeprowadzenia wątku w tryb jądra i odwrotnie. Przy spin-blokowaniu wątek, który pragnie zająć resurs zawarty w sekcję krytyczną, kilkakrotnie próbuje wejść w chroniony obszar kodu. Jedynie, kiedy ustawiona z góry ilość takich prób zakończy się niepowodzeniem (obiekt chroniony przez sekcje krytyczną nadal będzie zajęty), wątek zostanie wprowadzony w tryb jądra.

Poważną cechą algorytmów równoległych jest rozdrobnienie (ziarnistość, granularity) – jakościowa miara stosunku obliczeń do komunikacji. Wyróżnia się grube rozdrobienie (coarse granularity) – kiedy znaczna część roboty obliczeniowej jest wykonana pomiędzy zdarzeniami komunikacyjnymi, a także drobne rozdrobienie (fine granularity) – gdy odnośnie niewielka część roboty obliczeniowej jest wykonana pomiędzy zdarzeniami komunikacyjnymi.

Projektowanie aplikacji wielowątkowych składa się z:

- Rozdzielenia zadania na tyle drobnych zadań równoległych, na ile jest to możliwe (partitioning). Jednym z celów jest zabezpieczenie zrównoważonego obciążenia wątków (load balance). Dla wielu algorytmów z góry nie jest jasne, jak trzeba podzielić dane pomiędzy wątkami żeby zabezpieczyć to zrównoważenie. Dla takich algorytmów im drobniej podział, tym łatwiej to osiągnąć. Z drugiej strony zbyt drobny podział powoduje duże wymagania OS na tworzenie i obsługę wątków. Dla tego trzeba odnaleźć optymalny balans pomiędzy wymaganiami OS a zrównoważeniem wątków. Dla algorytmów prostych (obliczenie iloczynu skalarnego dwóch wektorów, obliczenie całki itd.) równomierny podział danych jest oczywisty, więc preferowane jest rozdrobienie grube (coarse granularity).
- Projektowania komunikacji z punktu widzenia danych i synchronizacji.

- Rozważań wydajności algorytmu.

Rozpatrzmy przykład [Grama A., Gupta A., Karypis G., Kumar V, 2003] – metodę iteracyjną dla rozwiązania układu równań liniowych algebraicznych. Stan danych na kroku iteracyjnym k+1 zależy od stanu 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 przetworzenia jednej porcji danych jest mniejszy od narzutu, który system operacyjny zużywa na tworzenie i obsługę wątku, zwiększenie ilości wątków będzie powodowało spadek wydajności. Jest to przykład zbyt drobnego rozdrobienia.

Tworzenie wątków wymaga pewnego nakładu czasu – dla OS Windows zajmuje to odcinek czasu, równoważny ~1000 podziałów 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 zaprojektowany, żeby zminimalizować nakłady OS, które ograniczają skalowalność – zdolność wykonywania większą robotę obliczeniową przy zwiększeniu ilości procesorów systemu obliczeniowego.

Przy wielokrotnym wchodzeniu w równoległy region (regiony) zwykle jest tworzony pool wątków. Oznacza to, że wątki tworzą się tylko przy pierwszym wejściu do równoległego regionu, a przy wyjściu one nie będą niszczone, tylko wprowadzone w stan idle. Przy następnym wejściu do regionu równoległego wątki budzą się. Obudzenie wątków zajmuje mniej czasu w porównaniu z ich tworzeniem.

3.3. Bezpieczne równoległe alokowanie pamięci, technika TLS. Osobliwości pracy z pamięcią podręczną. Zmienne lokalne i globalne.

Dynamiczne alokowanie pamięci w stosie (heap'ie) procesu jest procedurą kosztowną, która wymaga synchronizacji wątków przy dostępie do wspólnego resursu – heap'u procesu. 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. Jeżeli inne wątki też mają alokować pamięć w tym samym heap'ie, będą czekać dopóki pierwszy wątek nie zakończy alokować pamięć.

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

```
HANDLE HeapCreate(HEAP_NO_SERIALIZE ,  0, 0);
```

Jeśli *handle* tego heap'u przechowywać w pamięci lokalnej wątku (TLS – Thread Local Storage), ten heap można używać dla dynamicznego alokowania i zwolnienia pamięci danego wątku.

Najbardziej sensowne jest umieszczenie tej procedury w DLL [Intel - heap, (2011)]:

```
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);
}
```

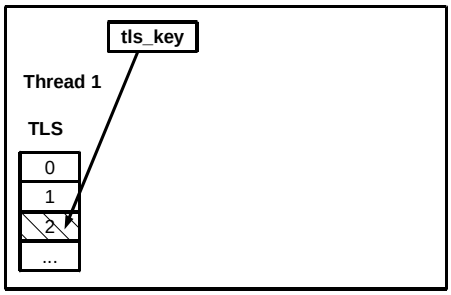
Tu każdy wątek tworzy swój własny heap, *handle* którego jest umieszczony w slot TLS o numerze *tls_key*. Każdy wątek dynamicznie alokuje pamięć w swoim własnym heap'ie, *handle* którego jest pobierany przez zmienną globalną *tls_key* . Przy tym nie powstaje problemu wspólnego heap'u dla różnych wątków.

Zmienne statyczne i globalne są wspólne dla każdego wątku danego procesu. TLS nadaje możliwość stworzenia dla każdego wątku unikalnych danych, do których proces może mieć dostęp, przy użyciu indeksu globalnego. Jeden wątek alokuje indeks, który może być użyty przez inny wątek celem dostępu do unikalnych danych związanych z pierwszym wątkiem.

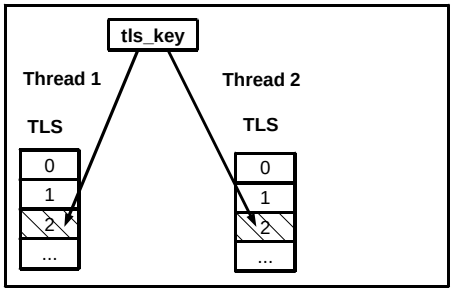
Kiedy wątki są tworzone, OS alokuje tablicę typu LPVOID dla TLS indeksów i inicjuje ją, jako NULL. Przed tym, jak użyć indeksu, powinien on zostać alokowany jednym z wątków przez wywołanie funkcji *TlsAlloc()*.

Każdy wątek przechowuje swoje dane, które odpowiadają TLS indeksowi, w TLS slocie. Dla tego dane, związane z TLS indeksem, muszą być rzutowane do typu LPVOID.

Schemat podany na rys. 3.8 – 3.13, wyjaśnia działanie fragmentu kodu, przedstawionego powyżej.



Rys. 3.8 Wątek 1 procesu alokuje TLS indeks – w tablice TLS wydziela się pozycję tls_key = TlsAlloc(); Zmienna globalna tls_key otrzymuje numer elementu w tablice (tls_key = 2)

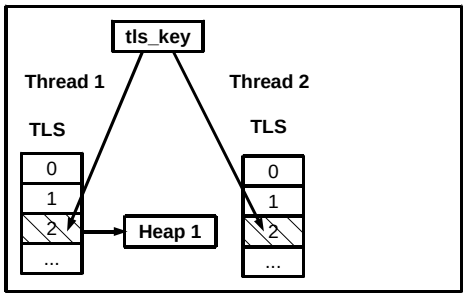


Rys. 3.9 Powstaje wątek 2, który ma swój własny slot TLS, numery elementów aktualnych w tym slocie są takie same

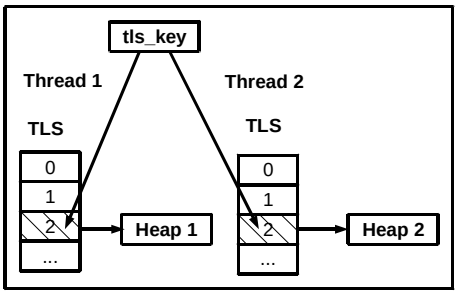
Przy podłączeniu DLL do procesu (case DLL_PROCESS_ATTACH) wątek pierwotny alokuje TLS indeks (rys. 3.8) oraz umieszcza w TLS slot o numerze tls_key *handle* domyślnego heap'u procesu: TlsSetValue(tls_key, GetProcessHeap()); . Przy tworzeniu każdego wątku potomnego Thread 1, Thread 2 (case DLL_THREAD_ATTACH) są tworzone własne heap'y

```
HANDLE heap_1 = HeapCreate(HEAP_NO_SERIALIZE, 0, 0);
HANDLE heap_2 = HeapCreate(HEAP_NO_SERIALIZE, 0, 0);
```

Wątki 1, 2 umieszczają *handles* dla swoich własnych heap'ów w pozycjach tls_key odpowiednich TLS slotów za pomocą funkcji TlsSetValue(tls_key, heap_i), gdzie heap_i = heap_1 dla pierwszego wątku oraz heap_i = heap_2 dla drugiego (rys. 3.10, 3.11).



Rys. 3.10 Wątek 1 łączy element TLS tablicy o numerze tls_key z HANDLE heap_1: TlsSetValue(tls_key, HANDLE heap_1)

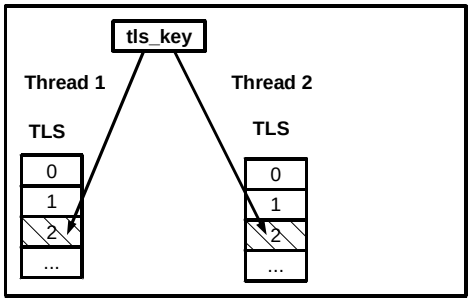


Rys. 3.11 Wątek 2 łączy element TLS tablicy o numerze tls_key z HANDLE heap_2: TlsSetValue(tls_key, HANDLE heap_2)

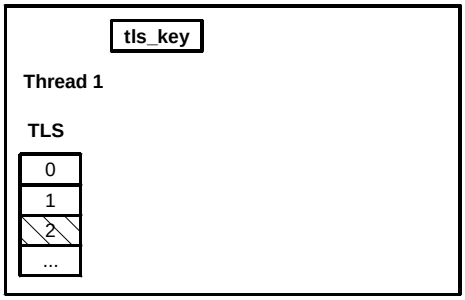
Teraz w każdym wątku mamy dostęp do swojego własnego heap'u przez zmienną globalną tls_key (patrz funkcje *thr_malloc*, *thr_free*) :

```
HANDLE heap_1 = TlsGetValue(tls_key);
HANDLE heap_2 = TlsGetValue(tls_key);
```

Przy odłączeniu wątku od procesu (case DLL_THREAD_DETACH) każdy wątek niszczy swój heap (rys. 3.12) .



Rys. 3.12 Przed zakończeniem pracy każdy wątek niszczy swój własny heap: HeapDestroy(TlsGetValue(tls_key));



Rys. 3.13 Przy zakończeniu pracy wątek 1 niszczy swoje resursy i slot TLS , oraz zwalnia element tablicy TLS o numerze tls_key – ustawia ten element na NULL: TlsFree(tls_key);

Przy odłączeniu DLL od procesu (case DLL_PROCESS_DETACH) pierwotny wątek zwalnia TLS slot o numerze tls_key (rys. 3.13) .

Wiele algorytmów dopuszczą usunięcie synchronizacji, jeśli użyć TLS zamiast danych wspólnych (shared data). To może istotnie zmniejszyć nakład i podnieść wydajność algorytmu. Przy użyciu TLS dane mogą pozostać w pamięci podręcznej procesorów znacznie dłuższej, niż w przypadku, kiedy są używane dane wspólne, jeśli procesory nie mają wspólnej pamięci podręcznej. Dane wspólne przy modyfikacji i zapisie jednym z procesorów powodują zgłoszenia o niepoprawności pamięci podręcznej innych procesorów, jeśli zmienne, zmodyfikowane różnymi procesorami, są mapowane na tą samą linię pamięci podręcznej – powstaje przeładowanie cache - linii. Przy użyciu TLS taka sytuacja jest wykluczona – dane TLS mogą być umieszczone w pamięci podręcznej tylko jednego procesora.

Nie dotyczy to Hyper-Threading'u (HT). Jest to taka technologia, w której dwa logiczne procesory rozdzielają pomiędzy sobą ten sam procesor fizyczny [Intel - HT, (2011)]. Oznacza to, że pamięć podręczna L1 tego samego procesora fizycznego powinna obsłużyć dwa wątki. Dla aplikacji o niskiej wydajności ($q \sim 1$), które powodują dużo pustych cykli procesora, technika HT może zmniejszyć ich ilość, ponieważ jeden fizyczny procesor liczy dwa takie zadania, a nie jedno. To może spowodować podniesienie wydajności do 30 % [Intel - HT, (2011)]. Dla algorytmów z wysoką wydajnością, które intensywnie stosują wielokrotne użycie pamięci podręcznej (cache reuse), HT może powodować spadek wydajności, ponieważ powstaje konkurowanie wątków za resursy pamięci podręcznej.

Przy HT częsty dostęp do adresów pamięci wirtualnej nie rozdzielonych wartością 64 KB, też może powodować istotny spadek wydajności. Heap wątku zwykłe nie jest wyrównany do granic 64 KB, ale jeżeli w tym samym cache znajdują się stosy dwóch wątków, przyczyna cache-konfliktów jest oczywista – stosy dwóch wątków mają wspólną stronę pamięci o rozmiarze 64 KB. Rozmiar strony pamięci (64 KB albo inna wartość) zależy od typu procesora.

Jednym ze sposobów, który daje możliwość naprawienia sytuacji, jest wprowadzenie stack offsets, które w sposób sztuczny rozdzielają przestrzeń adresową pomiędzy danymi każdego wątku. Szczegóły są podane w podręczniku użytkownika korporacji Intel [Intel – HT1, 2005].

3.4. Algorytm dot = $\mathbf{x}^T \cdot \mathbf{y}$.

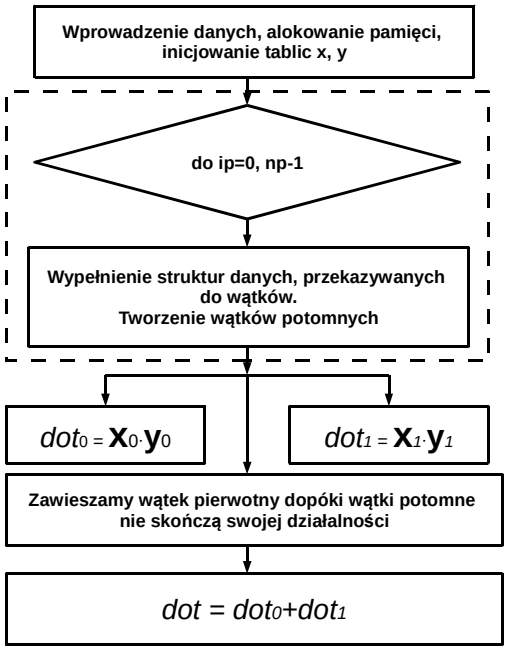
Na rys. 3.14 jest przedstawiony algorytm obliczenia iloczynu skalarnego dwóch wektorów w trybie dwuwątkowym. Ten algorytm można łatwo rozszerzyć na przypadek dowolnej ilości wątków.

Przy uruchomieniu zadania wątek pierwotny wprowadza dane (*N* – rozmiar zadania, *np* – ilość wątków, *ntimes* – ilość powtórzeń) , alokuje pamięć i wypełnia tablice **x**, **y** .

Dla przekazywania danych do wątków potomnych jest tworzona tablica struktur THREAD_DATA tDat[np], gdzie element tDat[0] jest przeznaczony dla wątku o numerze 0, tDat[1] – dla wątku o numerze 1 itd. Typ THREAD_DATA jest podany poniżej.

```
struct THREAD_DATA
{
      int loc_N;        //rozmiar podwektora przekazywanego
                        //do wątku potomnego
      int ntimes;       // ilość powtórzeń
      double *X;        //podwector X - tylko do odczytu
      double *Y;        //podwector Y - tylko do odczytu
      double res;       //wynik obliczeń odpowiednich
                        //podwektorów na każdym wątku
};
```

W pętli **do ip = 0, np-1** są wypełnione odpowiednie elementy tablicy struktur tDat oraz tworzone wątki potomne (na przykład, przez wywołanie funkcji *CreateThread*).



Rys. 3.14 Algorytm obliczenia iloczynu skalarnego dwóch wektorów w trybie wielowątkowym

Do każdego wątku potomnego będzie przekazywany element *tDat*[*ip*].

Po utworzeniu wątków potomnych wątek pierwotny trzeba zawiesić, inaczej sumowanie $dot = \sum_{ip=0}^{ip=np-1} dot_i$ może zacząć się wcześniej niż wątki potomne obliczą wyniki dot_i, $i = 0, 1, \ldots, np-1$. Zawieszenie wątku pierwotnego odbywa się poprzez wywołanie funkcji *WaitForMultipleObjects* :

```
DWORD WINAPI WaitForMultipleObjects(
  __in  DWORD nCount,              //number of child threads
  __in  const HANDLE *lpHandles,  //array of handles for child threads
  __in  BOOL bWaitAll,             //flag
  __in  DWORD dwMilliseconds       //The time-out interval
);
```

Przy swoim zakończeniu każdy wątek potomny podaje sygnał systemowi operacyjnemu. Funkcja *WaitForMultipleObjects* przechwytuje te sygnały, a jeżeli flaga *bWaitAll* = TRUE , to wątek pierwotny obudzi się, jak tylko wszystkie wątki potomne skończą swoją prace. Parametr *dwMilliseconds* = INFINITE oznacza, że wątek pierwotny będzie czekał na zakończenie wątków potomnych nieokreślony odcinek czasu, czyli nie obudzi się dopóki *WaitForMultipleObjects* nie dostanie sygnałów od wszystkich wątków potomnych. Tylko po obudzeniu wątek pierwotny dostaje wyniki obliczeń wątków potomnych i wykonuje sumowanie.

Tablice **x**, **y** w obszarze każdego wątku potomnego są tylko do odczytu, dla tego można nie martwić się o konflikty w pamięci podręcznej. Przyjmujemy zgrubny podział danych pomiędzy wątkami (rys. 3.4). Wskaźnik do początku części tablicy wektora, przekazywanego do wątku *ip*, można policzyć jak

tDat[ip].X = &X[ip*loc_N], tDat[ip].Y = &Y[ip*loc_N],

gdzie *loc_N* = *N*/*np* .

Każdy wątek potomny umieszcza swój wynik do osobnego elementu tablicy *tDat*[*ip*].*res* . To daje możliwość przekazać wynik wątkowi pierwotnemu po zakończeniu wątków potomnych. Oprócz tego, każdy wątek umieszcza swój wynik do osobnej zmiennej, dla tego synchronizacja przy zapisie w zmienną *tDat*[*ip*].*res* nie jest potrzebna.

Zmienne *rrr* każdego wątku definiujemy, jako zmienne lokalne, a po obliczeniu ich wartości przypisujemy do *tDat*[*ip*].*res*:

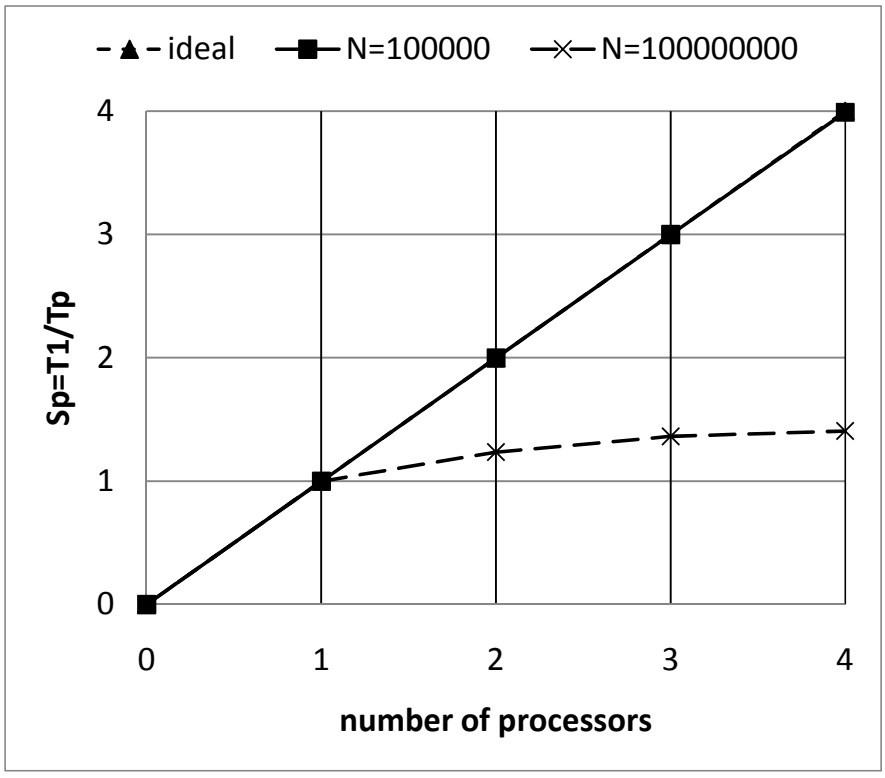
```
double register rrr;
for(it=0; it<ptrDat->ntimes; it++)  //powtarzamy mnożenie
                                    //ntimes razy
{
    rrr = 0.0;
```

```
    for(i=0; i<loc_N; i++)
    {
          rrr += X_loc[i]*Y_loc[i];
    }
    ptrDat->res = rrr;
}
```

Aby mierzone odcinki czasu nie były zbyt krótkie, powtarzamy obliczenia dla każdego wątku *ntimes* razy, ponieważ dla krótkich odcinków czasu funkcje pomiarowe podają niedokładne wartości. Wskaźnik *ptrDat* wskazuje na element *tDat*[*ip*] .

Ponieważ *rrr* jest zmienną lokalną, umieszczoną w stosie wątku, jej modyfikacja odbywa się bezpiecznie z punktu widzenia konfliktów w pamięci podręcznej procesorów.

Rozważmy wyniki dla dwóch zadań (rys. 3.15), wykonanych na komputerze z procesorem Intel® Core™2 Quad.



Rys. 3.15 Algorytm *dot* = $\mathbf{x}^T\mathbf{y}$. Komputer z procesorem Intel® Core™2 Quad

Pierwsze zadanie ma rozmiar N = 100 000 przy *ntimes* = 100 000, a drugie – N = 100 000 000 przy *ntimes* = 100. Liczba przetwarzanych danych jest dokładnie taka sama. W pierwszym zadaniu tablice wektorów $\mathbf{x}^T$, $\mathbf{y}$ są rozmieszczane w pamięci podręcznej procesorów, a w drugim – nie. Dla tego speed up dla pierwszego zadania jest praktycznie idealny, a dla drugiego – bardzo słaby.

Dalej nas będą interesować wyniki dla dużych tablic danych, które nie są umieszczane w pamięci podręcznej. Obliczenia zostały wykonane na komputerach:

- Komputer z cztery-rdzeniowym procesorem Intel® Core™2 Quad CPU Q6600 @2.40 GHz, cache L1: 4x32 KB, L2: 4096 KB, RAM DDR2 333.7 MHz, 8 GB, chipset Intel P35/G33/G31, OS Windows Vista™ Business (64-bit), Service Pack 2.

- Komputer z cztery-rdzeniowym procesorem AMD Phenom™ II x4 995 3.2 GHz; L1: 4x64 KB, L2: 4x512 KB, L3: 6 MB, RAM DDR3 1066 MT/s, 16 GB, chipset AMD 790X, OS Windows Vista™ Business (64-bit), Service Pack 2.

- Stacja robocza DELL z dwoma procesorami Intel Xeon X5660 @ 2.8 GHz /3.2 GHz (2×6 = 12 rdzeni), RAM DDR3, 24 GB, OS Windows 7 (64-bit).

W tab. 3.1 są podane wyniki rozwiązania zadania o rozmiarze N = 100 000 000, *ntimes* = 50, aplikacja 32-bitowa (platforma ia32) dla komputerów z procesorami Intel® Core™2 Quad i AMD Phenom™ II x4 995.

Tabela 3.1

Porównywanie wyników algorytmu *dot* = $\mathbf{x}^T\mathbf{y}$ na różnych komputerach.

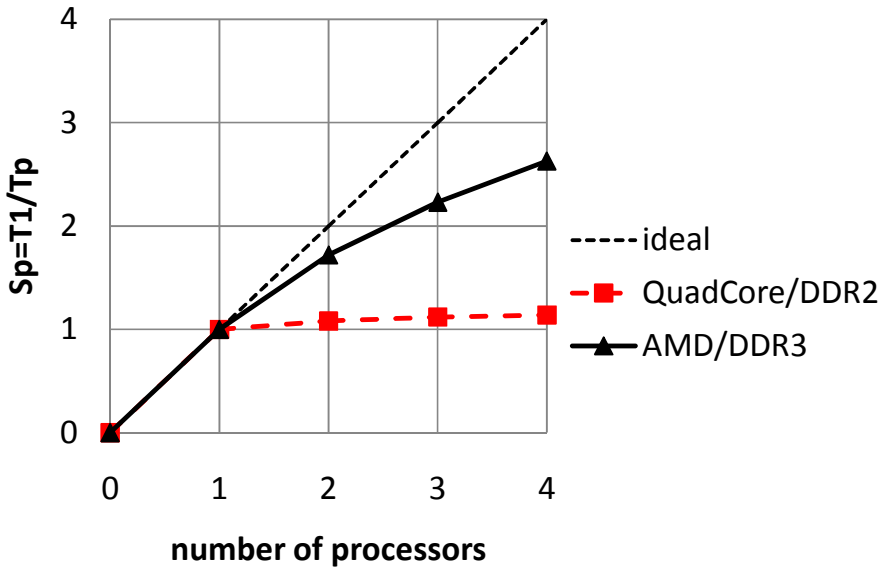
N = 100 000 000, *ntimes* = 50

Liczba procesorów	Core™2 Quad (2.4 GHz, DDR2)		AMD Phenom IIx4 995 (3.2 GHz, DDR3)	
	Czas obliczeń, ms	$S_p=T_1/T_p$	Czas obliczeń, ms	$S_p=T_1/T_p$
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

Wykresy speed up są przedstawione na rys. 3.16. Na jednym wątku komputer z procesorem Intel® Core™2 Quad wykazał wynik, lepszy od komputera z procesorem AMD Phenom™ II x4 995, chociaż ten ostatni ma większą częstotliwość taktowania. Na dwóch wątkach już wyraźnie widać przewagę

komputera z procesorem AMD Phenom™ II x4 995, a na czterech – czas obliczeń na drugim komputerze jest dwa razy mniejszy od czasu obliczeń na pierwszym.

Wykresy speed up (rys. 3.16) demonstrują, że komputer z procesorem AMD Phenom™ II x4 995 z pamięcią DDR3 ma znacznie lepsze przyspieszenie od komputera z procesorem Intel® Core™2 Quad z pamięcią DDR2.



Rys. 3.16 Algorytm *dot* = $\mathbf{x}^T\mathbf{y}$. Porównywanie speed up na różnych komputerach

W tab. 3.2 są podane wyniki obliczeń zadania o rozmiarze N = 100 000 000, *ntimes* = 40 na stacji roboczej DELL, a na rys. 3.17 – wykresy przyspieszenia przy zwiększeniu ilości procesorów.

Krzywa "ideal" odpowiada idealnemu przyspieszeniu, które nie może być zrealizowane na procesorach, wspierających tryb *turbo-boost* [Intel TB, 2011].

Krzywa "id-tb-fun" jest aproksymacją idealnego przyspieszenia w trybie *turbo-boost*, jest to parabola kwadratowa, 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 procesorów zmniejsza się do 2.8 GHz, a przy obciążeniu tylko jednego rdzenia częstotliwość wynosi 3.2 GHz).

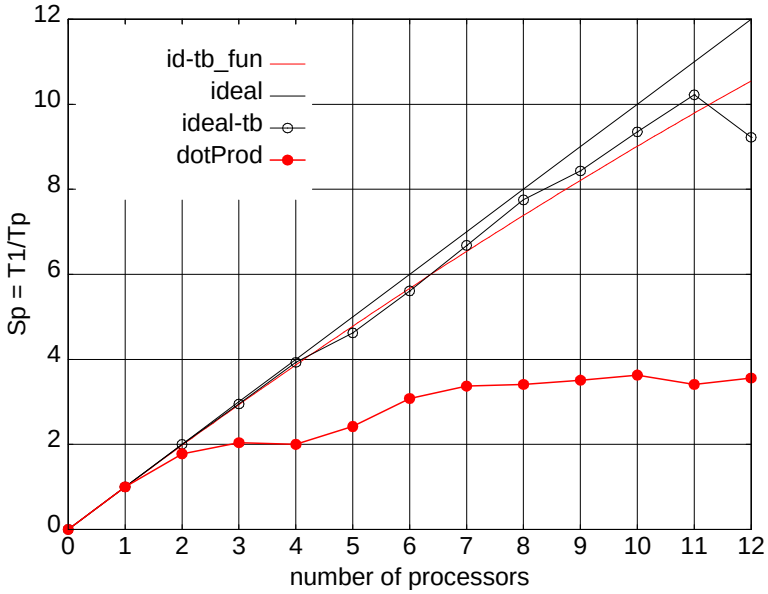
Krzywa "ideal-tb" jest to wynik testowania programem, który intensywnie liczy funkcje trygonometryczne przy tym praktycznie nie obciążając układu pamięci. Można uznać, że speed-up dla takiego algorytmu będzie bardzo zbliżony do przyspieszenia idealnego.

Krzywe "id-tb_fun" i "ideal-tb" są bliskie, stąd wynika, że mogą one przedstawiać przyspieszenie idealne dla podanego sprzętu.

Tabela 3.2

Przyspieszenie obliczeń przy zwiększeniu ilości wątków dla zadania *dot* = $\mathbf{x}^T\mathbf{y}$ na stacji roboczej DELL, dwa procesory Intel Xeon X5660 @ 2.8 GHz /3.2 GHz (2×6 = 12 rdzeni), RAM DDR3, 24 GB. *N* = 100 000 000, *ntimes* = 50

Ilość procesorów	Czas obliczeń, s	$S_p=T_1/T_p$
1	8.611	1
2	4.851	1.78
3	4.228	2.04
4	4.305	2.00
5	3.557	2.42
6	2.792	3.08
7	2.558	3.37
8	2.527	3.41
9	2.450	3.51
10	2.371	3.63
11	2.527	3.41
12	2.418	3.56



Rys. 3.17 Algorytm *dot* = $\mathbf{x}^T\mathbf{y}$. Speed-up, stacja robocza DELL

Nawet na 12-rdzeniowej stacji roboczej DELL podany algorytm wykazuje bardzo ograniczone przyspieszenie. Współczynnik wydajności $q = f/m$ (f – liczba operacji arytmetycznych, m – liczba transferów danych) wynosi 1. To oznacza, że na każdy transfer danych wypada jedna operacja arytmetyczna. Załóżmy, że trwałość jednej operacji arytmetycznej wynosi 1 ns, a trwałość transferu jednego słowa double – 100 ns. Stąd wynika, że 99 ns procesor będzie czekał na następne przesłanie danych.

Teraz zwiększymy liczbę procesorów do 2. W przeciągu 1 ns każdy procesor będzie wykonywał 1 operację arytmetyczną – razem 2 operacje arytmetyczne. Magistrala musi obsłużyć 2 procesory i przesłać 2 kolejne słowa double. Jeśli magistrala nie ma zapasu przepustowości, to będzie potrzebowała 200 ns. Czas oczekiwania każdego procesora będzie zwiększony praktycznie dwukrotnie, ponieważ liczba operacji arytmetycznych, wykonywanych każdym procesorem, zmniejszyła się też dwukrotnie, więc ogólny czas wykonania zadania pozostaje taki sam, jak w przypadku obliczeń sekwencyjnych.

Z tego wynika, że algorytmy o niskiej wydajności na komputerach wielordzeniowych klasy PC, które mają odnośnie nie wielką przepustowość układu pamięci, nie nadają się do istotnego przyspieszenia przy zwiększeniu ilości wątków, ponieważ ta sama magistrala nie jest w stanie jednocześnie obsłużyć kilka procesorów.

3.5. Wyniki testów typowych algorytmów na komputerach wielordzeniowych PC.

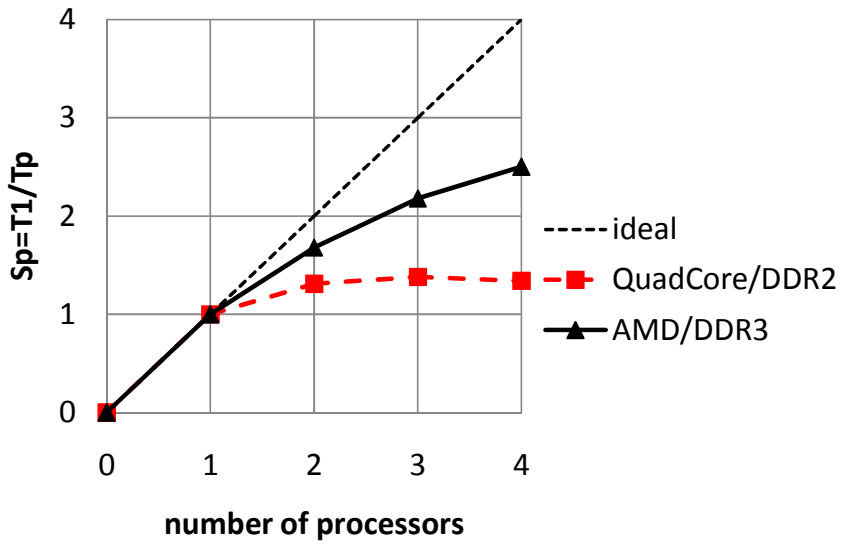
Podobna sytuacja wynika także dla algorytmu mnożenia macierzy przez wektor $\mathbf{y} = \mathbf{A} \cdot \mathbf{x}$. Dla zadania o rozmiarze $N = 10\,000$ w pamięci podręcznej L2 mieści się wiersz macierzy oraz wektor $\mathbf{x}$. Dla tego wskaźnik wydajności $q = 2$ (rozdział 2.1.3). Wyniki tego testu na komputerach z procesorami Intel® Core™2 Quad oraz AMD Phenom™ II x4 995 są podane w tab. 3.3 i na rys. 3.18.

Chociaż speed up dla tego zadania jest trochę lepszy od poprzedniego, nie zmienia to wniosku, że dla algorytmów o niskiej wydajności na wielordzeniowych komputerach klasy PC nie da się osiągnąć istotnego przyspieszenia przy zwiększeniu ilości wątków.

Tabela 3.3

Porównywanie wyników algorytmu $\mathbf{y} = \mathbf{A}\cdot\mathbf{x}$ na różnych komputerach. N = 10 000, *ntimes* = 50

Liczba procesorów	Core™2 Quad (2.4 GHz, DDR2)		AMD Phenom IIx4 995 (3.2 GHz, DDR3)	
	Czas obliczeń, ms	$S_p=T_1/T_p$	Czas obliczeń, ms	$S_p=T_1/T_p$
1	9 469	1	8 424	1
2	7 250	1.31	5 023	1.68
3	6 884	1.38	3 868	2.18
4	7 078	1.34	3 370	2.50



Rys. 3.18 Algorytm $\mathbf{y} = \mathbf{A}\cdot\mathbf{x}$. Porównywanie speed up na różnych komputerach

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

Tak, są to algorytmy, dla których magistrala nie jest obciążona.

Takim algorytmem jest obliczenie numeryczne całki metodą trapezów, jeśli funkcja podcałkowa jest przedstawiona analitycznie. Zadanie to nie zawiera dużo transferów danych RAM – cache – RAM dla tego wykazuje bardzo dobry speed up.

Inny algorytm, który pracuje z dużymi tablicami danych – to algorytm mnożenia macierzy przez macierz metodą blokową. Oszacowanie wskaźnika wydajności wynosi $q = \sqrt{M/3}$, gdzie M – rozmiar pamięci podręcznej (rozdział 2.1.4). Na każdy transfer danych taki algorytm wykonuje $\sqrt{M/3}$ operacji arytmetycznych. Dla dość dużego rozmiaru cache M liczba pustych cykli procesora będzie minimalna, a magistrala zostanie odciążona. Powstaje to w skutek wielokrotnego użycia pamięci podręcznej (cache reuse) .

W tab. 3.4 są podane wyniki zadania **C** = **A**·**B** dla jego różnych realizacji na komputerze z procesorem Intel Core™2 Quad z pamięcią DDR2, który dla zadań poprzednich wykazywał najgorszy speed-up.

Tabela 3.4

Wydajność (MFLOPS) różnych algorytmów zadania **C** = **A**·**B** , N = 4 000, przy zwiększeniu liczby procesorów *np* (platforma ×64, komputer z procesorem Intel® Core™2 Quad).

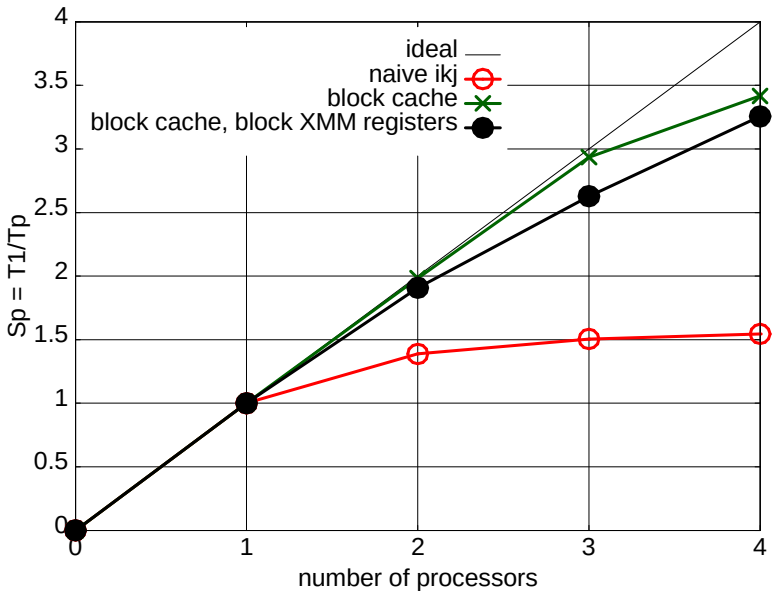
Algorytm	*np*=1	*np*=2	*np*=3	*np*=4
Naiwny (*ikj*)	1 104	1 532	1 662	1 706
Metoda blokowa (cache blocking)	1 925	3 818	5 649	6 577
Metoda blokowa w technice Intel MKL (cache blocking, XMM register's blocking)	7 239	13 813	19 026	23 567
DGEMM Intel MKL	8 984	17 792	17 811	29 293

Każdy z podanych algorytmów jest przedstawiony w rozdz. 2.1.4, a ich speed up – na rys. 3.19 . Ostatnia metoda była pobrana z biblioteki Intel MKL.

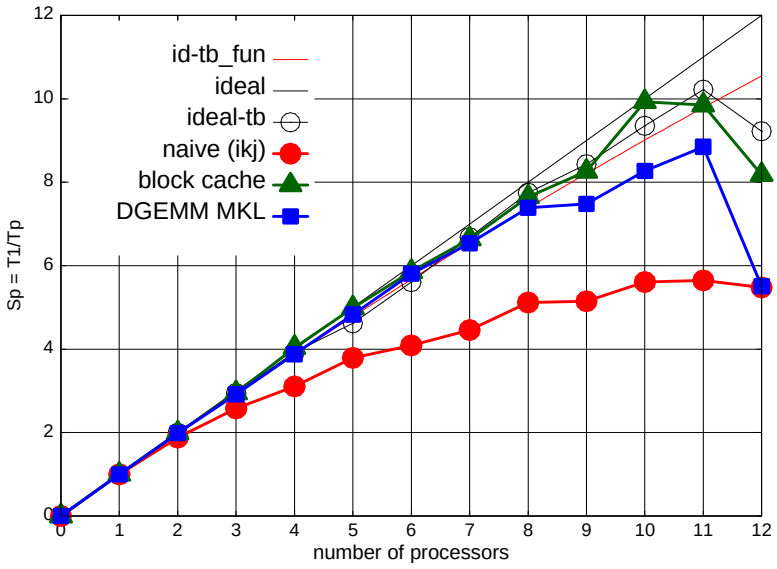
Algorytm naiwny (*ikj*) dla rozmiaru zadania N = 4 000 ma wskaźnik wydajności 2, ponieważ dwa wiersze macierzy są umieszczone w pamięci podręcznej L2. Jest to algorytm o niskiej wydajności. Dla tego speed up jest dla niego ograniczony, na podanym sprzęcie nie przekracza on wartości 1.6 .

Algorytm blokowy oraz algorytm wykonany w technice Intel MKL (rozdział 2.1.4) intensywnie wykorzystują pamięć podręczną i w skutek tego odciążają magistralę. Dla tego otrzymaliśmy na 4 wątkach przyspieszenie rzędu 3.2 – 3.4 .

Na rys. 3.20 są podane wyniki porównywania speed up algorytmu naiwnego, algorytmu blokującego tylko cache oraz algorytmu DGEMM z Intel MKL, otrzymane na stacji roboczej DELL. Krzywe "id-tb_fun" i "ideal-tb" są oszacowaniami przyspieszenia idealnego dla podanego komputera, procesory którego wspierają tryb *turbo-boost*. Tu dla algorytmu naiwnego (ikj) udało się uzyskać maksymalne przyspieszenie S_p = 5.6 , a dla algorytmu blokowego i DGEMM – odpowiednio S_p = 10 i S_p = 8.8.

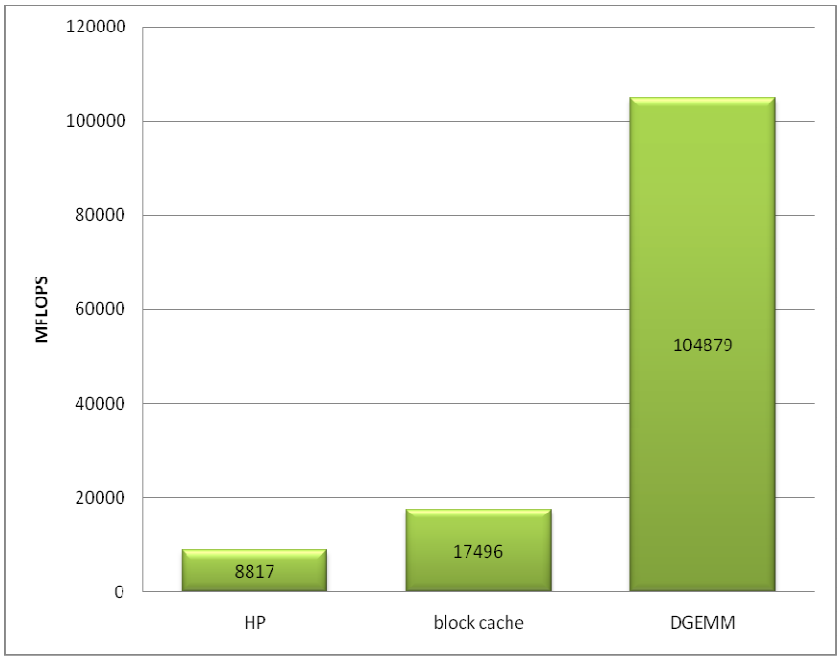


Rys. 3.19 Zadanie **C** = **A**·**B** . Porównywanie speed up dla różnych algorytmów na komputerze z procesorem Intel® Core™2 Quad



Rys. 3.20 Zadanie **C** = **A**·**B**. Porównywanie speed up dla różnych algorytmów na stacji roboczej DELL (12 rdzeni)

Na rys. 3.21 został podany diagram największej wydajności, osiągniętej dla algorytmów naiwnego (ikj), blokowego i DGEMM.



Rys. 3.21 Zadanie **C** = **A**·**B**. Porównywanie maksymalnej wydajności dla różnych algorytmów na stacji roboczej DELL (12 rdzeni)

Jeśli dla algorytmu naiwnego (ijk) maksymalna wydajność wynosi 8.8 GFLOPS, to dla metody DGEMM z Intel MKL – 104.8 GFLPOS. Użycie zaawansowanych technik daje możliwość podniesienia wydajności o 11.9 razy na podanym komputerze.

Przy konstruowaniu algorytmów złożonych wszystko, co nie zostanie doprowadzone do algorytmu mnożenia macierzy przez macierz albo algorytmów, podobnych, co do zakresu wydajności, będzie miało słabe przyspieszenie. Ten wniosek dotyczy algorytmów algebry liniowej dla macierzy gęstych, zrealizowanych dla komputerów wielordzeniowych klasy PC oraz pracujących z dużymi tablicami danych.

3.6. Obliczenie dopełnienia Schura przy zrównolegleniu na podstawie OpenMP. Mikrojądro dla pracy z rejestrami XMM, pakowanie danych.

Rozważmy rozwiązywanie układów równań liniowych algebraicznych z macierzą symetryczną metodą blokową Choleskiego, podaną w rozdziale 2.2.2. Podstawową procedurą w tej metodzie jest obliczenie dopełnienia Schura (2.26). Właśnie na tym etapie w pierwszej kolejności trzeba stosować techniki zaawansowane.

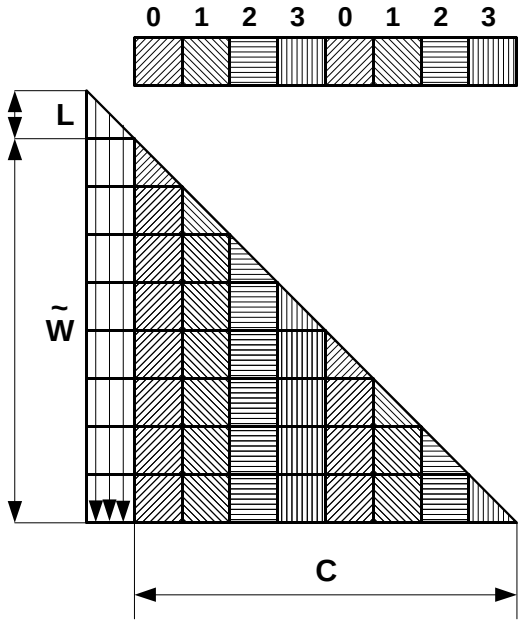
Rozwiązanie takiego zadania procedurami DGEMM, DSYRK, przedstawionymi bibliotekami wysokiej wydajności (na przykład, Intel MKL), wymaga umieszczenia w pamięci głównej całej macierzy, nawet jeśli jest to macierz symetryczna. Przy tym w pamięci głównej trzeba przechowywać N^2 liczbę elementów. Dla macierzy symetrycznej wystarczy przechowywać tylko dolny albo tylko górny trójkąt. Wynosi to $N\cdot(N+1)/2 \approx N^2/2$ elementów. W wielu przypadkach przy rozwiązywaniu zadań technicznych zaoszczędzenie pamięci głównej jest zasadniczym faktorem, ponieważ od niego zależy, czy określone zadanie zostanie rozwiązane na dostępnym komputerze. Typowym przykładem jest rozwiązywanie układów równań liniowych algebraicznych MES (Metoda Elementów Skończonych) metodą wielo-frontalną [Fialko, 2009, CT], [Fialko, 2009, M]. Metoda wielo-frontalna doprowadza sfaktoryzowanie macierzy rzadkiej do sekwencji sfaktoryzowania macierzy gęstych – tak zwanych macierzy frontalnych (rozdziały 5.2, 5.6, 5.7). Dla tego żeby zadanie było rozwiązane, koniecznie jest umieszczenie największej macierzy frontalnej w pamięci RAM. Dla dużych zadań (1 000 000 – 5 000 000 równań) rozmiar macierzy frontalnej wynosi 10 000 – 16 000. Dla ogólnego schematu przechowywania potrzebne są 760 – 1 950 MB pamięci RAM, a dla symetrycznego schematu – 380 – 975 MB, co daje możliwość rozwiązywania takich zadań nawet na platformie 32 bitowej. Właściwie dla tego powstaje konieczność tworzenia algorytmu blokowej metody Choleskiego przy symetrycznym schemacie przechowywania macierzy.

Dolna trójkątna część symetrycznej macierzy **A** jest umieszczona kolumna po kolumnie w pamięci RAM (rys. 3.22).

Macierz została podzielona na bloki. Kolumny blokowe są mapowane na różne wątki, gdzie liczby 0, 1, ... , 3 oznaczają odpowiednie numery wątków. Jeden krok faktoryzacji blokowej składa się z następujących etapów (rozdział 2.2.2) .

W pierwszym etapie sfaktoryzujemy blok diagonalny (2.24) – rozdział 2.2.2. W drugim etapie obliczamy bloki pod diagonalą (2.25). Algorytm obliczenia dopełnienia Schura (2.26) – etap trzeci – jest przedstawiony na rys. 3.22, a diagram mnożenia bloków oraz pakowanie danych – na rys. 3.23. Zrównoleglenie zostało wykonane na podstawie OpenMP, przy czym zrównoleglona jest pętla zewnętrzna.

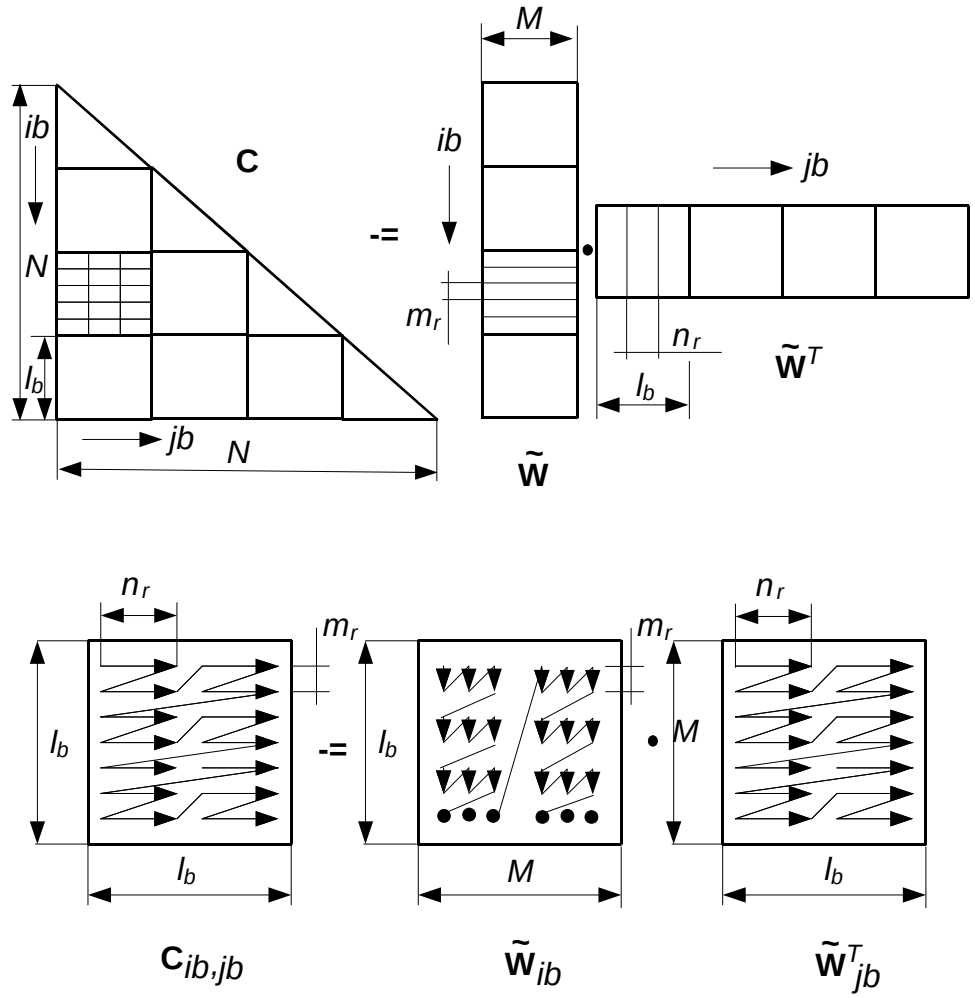
Dla podanego zadania nie jest oczywiste, jak podzielić dane pomiędzy wątkami, ponieważ każdy wątek przetwarza różne objętości danych (rys. 3.22). Okazało się że klauzula *sheduler (dynamic)* zapewnia dobre zrównoważenie obciążenia pomiędzy wątkami.



Rys. 3.22 Symetryczny sposób przechowywania macierzy **A** przy umieszczeniu jej elementów kolumna po kolumnie.

```
// Pack W̃
# pragma omp parallel for private(ib, jb) scheduler(dynamic)
for( jb = 0; jb < Nb; jb++)
{
   // Pack  W̃^T_jb
   for(ib = jb; ib < Nb; ib++)
   {
      C̃_{ib, jb} = C̃_{ib, jb} − W̃_ib · W̃^T_jb
   }
}
```

Rys. 3.23 Algorytm obliczenia dopełnienia Schura



Rys. 3.24 Diagram mnożenia bloków oraz pakowanie danych

W pętli wewnętrznej mnożymy bloki macierzy $\tilde{\mathbf{W}}_{ib}$, $\tilde{\mathbf{W}}^T_{jb}$ i otrzymujemy blok $\mathbf{C}_{ib,\,jb}$. Dane bloków $\tilde{\mathbf{W}}_{ib}$, $\tilde{\mathbf{W}}^T_{jb}$ są tylko do odczytu, a dane bloku $\mathbf{C}_{ib,\,jb}$ – tak do odczytu jak i do zapisu. Każdy wątek dokonuje zapisu do swojego osobnego bloku $\mathbf{C}_{ib,\,jb}$. Dla tego synchronizacja typu sekcji krytycznych nie jest potrzebna.

Technika mnożenia bloków jest przedstawiona na rys. 3.24. W celu wektoryzowania obliczeń używamy rejestrów XMM. Podział bloków na klatki

$m_r \times n_r$ przy blokowaniu XMM rejestrów jest oznaczony cienkimi liniami. Na platformie 32-bitowej $m_r = 2$, $n_r = 4$, a na platformie 64-bitowej – $m_r = 4$, $n_r = 4$. W dolnej części rys. 3.24 jest podane pakowanie danych wewnątrz bloków z celem zmniejszenia chybień w pamięci podręcznej przy odczycie.

Przepakowujemy całą kolumnę blokową $\tilde{\mathbf{W}}$ przed uruchomieniem pętli *jb* (rys. 3.23), ponieważ blok $\tilde{\mathbf{W}}_{ib}$ zależy od indeksu iteracji pętli wewnętrznej. Blok $\tilde{\mathbf{W}}_{jb}^{T}$ nie zależy od indeksu *ib* , dla tego przepakowujemy go przed uruchomieniem pętli wewnętrznej. Przyjęty schemat przepakowania jest optymalny, ponieważ przy wykonaniu całego zadania liczba transferów danych dla każdej tablicy nie przekroczy liczby elementów danej tablicy (rozdział 2.1.4.6).

Rozmiar bloku l_b jest wyznaczony pod warunkiem, że trzy bloki zmieszczą się w pamięci podręcznej L1. Bezpośrednie zastosowanie procedury DGEMM dla obliczenia wyrażenia w pętli wewnętrznej nie jest skuteczne, ponieważ dla przyjętego schematu przechowywania macierzy symetrycznej przed każdym mnożeniem trzeba wykonywać przepakowanie danych w obrębie każdego bloku. Dla schematu przechowywania ogólnego przepakowanie danych nie było by potrzebne – wystarczyłoby tylko podać adres początku bloku oraz "leading dimension" – rozmiar kolumny całej macierzy.

Dla tego własne mikrojądro było napisane przy użyciu techniki SSE2 – kod niskiego poziomu, który bezpośrednio realizuje operacje nad rejestrami XMM. Ten kod jest podobny do przedstawionego w rozdziale 2.1.4.5.

W tab. 3.5 została podana wydajność proponowanej metody dla 1, 2, 3 i 4 wątków. Wyniki są otrzymane na komputerze z procesorem Intel® Core™2 Quad.

Tabela 3.5

Wydajność (MFLOPS) blokowej metody Choleskiego dla zadania o rozmiarze $N = 4\,000$ w zależności od liczby procesorów (komputer z procesorem Intel® Core™2 Quad). Na poziomie mnożenia bloków są używane techniki zaawansowane.

Platforma	ia32				Intel 64 (×64)			
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

W tab. 3.6 jest podana wydajność blokowej metody Choleskiego, gdzie na poziomie mnożenia bloków jest stosowany algorytm naiwny.

Przyspieszenie przy zwiększeniu ilości procesorów w obuch przypadkach jest dobre, jednak użycie technik zaawansowanych na poziomie mnożenia bloków istotnie podnosi wydajność.

Na platformie 64-bitowej stosuje się blokowanie rejestrów 4 × 4, a na platformie 32-bitowej – 2 × 4. Dla tego wydajność metody z użyciem technik zaawansowanych na platformie 64 bitowej jest wyższa. Przy metodzie, używającej w czasie mnożenia bloków algorytmu naiwnego, wydajność na platformie 64-bitowej mało się różni od wydajności na platformie 32-bitowej.

Tabela 3.6

Wydajność (MFLOPS) blokowej metody Choleskiego dla zadania o rozmiarze $N = 4\,000$ w zależności od liczby procesorów (komputer z procesorem Intel® Core™2 Quad). Na poziomie mnożenia bloków jest używany algorytm naiwny.

Platforma	ia32				Intel 64 (×64)			
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
$Sp=T_1/T_p$	1	1.97	2.87	3.32	1	1.97	2.90	3.47

3.7. Jeśli Państwa komputer ma kilka rdzeni ... .

Często od producentów oprogramowania można usłyszeć: "Jeśli Państwa komputer ma dwa procesory, to nasz program będzie liczył dwa razy szybciej…".

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

Dla odpowiedzi na to pytanie rozważmy przykład obliczenia iloczynu skalarnego dwóch wektorów w trybie wielowątkowym (rozdział 3.4) przy rozmiarze zadania $N = 10\,000\,000$, *ntimes* = 100. Tablicy **x**, **y** nie mogą być umieszczone w pamięci podręcznej, dlatego przy wykonaniu zadania będą ciągłe pobierane z pamięci RAM.

Dla zrównoleglenia zastosujemy Intel Threading Building Blocks [TBB]. W pierwszej realizacji użyjemy dla wektorów typ danych *concurrent_vector*, który został wprowadzony przez Intel i przeznaczony dla obliczeń wielowątkowych. Kolejno w drugiej zastosujemy dla tablic **x**, **y** typ *double*. Kod programu jest przedstawiony w załączniku 4. Program był tworzony w IDE Visual Studio 2008 z kompilatorem Intel C++ 11.1.046 ze wsparciem TBB. Opcji kompilatora są /c /O2 /Qipo /EHsc /MD /GS /fp:fast. Obliczenia były wykonane na komputerze z procesorem Intel® Core™2 Quad w wersji *release* na platformie 32-bitowej.

W tabeli 3.7 są podane czasy wykonania zadania dla typów danych *concurrent_vector* i *double*, a na rys. 3.25 jest przedstawiony speed up.

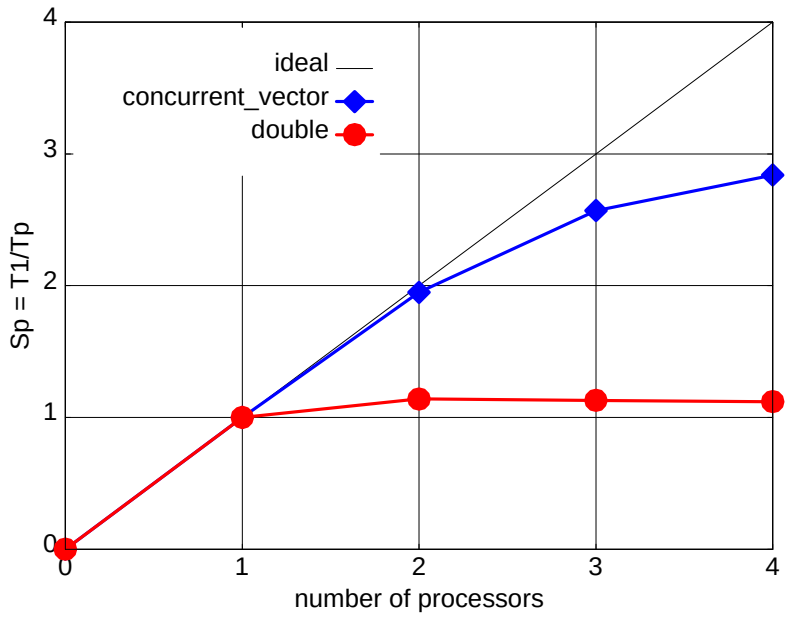
Dla typu danych *concurrent_vector* przyspieszenie wygląda dość dobrze. Dla typu *double* speed up jest bardzo słaby. Na pierwszy rzut oka zastosowanie typu danych *concurrent_vector* rozwiązało problem z niską przepustowością układu

pamięci. Jednak analizując wyniki, jakie są podane w tabeli 3.6, zauważmy, że czas obliczeń na dwóch wątkach dla typu *double* jest mniejszy od czasu obliczeń na czterech wątkach dla typu *concurrent_vector*. Oznacza to, że dostęp do danych typu *concurrent_vector* jest znacznie wolniejszy od dostępu do danych typu *double*. Przyspieszenie jest lepsze, ponieważ dla danych typu *double* czas obliczeń jest mniejszy.

Tabela 3.7

Czasy wykonania zadania (ms) dla różnych typów danych. *N* = 10 000 000, *ntimes* = 100.

Liczba procesorów	*concurrent_vector*	*double*
1	8 112	3 120
2	4 150	2 730
3	3 151	2 762
4	2 855	2 854



Rys. 3.25 Zadanie dot = $\mathbf{x}^T \cdot \mathbf{y}$. Zrównoleglenie na podstawie TBB. Porównywanie speed up dla różnych typów danych. Komputer z procesorem Intel® Core™2 Quad

Z tego wynika: speed up nie może być przyjęty, jako jedyna miara jakości systemu obliczeniowego równoległego. Konieczne jest oszacowanie czasu obliczeń albo wydajności razem ze speed up.

4. Macierzy rzadkie symetryczne.

4.1 Podstawowe pojęcia. Graf przyległości. Zbiór przyległy. Struktura poziomów z korzeniem w wierzchołku peryferyjnym. Formaty skompresowane.

Istnieje wiele technicznych i naukowych problemów, w których zastosowanie formalizacji matematycznej prowadzi do działań nad macierzami rzadkimi symetrycznymi. To są zadania mechaniki, hydromechaniki, elektromagnetyzmu, fizyki jadra atomowego, modelowanie różnych systemów ekologicznych itd. Zwykłe w takich zadaniach powstają macierze rzadkie o dużym rozmiarze – setki tysięcy i miliony równań.

Rzadkimi nazywamy macierze, zawierające wiele zerowych elementów. Działania nad takimi macierzami wymagają opracowania specjalnych struktur danych, które dają możliwość obejść operacje nad elementami zerowymi i w taki sposób istotnie podnieść wydajność obliczeń i zmniejszyć zapotrzebowanie na pamięć główną.

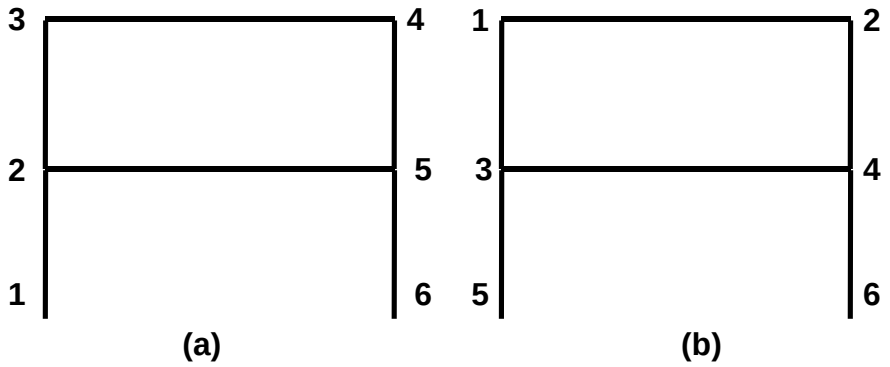
Podstawowymi działaniami nad macierzami rzadkimi są:

- Mnożenie macierzy przez wektor.
- Faktoryzacja macierzy.
- Podstawienia bezpośrednie i wsteczne.

Na przykładach z dziedziny mechaniki konstrukcji – ramach płaskich – rozważmy, jak powstają macierze rzadkie. Będziemy wyróżniać z całego modelu MES (metody elementów skończonych) węzły i elementy skończone. Węzły – są to nieskończone małe twarde ciała, w których są łączone pręty ramy płaskiej – elementy skończone.

Przy tworzeniu modelu obliczeniowego preprocesor programu MES automatycznie numeruje węzły. Użytkownik może w różnej kolejności tworzyć elementy tej konstrukcji. Dla tego ten sam model obliczeniowy może mieć różne układy numerowania węzłów. Na rys. 4.1 są podane dwa różne sposoby takiego numerowania.

Macierz sztywności MES jest macierzą rzadką, struktura, której zależy od topologii modelu obliczeniowego (od tego, jakie węzły są łączone elementami skończonymi, a jakie – nie) oraz od sposobu numerowania węzłów. Każdemu niepodpartemu węzłowi modelu obliczeniowego ramy płaskiej w macierzy rzadkiej odpowiada kolumna blokowa (wiersz blokowy) o szerokości 3, ponieważ każdy węzeł ramy płaskiej zawiera trzy stopnie swobody. Nadanie podpór zmniejsza ilość stopni swobody w węźle, ale podstawowo nie wpływa na istotę zawartości, prezentowanych w danym rozdziale. Dlatego dalej będziemy rozważali układy niepodparte.



Rys. 4.1 Różne sposoby numerowania węzłów ramy płaskiej

Na rys. 4.2 są przedstawione umieszczania elementów niezerowych – portrety tych macierzy albo niezerowe struktury rzadkości.

$$\begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} 1 & x & & & & \\ x & 2 & x & & x & \\ & x & 3 & x & & \\ & & x & 4 & x & \\ & x & & x & 5 & x \\ & & & & x & 6 \end{pmatrix} \end{array} \qquad \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} 1 & x & x & & & \\ x & 2 & & x & & \\ x & & 3 & x & x & \\ & x & x & 4 & & x \\ & & x & & 5 & \\ & & & x & & 6 \end{pmatrix} \end{array}$$

(a) (b)

Rys. 4.2 Niezerowe struktury rzadkości dla ramy płaskiej z numerowaniem węzłów, oznaczonym na rys. 4.2 (a, b)

Dla takich układów równań na głównej diagonali znajdują się elementy niezerowe, oznaczone jako 1, 2, ... , a niezerowe elementy poza diagonalne są oznaczone jako "*x*". Każdemu niezerowemu elementowi w danym zadaniu odpowiada gęsta podmacierz o rozmiarze 3×3. Dalej będziemy zakładać, że w każdym węźle modelu obliczeniowego znajduje się tylko jedno równanie. Uprości to zrozumienie materiału bez obniżenia ogólności wniosków.

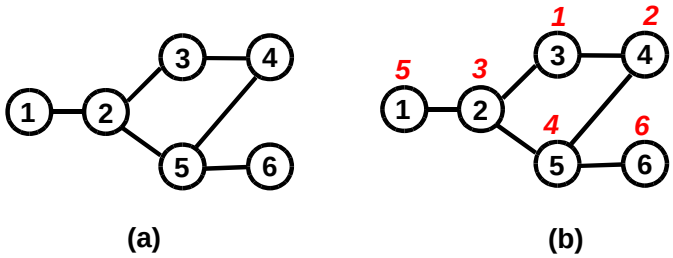
Na rys. 4.2 są podane numerowania kolumn macierzy oraz elementów diagonalnych. Struktura macierzy rzadkiej odzwierciedla topologie modelu obliczeniowego. Dla układu numerowania (a) (rys. 4.1, a) węzeł 1 jest łączony z węzłem 2, lecz nie jest łączony ani z węzłem 3, ani z węzłem 4, itd. W macierzy rzadkiej (rys. 4.2, a) w pierwszym równaniu mamy tylko niezerowy poza

diagonalny element a_{12} . Węzeł 2 ma za sąsiadów węzły 1, 3, 5. W macierzy rzadkiej w drugim wierszu mamy niezerowe poza diagonalne elementy a_{21}, a_{23}, a_{25} . itd. Inaczej ujmując, węzeł 1 nie ma na liście sąsiadów węzłów 3, 4, 5, 6. W pierwszym wierszu macierzy rzadkiej elementy a_{13}, a_{14}, a_{15}, a_{16} są zerowe. Węzeł 2 nie ma wśród swoich sąsiadów węzłów 4, 6. W drugim wierszu elementy a_{24}, a_{26} są zerowe. itd. Dokładnie taka sama zależność istnieje i dla układu numerowania (b). Zalecam czytnikowi rozważyć to samodzielnie.

Ogólny wniosek jest taki. Jeśli dowolna para węzłów *i*, *j* jest połączona chociażby jednym elementem skończonym, w macierzy rzadkiej powstaje niezerowy element poza diagonalny a_{ij}. W przeciwnym wypadku (para węzłów *i*, *j* nie jest łączona żadnym elementem skończonym) element macierzy $a_{ij} = 0$.

Podany przykład ukazuje, że struktura macierzy rzadkiej zależy od sposobu numerowania węzłów modelu MES (porównaj rys. 4.2,a i rys. 4.2,b). Ograniczymy się rozważeniem macierzy rzadkich symetrycznych. Niezerowa struktura takiej macierzy jest przedstawiona grafem spójności. Często taki graf jest nazywany grafem przyległości i oznaczany jako $G(X, E)$. Tu X jest zbiorem wierzchołków, a E – zbiorem krawędzi. Każdemu wierzchołkowi grafu w macierzy rzadkiej odpowiada kolumna albo wiersz, a każda krawędź grafu reprezentuje niezerowy poza diagonalny element macierzy. Diagonalne elementy są przedstawione pętlami, które wychodzą z podanego wierzchołku i zamykają się na nim. Zazwyczaj na tego typu grafie wspomniane pętle nie są pokazywane. Graf jest nieskierowany, ponieważ macierz jest macierzą symetryczną ($a_{ij} = a_{ji}$).

Będziemy uważać, że układ numerowania węzłów (równań macierzy rzadkiej) (a) (rys. 4.1, 4.2) jest podstawowym, czyli takim, jaki powstał przy tworzeniu modelu obliczeniowego. Przy przejściu od układu numerowania (a) do układu (b) macierz rzadka zmienia swoją strukturę. Procedura przejścia od jednego układu numerowania do innego otrzymała nazwę uporządkowania (reordering) układu równań. Na rys. 4.3 (a, b) są podane odpowiednie grafy przyległości dla każdego z układów numerowania. Dla układu (b) numery wierzchołków są oznaczone nad podstawowymi numerami.



Rys. 4.3 Grafy przyległości, reprezentujące macierze rzadkie, przedstawione na rys. 4.2 (a, b)

Przejście od jednego układu numerowania do innego jest przedstawione za pomocą tablicy permutacji (permutation array) *Perm*:

$$old\_number = Perm[new\_number] .$$

Tu *new_number* – numerowanie wierzchołków w układzie (b), a *old_number* – w podstawowym układzie (a). Dla podanego przykładu *Perm* = (3, 4, 2, 5, 1, 6) . Oznacza to, że pierwszym w układzie (b) jest wierzchołek 3 (rys. 4.3, b), drugim – 4, trzecim – 2, itd.

Przetwarzanie macierzy układu równań liniowych algebraicznych przy zmianie numerowania niewiadomych nazywa się permutacjami. Operatorem takiego przetwarzania służy kwadratowa macierz permutacji **P**, każda kolumna i każdy wiersz, której zawierają tylko jeden element – 1. Pozostałymi elementami są zera.

$$\mathbf{B} = \mathbf{P}^T \cdot \mathbf{A} \cdot \mathbf{P} \ , \tag{4.1}$$

gdzie **A** – macierz rzadka w podstawowym układzie numerowania, **P** – macierz permutacji, **B** – macierz rzadka w nowym układzie numerowania, przy czym $\mathbf{P} \cdot \mathbf{P}^T = \mathbf{I}$, **I** – jednostkowa macierz. Dla podanego przykładu

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & & & & \\ a_{21} & a_{22} & a_{23} & & a_{25} & \\ & a_{32} & a_{33} & a_{34} & & \\ & & a_{43} & a_{44} & a_{45} & \\ & a_{52} & & a_{54} & a_{55} & a_{56} \\ & & & & a_{65} & a_{66} \end{pmatrix}; \quad \mathbf{P} = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix};$$

$$\mathbf{B} = \begin{pmatrix} a_{33} & a_{34} & a_{32} & & & \\ a_{43} & a_{44} & & a_{45} & & \\ a_{23} & & a_{22} & a_{25} & a_{21} & \\ & a_{54} & a_{52} & a_{55} & & a_{56} \\ & & a_{12} & & a_{11} & \\ & & & a_{65} & & a_{66} \end{pmatrix}.$$

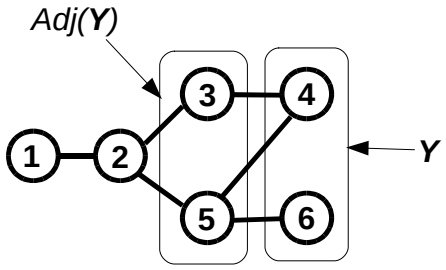
W pierwszej kolumnie macierzy permutacji **P** jedynka znajduje się w trzeciej pozycji. Odpowiada to pierwszemu elementowi tablicy *Perm*[1] = 3. W drugiej kolumnie jedynka znajduje się w czwartej pozycji – odpowiada to *Perm*[2] = 4. I

tak dalej. Oznacza to, że tablica permutacji *Perm* jednoznacznie definiuje macierz permutacji **P**.

Obliczenie bezpośrednie (4.1) dla dużych macierzy jest procedurą dość pracochłonną, dlatego w praktyce najpierw tworzy się tablicę permutacji *Perm*, a później – agreguje się macierz rzadką od razu w nowym układzie numerowania. Tworzenie tablicy permutacji *Perm* odbywa się na podstawie analizy grafu przyległości dla macierzy rzadkiej w podstawowym układzie numerowania. Dla tego trzeba stworzyć tylko ten graf i wykonać szereg działań, związanych z uporządkowaniem. Agregacja macierzy rzadkiej w podstawowym układzie numerowania nie jest potrzebna.

Nazywamy dwa wierzchołki grafa sąsiednimi, jeśli $\{x, y\} \in E$.

Jeśli $Y \subset X$, to zbiór przyległy dla Y jest $Adj(Y) = \{x \in X - Y | (\forall y \in Y)\{x, y\} \in E\}$. Na przykład, na rys. 4.4 dla zbioru Y, składającego się z wierzchołków 4, 6, zbiorem przyległym będą wierzchołki 3, 5. Żeby wyznaczyć wierzchołki zbioru przyległego wystarczy prześledzić za każdą krawędzią, łączącą wierzchołki zbioru Y z wierzchołkami zbioru $X - Y$. Te wierzchołki zbioru $X - Y$, jakie są sąsiadami wierzchołków zbioru Y, tworzą zbiór przyległy $Adj(Y)$ dla Y. Bierzemy wierzchołek 4 ze zbioru Y (rys. 4.4). Krawędzi 3-4 i 5-4 wychodzą poza granicy zbioru Y. Stąd wierzchołki 3, 5 należą do zbioru $Adj(Y)$. Z wierzchołku 6 wychodzi krawędź 6-5, ale wierzchołek 5 już znajduje się w zbiorze $Adj(Y)$. Każdy wierzchołek może wejść do każdego ze zbiorów tylko jednokrotnie.



Rys. 4.4 Zbiór przyległy *Adj*(*Y*) dla zbioru *Y*

Graf przyległości jest przedstawiony listą przyległości (tab. 4.1).

Tabela 4.1

Lista przyległości dla grafu, podanego na rys. 4.4

Wierzchołek	Lista sąsiadów
1	2
2	1, 3, 5
3	2, 4
4	3, 5
5	2, 4, 6
6	5

W algorytmach lista przyległości jest zrealizowana za pomocą następującej struktury danych – struktury przyległości (tab. 4.2).

Tabela 4.2

Przedstawienie grafu, podanego na rys. 4.4, za pomocą struktury przyległości

pos	1	2	3	4	5	6	7	8	9	10	11	12	13
List	2	1	3	5	2	4	3	5	2	4	6	5	
Pos	1		2		5		7		9		12		13
i	1		2		3		4		5		6		

W tabeli *List* wypisujemy po kolei listy sąsiadów dla wierzchołków 1, 2, ..., 6. W wierszu *pos* są podane indeksy elementów tablicy *List*. W wierszu *i* są umieszczone numery wierzchołków, a w wierszu *Pos* – wskaźnik pozycji *pos* w tablice *List* pierwszego elementu listy sąsiadów dla wierzchołku *i*. Dla obejrzenia sąsiadów dla wszystkich wierzchołków grafu wystarczy wykonać następujący fragment kodu:

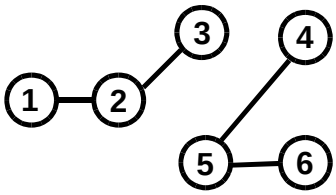
```
for(i=1; i<=N; ++i) //pętle po wierzchołkach grafu
{
    for(pos=Pos[i]; pos<Pos[i+1]; ++pos)
    {
          nabor = List[pos]; //sąsiad wierzchołku i
    }
}
```

Tu N – liczba wierzchołków grafu. Liczba sąsiadów dla wierzchołku *i* jest wyznaczona jako *Pos*[*i*+1]–*Pos*[*i*] . Na przykład, wierzchołek 2 ma *Pos*[*2*+1]–*Pos*[*2*] = 5 – 2 = 3 sąsiada, a wierzchołek 6 – *Pos*[*6*+1]–*Pos*[*6*] = 13 – 12 = 1 sąsiada. Dlatego że ogólny wzór powinien działać dla dowolnych wierzchołków grafu, ostatni element wiersza *pos* wskazuje na pierwszą wolną pozycje tablicy *List* . Ta wartość jest ustalona do elementu tablicy *Pos*[N+1].

Ścieżką od wierzchołku *x* do wierzchołku *y* długością *L* jest zbiór uporządkowany z *L*+1 różnych wierzchołków $(x_1, x_2, \ldots, x_{L+1})$, $x_{i+1} \in Adj(x_i)$, $1 \le i \le L$. Na przykład, na grafie, podanym na

rys. 4.4, pomiędzy wierzchołkami 1, 3 istnieje jedna ścieżka 1 – 2 – 3 i druga 1 – 2 – 5 – 4 – 3.

Odległością $d(x, y)$ pomiędzy dwoma wierzchołkami grafu spójnego $G(X, E)$ jest długość najkrótszej ścieżki, która łączy te wierzchołki. Graf spójny – to taki graf, dla którego każda para wierzchołków jest połączona przynajmniej jedną ścieżką. Na przykład, graf, przedstawiony na rys. 4.4, jest grafem spójnym, a graf, podany na rys. 4.5, jest grafem niespójnym, ponieważ nie istnieje żadnej ścieżki pomiędzy wierzchołkami 3, 4.



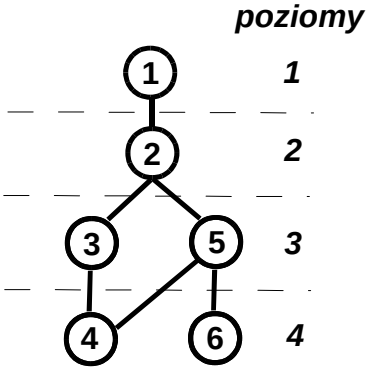
Rys. 4.5 Przykład niespójnego grafu

Mimośrodem wierzchołku x jest $l(x) = \max\{d(x, y) | y \in X\}$, czyli dla podanego wierzchołku x trzeba odnaleźć taki wierzchołek y, żeby odległość $d(x, y)$ była największą.

Średnica grafu: $\delta(G) = \max\{l(x) | x \in X\} = \max\{d(x, y) | x, y \in X\}$. Dla tego trzeba odnaleźć wierzchołek z największym mimośrodem albo taką parę wierzchołków (x, y), dla której odległość będzie największą.

Wierzchołek peryferyjny – to taki wierzchołek, mimośród, którego jest równy średnicy grafu: $l(x) = \delta(G)$.

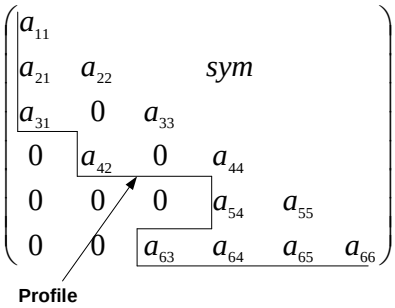
Na rys. 4.6 jest przedstawiona struktura poziomów z korzeniem w peryferyjnym wierzchołku 1 dla grafu przyległości, podanym na rys. 4.4.



Rys. 4.6 Struktura poziomów dla grafu, podanym na rys. 4.4, z korzeniem w wierzchołku 1

W pierwszym poziomie umieszczamy wierzchołek 1. Jego sąsiadem jest wierzchołek 2 – umieszczamy go na drugim poziomie. Sąsiadami wierzchołku 2 są wierzchołki 3, 5 i 1. Ponieważ 1 jest już włączony do struktury poziomów, to nie wolno dokładać go po raz drugi. Wierzchołki 3, 5 trafią na trzeci poziom. Pozostałych sąsiadów wierzchołków 3, 5 – są to wierzchołki 4, 6 – umieszczamy na czwarty poziom.

Dla przechowywania macierzy rzadkich opracowano kilka formatów specjalnych. Format profilowy jest stosowany dla przechowywania macierzy pasmowych (rys. 4.7).



Rys. 4.7 Portret macierzy rzadkiej, przechowywanej w formacie profilowym

Profilem będziemy nazywać linię, która przechodzi przez pierwsze niezerowe elementy każdego wiersza, licząc z lewa naprawo. W formacie profilowym są przechowywane wszystkie elementy, znajdujące się wewnątrz profili (tab. 4.3).

Tabela 4.3

Przechowywanie danych wiersz po wierszu w formacie profilowym dla macierzy, podanej na rys. 4.7

pos	1	2	3	4	5	6	7	8	9	10
Space	a_{21}	a_{31}	0	a_{42}	0	a_{54}	a_{63}	a_{64}	a_{65}	
Pos	1	1	2	4	6	7	10			
Diag	a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}				
i	1	2	3	4	5	6				

Tu *i* – numer wiersza macierzy, Diag – tablica jednowymiarowa dla przechowywania elementów diagonalnych, Space – tablica jednowymiarowa dla przechowywania elementów poza diagonalnych, umieszczonych w profilu wiersz po wierszu, Pos[*i*] jest indeksem pierwszego niezerowego elementu tablicy Space, z którego zaczyna się wiersz *i*, pos – indeks elementu tablicy Space.

Wiersz 2 zaczyna się z elementu a_{21}, który znajduje się w pozycji 1 tablicy Space – Pos[2] = 1. Wiersz 3 zaczyna się z elementu a_{31} , który znajduje się w

pozycji 2 tablicy Space – Pos[3] = 2. Wiersz 4 zaczyna się z elementu a_{41} , który znajduje się w pozycji 4 tablicy Space – Pos[4] = 4. Itd.

Żeby po kolei pobrać elementy poza diagonalne wierszu *i*, trzeba wykonać taki fragment kodu:

```
j = i-(Pos[i+1]-Pos[i]);  //indeks j pierwszego elementu w
                          //profilu
for(pos=Pos[i]; pos<Pos[i+1]; ++pos, ++j)
{
    a_ij = Space[pos];
}
```

Liczba elementów poza diagonalnych w wierszu *i* jest liczona jako Pos[*i*+1] – Pos[*i*]. Na przykład, w pierwszym wierszu znajduje się Pos[2] – Pos[1] = 1 – 1 = 0 elementów poza diagonalnych, w czwartym wierszu – Pos[5] – Pos[4] = 6 – 4 = 2 elementy, a w ostatnim wierszu – Pos[7] – Pos[6] = 10 – 7 = 3. Dla tego indeks pierwszej wolnej pozycji tablicy Space pozostał przeniesiony do elementu *N*+1 tablicy Pos, gdzie *N* – liczba wierszy (równań) w macierzy.

Format skompresowany (CSR – compressed sparse row, CSC – compressed sparse column), używany dla przechowywania macierzy rzadkich, zawiera tylko niezerowe elementy. Istnieje kilka typów formatów skompresowanych. One są podobne, dlatego wystarczy rozważyć jeden z nich. Będziemy umieszczać niezerowe elementy poza diagonalne w tablicy Space (tab. 4.4) wiersz po wierszu.

Tabela 4.4

Przechowywanie danych w formacie skompresowanym (wiersz po wierszu – CSR) dla macierzy, podanej na rys. 4.7

pos	1	2	3	4	5	6	7	8
Space	a_{21}	a_{31}	a_{42}	a_{54}	a_{63}	a_{64}	a_{65}	
jnd	1	1	2	4	3	4	5	
Pos	1	1	2	3	4	5	8	
Diag	a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	a_{66}		
i	1	2	3	4	5	6		

Jeśli w formacie profilowym wartość indeksu *j* można policzyć dla każdego elementu poza diagonalnego, to w formacie skompresowanym indeks *j* jest przechowywany w tablice *jnd*.

Żeby po kolei pobrać elementy poza diagonalne wierszu *i*, trzeba wykonać taki fragment kodu:

```
for(pos=Pos[i]; pos<Pos[i+1]; ++pos)
{
    a_ij = Space[pos];
    j    = jnd[pos];
}
```

Jak i w poprzednim wypadku, liczba elementów poza diagonalnych w wierszu *i* jest liczona jako Pos[*i*+1] – Pos[*i*] .

Formaty profilowe i skompresowane, w jakich elementy poza diagonalne są przechowywane kolumna po kolumnie (CSC), są podobne do przedstawionych powyżej. Polecam czytnikowi rozważyć ich samodzielnie.

4.2 Wpływ uporządkowania na skuteczność rozwiązania układów równań liniowych algebraicznych metodami bezpośrednimi.

Dla sfaktoryzowanej macierzy rzadkiej liczba elementów niezerowych istotnie zależy od sposobu numerowania równań. Wykonamy jeden krok eliminacji Gaussa dla podanej poniżej macierzy.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & 0 & 0 \\ a_{31} & 0 & a_{33} & 0 \\ a_{41} & 0 & 0 & a_{44} \end{pmatrix} \rightarrow \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & \tilde{a}_{22} & \tilde{a}_{23} & \tilde{a}_{24} \\ 0 & \tilde{a}_{32} & \tilde{a}_{33} & \tilde{a}_{34} \\ 0 & \tilde{a}_{42} & \tilde{a}_{43} & \tilde{a}_{44} \end{pmatrix} \tag{4.2}$$

Pierwszy wiersz pozostawiamy bez zmian. Dalej dobieramy do pierwszego wiersza taki mnożnik λ_{21}, żeby $a_{21} - a_{11}\lambda_{21} = 0 \rightarrow \lambda_{21} = a_{21}/a_{11}$. Mnożymy pierwszy wiersz przez mnożnik λ_{21} i odejmujemy od drugiego. Drugi element pierwszej kolumny będzie równy zero, ponieważ tak dobraliśmy mnożnik λ_{21}. Element a_{22} ulegnie zmianie, dla tego oznaczymy jego symbolem "~" z góry. Elementy macierzy źródłowej $a_{23} = a_{24} = 0$. Po wykonaniu odejmowania na pozycjach zerowych elementów drugiego wiersza powstają niezerowe elementy $\tilde{a}_{23} = 0 - \lambda_{21}a_{13}$, $\tilde{a}_{24} = 0 - \lambda_{21}a_{14}$. Takie niezerowe elementy macierzy sfaktoryzowanej będziemy nazywać zapełnieniami.

Dalej powtarzamy tą procedurę z elementami trzeciego wiersza. Dobieramy mnożnik λ_{31} tak, żeby $a_{31} - a_{11}\lambda_{31} = 0 \rightarrow \lambda_{31} = a_{31}/a_{11}$, i odejmujemy od trzeciego wiersza pierwszy, mnożony przez λ_{31}. Powstają zapełnienia $\tilde{a}_{32}, \tilde{a}_{34}$. itd. Po pierwszym kroku eliminacji Gaussa pozostała podmacierz jest gęsta, dlatego górna macierz trójkątna też będzie macierzą gęstą, zawierającą trzy zapełnienia: $\tilde{a}_{23}, \tilde{a}_{24}, \tilde{\tilde{a}}_{34}$.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & \tilde{a}_{22} & \tilde{a}_{23} & \tilde{a}_{24} \\ 0 & 0 & \approx{a}_{33} & \approx{a}_{34} \\ 0 & 0 & 0 & \approx\tilde{a}_{44} \end{pmatrix} \tag{4.3}$$

Liczba powtórzeń znaku "~" jest równa ilości modyfikacji odpowiedniego elementu w trakcie faktoryzacji tej macierzy metodą Gaussa.

Teraz przestawimy pierwszą kolumnę i czwartą oraz pierwszy wiersz i czwarty. Macierze źródłowa i górna trójkątna mają postać:

$$\begin{pmatrix} a_{44} & 0 & 0 & a_{41} \\ 0 & a_{22} & 0 & a_{21} \\ 0 & 0 & a_{33} & a_{31} \\ a_{14} & a_{12} & a_{13} & a_{11} \end{pmatrix} \rightarrow \begin{pmatrix} a_{44} & 0 & 0 & a_{41} \\ 0 & \tilde{a}_{22} & 0 & \tilde{a}_{21} \\ 0 & 0 & \approx{a}_{33} & \approx{a}_{31} \\ 0 & 0 & 0 & \approx\tilde{a}_{11} \end{pmatrix} \tag{4.4}$$

Tu nie powstało żadne zapełnienie.

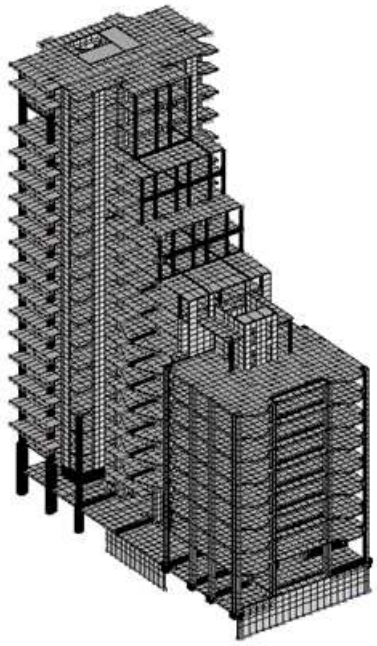
Ten przykład podkreśla, że liczba zapełnień przy sfaktoryzowaniu macierzy rzadkich istotnie zależy od kolejności eliminowania równań albo, co jest tym samym, od sposobu ich numerowania w macierzy. Dobór odpowiedniej sekwencji numerowania równań z celą zmniejszenia ilości zapełnień w macierzy sfaktoryzowanej nazywa się uporządkowaniem.

Rozważmy dalej problem rzeczywisty z dziedziny mechaniki konstrukcji, pobrany z biblioteki zadań SCAD Soft [SCAD] – firmy informatycznej, produkującej oprogramowanie MES dla projektowania i obliczeń konstrukcji budowlanych. Na rys. 4.8 jest przedstawiony model MES budynku wielopiętrowego, zawierający 115 362 równań.

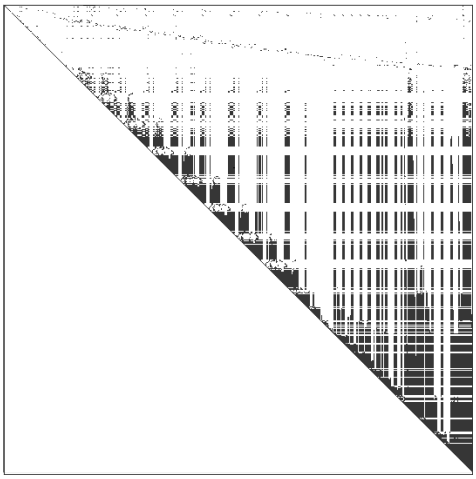
Na rys. 4.9 – 4.15 są portrety macierzy rzadkiej dla tego zadania przy różnych metodach uporządkowania równań. Macierz jest symetryczna, dlatego jest podana tylko górna część. Oznaczono również liczbę niezerowych elementów w macierzy sfaktoryzowanej w megabajtach dla każdej metody uporządkowania.

Przy braku uporządkowania (rys. 4.9) numerowanie węzłów jest takie, jakie powstało przy tworzeniu modelu obliczeniowego przez preprocesor programu SCAD. Przy zastosowaniu algorytmu reverse Cuthill-McKee – RCM [George and Liu, 1981], który pragnie zminimalizować szerokość profile (rys. 4.10), liczba elementów niezerowych macierzy sfaktoryzowanej zmniejsza się ponad dwa razy.

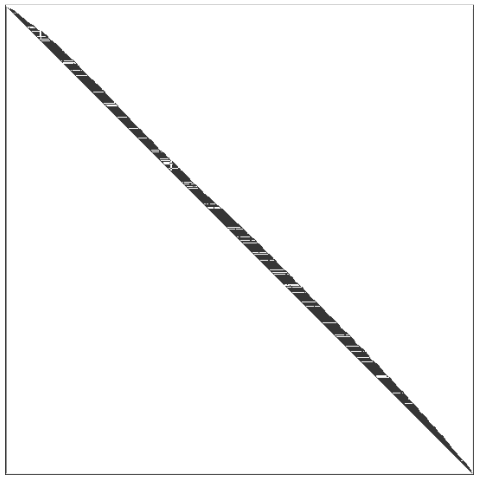
Trochę lepszy wynik udaje się uzyskać metodą Sloana [Sloan S. W., 1986] – rys. 4.11.



Rys. 4.8 Model MES budynku wielopiętrowego (115 362 równań)

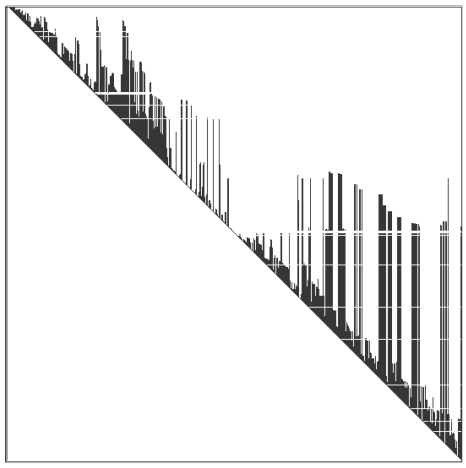


Rys. 4.9 Brak uporządkowania. 3 741 MB

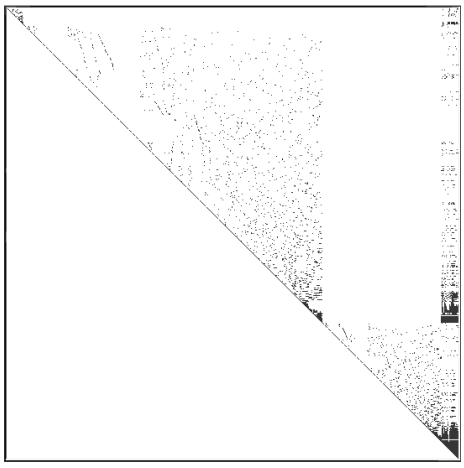


Rys. 4.10 Uporządkowanie algorytmem RCM. 1 618 MB

Metoda równoległych przekrojów (parallel section method – PSM) doprowadzi macierz do postaci blokowo-przekątnej z obramowaniem. Jeśli jeszcze wewnątrz każdego bloku zastosować algorytm minimalnego stopnia w wersji MMD [George and Liu, 1989], otrzymamy portret macierzy, przedstawionej na rys. 4.12.

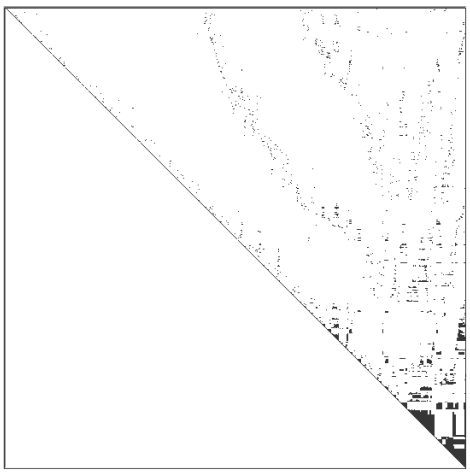


Rys. 4.11 Uporządkowanie metodą Sloana 1 386 MB

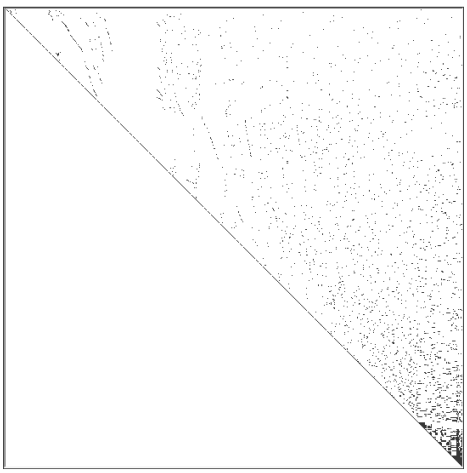


Rys. 4.12 Uporządkowanie metodą PSM+MMD. 345 MB

Metoda włożonych przekrojów (nested dissection method – ND) [George and Liu, 1981] rekursywnie dzieli obszar obliczeniowy na dwa równe podobszary (rys. 4.13). Po zastosowaniu algorytmu minimalnego stopnia w wersji MMD [George and Liu, 1989] portret macierzy przyjmuje postać, podany na rys. 4.14.

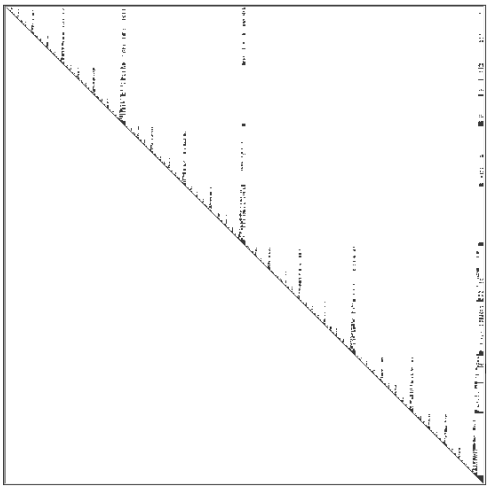


Rys. 4.13 Uporządkowanie metodą ND. 644 MB



Rys. 4.14 Uporządkowanie metodą MMD. 191 MB

Zastosowanie metody METIS [Karypis and Kumar, 1995] prowadzi do rozmiaru macierzy sfaktoryzowanej, bliskiemu do metody MMD (rys. 4.15).



Rys. 4.15 Uporządkowanie metodą METIS.
193 MB

Ostateczne wyniki są podane w tab. 4.5.

Tabela 4.5

Liczba niezerowych elementów macierzy sfaktoryzowanej dla różnych metod uporządkowania

Metoda uporządkowania	Liczba niezerowych elementów w macierzy	Rozmiar macierzy sfaktoryzowanej w MB
Brak uporządkowania	490 366 701	3 741
RCM	212 143 113	1 618
Metoda Sloana	181 750 005	1 386
PSM+MMD	45 281 385	345
ND	84 522 753	644
MMD	25 142 373	191
METIS	25 341 381	193

Różne metody uporządkowania prowadzą do różnej ilości elementów niezerowych w macierzy sfaktoryzowanej. Przy braku uporządkowania solwer – moduł, jaki realizuje rozwiązywanie układu równań liniowych algebraicznych, powinien przetworzyć 3 741 MB danych. W przypadku zastosowania uporządkowania algorytmem minimalnego stopnia MMD objętość danych zmniejsza się ponad 19 razy. Oznacza to, że czas rozwiązania takiego problemu udaję się skrócić ponad 100 razy.

Podany przykład demonstruje, że wybór odpowiedniej metody uporządkowania wielokrotnie zmniejsza objętość danych, przetwarzanych solwerem, oraz istotnie zmniejsza czas rozwiązywania zadania.

4.3 Podstawowe algorytmy uporządkowania. Metoda włożonych przekrojów i algorytm minimalnego stopnia. Algorytmy hybrydowe, METIS

Jak widać z materiału, przedstawionego w poprzednim rozdziale, przy faktoryzacji macierzy rzadkich powstaje problem odnalezienia takiej kolejności numerowania równań – takiego uporządkowania, przy którym solwer będzie działał w sposób optymalny. Najczęściej sposób optymalny oznacza osiągnięcie najmniejszej ilości zapełnień. W praktyce jednak spotka się zadania, dla których minimalny rozmiar macierzy sfaktoryzowanej nie będzie odpowiadał najmniejszej trwałości faktoryzacji.

W obrębie podanego kursu w celu optymalizacji będziemy dowiadywać się, jak osiągnąć najmniejszą ilość zapełnień. Rozwiązanie takiego zadania w sposób ścisły jest bardzo pracochłonne, szybszym sposobem będzie rozwiązanie układu równań liniowych algebraicznych bez żadnego uporządkowania. Dlatego w praktyce zamiast ścisłego rozwiązania zadania odnalezienia optymalnego uporządkowania zastosowują algorytmy heurystyczne. Stąd wynika, że tych algorytmów jest kilka, żaden z nich nie prowadzi do rozwiązania optymalnego, a podaje tylko lepsze albo gorsze przybliżenie do rozwiązania optymalnego, i dla rozwiązywanego zadania z góry nie wiadomo, jaki algorytm uporządkowania doprowadzi do najlepszego wyniku.

Rozważmy dwie różne idee. Pierwsza była zrealizowana metodą włożonych przekrojów (ND – nested dissection), a druga – algorytmem minimalnego stopnia (MDA – minimum degrees algorithm).

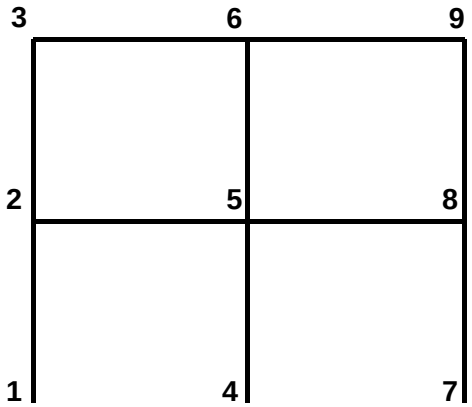
W metodzie **włożonych przekrojów** (ND – nested dissection) jest dany spójny graf $G(X, E)$. Zbiór węzłów S tworzy separator. Oznacza to, że usunięcie tych węzłów z grafu G powoduje rozdzielenie go na dwa lub więcej spójnych podgrafy $G1, G2, \ldots$. Algorytm metody ND wygląda następująco:

1. Tworzymy strukturę poziomów z korzeniem w pseudo peryferyjnym wierzchołku. Ustawiamy $i=1$.
2. Zbiór wierzchołków, który znajduje się na poziomie środkowym struktury poziomów, zaznaczamy jako separator S_i, jeśli liczba poziomów $Nlvl > 2$, i zapisujemy te wierzchołki w tablice permutacji *Perm*. Jeśli $Nlvl \leq 2$, cały podgraf traktujemy jako nierozdzielny i zapisujemy w *Perm*.

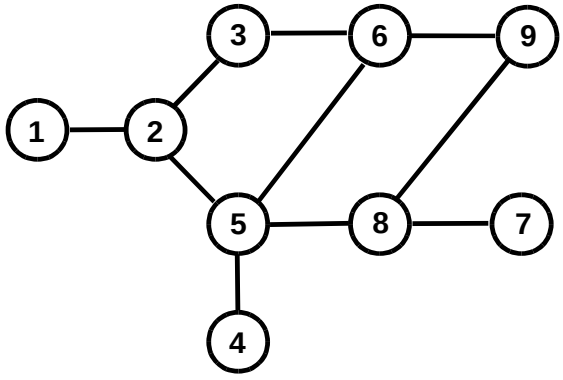
3. Usuwamy zbiór S_i z grafu, graf pozostaje rozdzielony na podgrafy, inkrementujemy i++.
4. Dla każdego z podgrafów tworzymy strukturę poziomów z korzeniem w pseudo peryferyjnym wierzchołku i przechodzimy do punktu 2.
5. Algorytm renumeracji będzie przerwany, kiedy wszystkie wierzchołki zostaną przenumerowane.

Na każdym kroku podziału graf rozdziela się na dwa (lub kilka) mniej-więcej równoważne podgrafy, ponieważ separatory są wybrane z poziomu środkowego. Dla tego tą metodę nazywają jeszcze metodą rekursywnych bisekcji.

Rozważmy następujący przykład. Dla ramy płaskiej (rys. 4.16) graf przyległości jest podany na rys. 4.17.

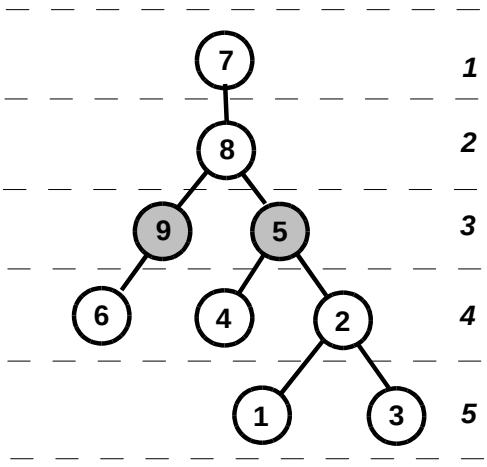


Rys. 4.16 Rama płaska



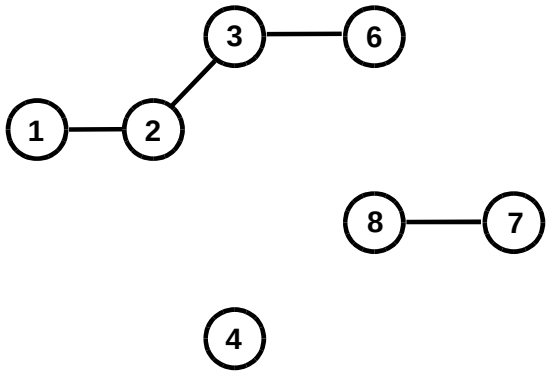
Rys. 4.17 Graf przyległości dla ramy płaskiej, podanej na rys. 4.16

Struktura poziomów z korzeniem w wierzchołku peryferyjnym 7 jest podana na rys. 4.18.

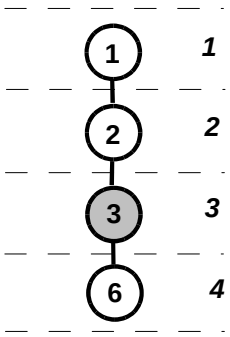


Rys. 4.18 Struktura poziomów z korzeniem w wierzchołku peryferyjnym 7 dla grafu przyległości, podanym na rys. 4.17. Zbiór wierzchołków środkowego poziomu 3 tworzy separator $S_1 = \{9,5\}$

Wierzchołki 9, 5 poziomu środkowego 3 tworzą separator $S_1 = \{9, 5\}$. Graf przyległości po usunięciu wierzchołków 9, 5 rozpada się na podgrafy (rys. 4.19). Dla podgrafu 1-2-3-6 znów tworzymy strukturę poziomów z korzeniem w wierzchołku peryferyjnym 1 (rys. 4.20), a pozostałe podgrafy są nierozdzielne, ponieważ zawierają nie więcej niż dwóch poziomów ($Nlvl \leq 2$).



Rys. 4.19 Po usunięciu wierzchołków 9, 5 graf przyległości (rys. 4.17) rozpada się na podgrafy



Rys. 4.20 Struktura poziomów dla grafu 1-2-3-6 z korzeniem w wierzchołku 1. Poziom 3 tworzy separator S_2 = { 3 }

Tablicę permutacji wypełniamy od końca do początku. W końcu umieszczamy separatory w kolejności S_1, S_2, a później – wierzchołki z podgrafów nierozdzielnych:

$$Perm = \left\{ 7, 8, 6, 4, 1, 2, \underbrace{3}_{S_2}, \underbrace{9, 5}_{S_1} \right\}.$$

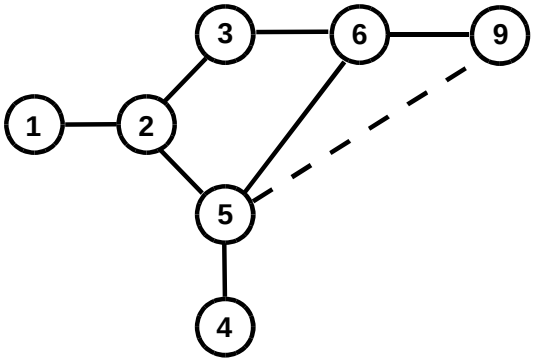
Tu S_1={9, 5}. Jednakże można umieścić w pierwszym separatorze te same wierzchołki w odwrotnej kolejności: S_1={5, 9}. Chociaż spowoduje to inną tablicę permutacji, z punktu widzenia głównej idei metody włożonych przekrojów są to równoważne rozwiązania. Struktura poziomów na rys. 4.20 ma parzystą liczbę poziomów, stąd wynika, że jako poziom "środkowy" można przyjąć albo poziom 2, albo 3. W pierwszym wypadku S_2 = {2}, a w drugim – S_2 = 3. Są to też równoważne rozwiązania dla metody ND.

Wierzchołki z podgrafów nierozdzielnych można umieścić w dowolnej kolejności, ponieważ nie ma znaczenia, którą z tych podstruktur będziemy potraktować, jako pierwszą, a jaką –drugą.

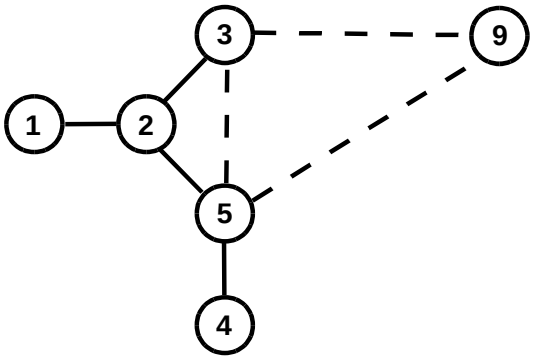
Wychodzi tak, że metoda włożonych przekrojów dla tego samego grafu przyległości może prowadzić do różnych tablic permutacji, jednakże te tablice będą równoważne z punktu widzenia idei, na której polega ta heurystyczna metoda.

Przy eliminowaniu równań w kolejności, ustalonej przez tablicę permutacji, w macierzy rzadkiej powstają zapełnienia. Na podstawie twierdzenia D. J. Rose [Rose D. J., 1972] można przy pomocy grafu przyległości wyjaśnić, w których pozycjach powstają zapełnienia. Dla tego trzeba krok po kroku wycinać z grafu wierzchołki z numerami, które odpowiadają kolejności, ustalonej przez tablicę permutacji *Perm*.

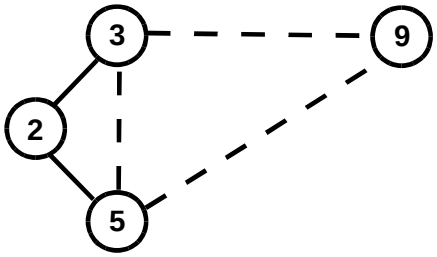
Na każdym kroku dla wyciętego wierzchołku znajdujemy zbiór przyległy – jego sąsiadów, następnie wszystkie pary tych wierzchołków łączymy krawędziami. Wycinamy wierzchołek 7. Jego sąsiadem jest wierzchołek 8. W grafie nie powstaje żadna nowa krawędź. Dalej wycinamy wierzchołek 8. Zbiorem przyległym dla niego są wierzchołki 5, 9. Łączymy ich krawędzią – w grafie powstaje nowa krawędź (linia przerywana), a w macierzy – zapełnienie a_{59} – rys. 4.21. Wycinamy wierzchołek 6. Sąsiadami wierzchołku 6 są wierzchołki 3, 5, 9. Łączymy ich krawędziami – powstają zapełnienia a_{53} i a_{93} (rys. 4.22). Wycinamy dalej wierzchołki 4, 1. Nowe zapełnienia nie powstają (rys. 4.23). Wycinamy wierzchołek 2 (rys. 4.24). Nowe zapełnienia nie powstają, ponieważ wierzchołki 3, 5 już są połączone. Pozostały graf jest kliką, ponieważ każdy wierzchołek jest połączony z innymi. W macierzy klicie odpowiada gęsta podmacierz, dlatego nowe zapełnienia już nie powstaną.



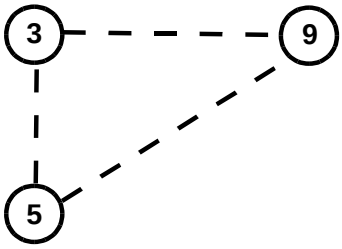
Rys. 4.21 Graf przyległości po wyeliminowaniu wierzchołków 7, 8. Powstaje zapełnienie a_{59}



Rys. 4.22 Graf przyległości po wyeliminowaniu wierzchołków 7, 8, 6. Powstają zapełnienia a_{53} i a_{93}

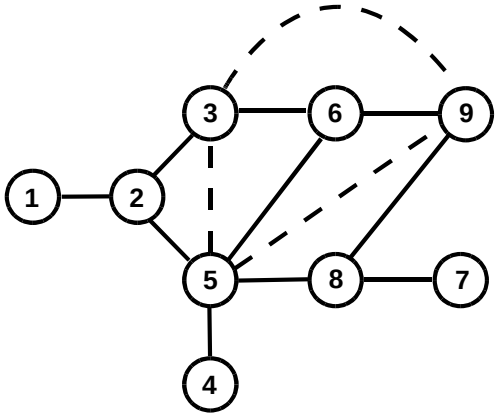


Rys. 4.23 Graf przyległości po wyeliminowaniu wierzchołków 7, 8, 6, 4, 1



Rys. 4.24 Graf przyległości po wyeliminowaniu wierzchołków 7, 8, 6, 4, 1, 2

Na rys. 4.25 jest podany faktor-graf, który reprezentuje niezerowe pozadiagonalne elementy macierzy źródłowej oraz zapełnienia.



Rys. 4.25 Faktor-graf reprezentuje niezerowe pozadiagonalne elementy macierzy źródłowej oraz zapełnienia

Na rys. 4.26 jest przedstawiony portret macierzy sfaktoryzowanej. Niezerowe pozadiagonalne elementy macierzy źródłowej są oznaczone jako "*x*". Symbolem "#" pokazujemy zapełnienia.

$$\begin{pmatrix}
7 & x & & & & & & & \\
x & 8 & & & & & & x & x \\
 & & 6 & & & & x & x & x \\
 & & & 4 & & & & & x \\
 & & & & 1 & x & & & \\
 & & & & x & 2 & x & & x \\
 & & x & & & x & 3 & \# & \# \\
 & x & x & & & & \# & 9 & \# \\
 & x & x & x & & x & \# & \# & 5
\end{pmatrix}$$

Rys. 4.26 Portret macierzy sfaktoryzowanej przy uporządkowaniu metodą ND

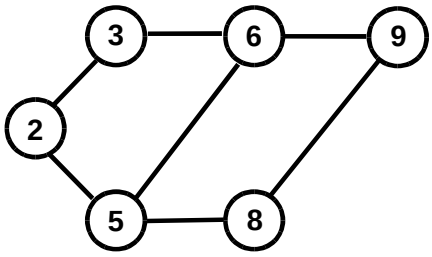
Przedstawiona powyżej procedura otrzymała nazwę faktoryzacji symbolicznej. Daje ona możliwość wyjaśnienia niezerowej struktury macierzy sfaktoryzowanej dla podanego uporządkowania, które jest przedstawione tablicą permutacji *Perm*. W trakcie symbolicznej faktoryzacji wszystkie działania są wykonane na grafie przyległości bez obliczeń wartości elementów macierzy rzadkiej. Szybkość tej procedury zezwala na zwykłych PC w przeciągu kilku sekund przeanalizować macierze rzadkie o rozmiarze do kilka milionów równań i wyjaśnić ich niezerową strukturę. Oprócz tego można spróbować kilka różnych algorytmów uporządkowania oraz wybrać taki, który doprowadzi do mniejszej ilości elementów niezerowych w macierzy sfaktoryzowanej. Wynikiem faktoryzacji symbolicznej jest wypełnione tablicy *Pos* i *jnd* formatu skompresowanego. Daje to możliwość wypełnienia tablicy *Space* przy agregowaniu macierzy źródłowej.

Algorytm minimalnego stopnia (MDA – minimum degrees algorithm). Główna idea tej metody polega na tym, że w każdym kroku eliminujemy równanie, które ma najmniejszą ilość niezerowych elementów w wierszu macierzy. "W średnim" powoduje to największe zmniejszenie ilości zapełnień. Takiemu równaniu w grafie przyległości odpowiada wierzchołek z minimalną ilością sąsiadów.

Ilość sąsiadów wierzchołku (albo ilość wychodzących z niego krawędzi) będziemy nazywać jego stopniem. Nazwa tej metody powstała z tego, że w każdym kroku eliminujemy z grafu wierzchołek minimalnego stopnia. Algorytm tej metody jest przedstawiony poniżej.

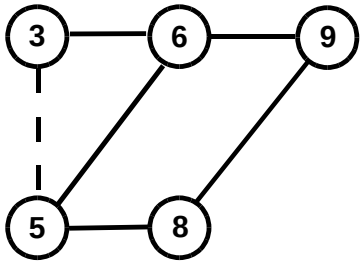
1. Znajdujemy wierzchołek, stopień, którego jest minimalny.
2. Eliminujemy ten wierzchołek z grafu. Wyznaczamy zbiór przyległy dla tego wierzchołku.
3. Wszystkie pary wierzchołki zbioru przyległego łączymy krawędziami pomiędzy sobą.
4. Przeliczamy stopnie pozostałych wierzchołków. Wracamy do punktu 1.

Algorytm zakończy się w chwili, kiedy wszystkie wierzchołki zostaną wyeliminowane. Etapy 2, 3 są wykonane na podstawie twierdzenia D. J. Rose [Rose D. J., 1972]. Rozważmy zadanie, przedstawione na rys. 4.16. Graf przyległości macierzy źródłowej jest podany na rys. 4.17. Wierzchołki 1, 4, 7 mają stopień 1. Jest to stopień minimalny. Dowolny z tych wierzchołków może być wyeliminowany. Eliminujemy te wierzchołki właśnie w takiej kolejności. Zapełnienia przy tym nie powstają. Pozostały graf przyległości jest przedstawiony na rys. 4.27.



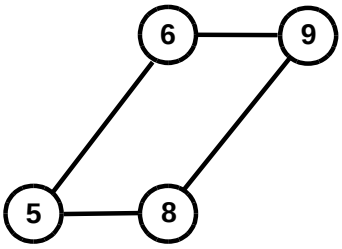
Rys. 4.27 Graf przyległości po wyeliminowaniu wierzchołków 1, 4, 7

Wierzchołki 2, 3, 9, 8 mają stopień 2. Jest to stopień minimalny. Możemy wyeliminować dowolny z tych wierzchołków. Niech to będzie wierzchołek 2. Po jego wyeliminowaniu powstaje zapełnienie a_{35} – rys. 4.28.



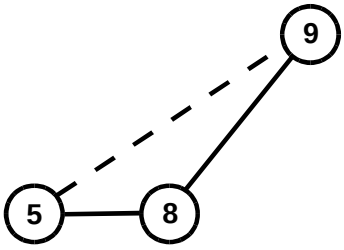
Rys. 4.28 Graf przyległości po wyeliminowaniu wierzchołków 1, 4, 7, 2

Wierzchołki 3, 9, 8 mają stopień 2. Jest to stopień minimalny. Eliminujemy wierzchołek 3 (rys. 4.29). Nowe zapełnienia nie powstają.



Rys. 4.29 Graf przyległości po wyeliminowaniu wierzchołków 1, 4, 7, 2, 3

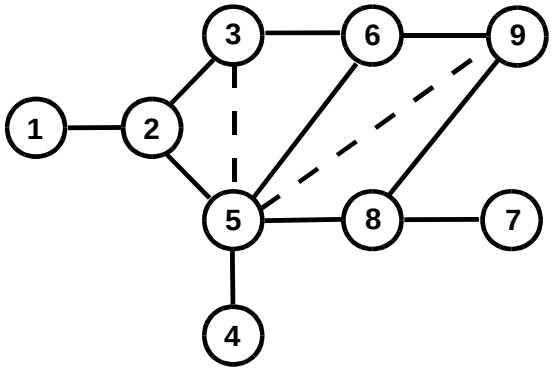
Wszystkie wierzchołki mają stopień 2. Eliminujemy wierzchołek 6. Powstaje zapełnienie a_{59} – rys. 4.30.



Rys. 4.30 Graf przyległości po wyeliminowaniu wierzchołków 1, 4, 7, 2, 3, 6

Pozostały graf jest kliką. Każdy z wierzchołków ma stopień 2. Możemy wyeliminować wierzchołki w dowolnej kolejności – nowe zapełnienia nie powstają.

Otrzymaliśmy tablicę permutacji *Perm* = (1, 4, 7, 2, 3, 6, 9, 5, 8). Faktor-graf dla takiej kolejności eliminowania wierzchołków jest podany na rys. 4.31.



Rys. 4.31 Faktor-graf dla kolejności eliminowania wierzchołków *Perm* = (1, 4, 7, 2, 3, 6, 9, 5, 8)

Niezerowa struktura macierzy sfaktoryzowanej jest podana na rys. 4.32.

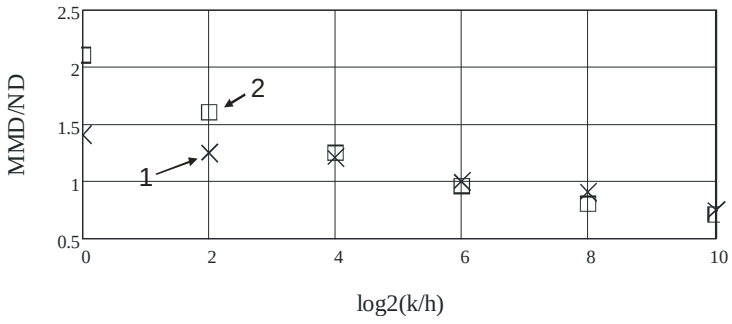
$$\begin{pmatrix}
1 & & & x & & & & \\
 & 4 & & & & & x & \\
 & & 7 & & & & & x \\
x & & & 2 & x & & x & \\
 & & & x & 3 & x & \# & \\
 & & & & x & 6 & x & x \\
 & & & & & x & 9 & \# & x \\
 & x & & x & \# & x & \# & 5 & x \\
 & & x & & & & x & x & 8
\end{pmatrix}$$

Rys. 4.32 Niezerowa struktura macierzy sfaktoryzowanej przy uporządkowaniu algorytmem minimalnego stopnia

Dla tej samej macierzy źródłowej algorytm minimalnego stopnia może doprowadzać do różnych tablic permutacji, ponieważ często się zdarza, że na tym samym kroku eliminacji będą kilka wierzchołków o takim samym stopniu. Każdy z nich może być wybrany do eliminowania. Wszystkie te tablice permutacji są równoważne z punktu widzenia podanej metody.

Typowe portrety macierzy rzadkich dla metod ND i MDA są podane odpowiednio na rys. 4.13 i 4.14.

Na rys. 4.33 jest przedstawione porównywanie skuteczności tych metod dla zadania testowego [Ashcraft and Liu, 1996].



Rys. 4.33 Porównywanie skuteczności algorytmu minimalnego stopnia w wersji MMD i metody włożonych przekrojów ND

Rozważmy obszar prostokątny z siatką, która ma *k* węzłów we wzdłużnym kierunku i *h* węzłów – w poprzecznym. Wzdłuż osi ordynat (krzywa 1) odkładamy stosunek ilości operacji arytmetycznych *ops*, wykonanych w trakcie faktoryzacji macierzy sztywności przy uporządkowaniu algorytmem minimalnego stopnia wersji MMD [George and Liu, 1989] do ilości operacji, wykonanych przy uporządkowaniu metodą włożonych przekrojów (ND). Krzywa 2 reprezentuje stosunek elementów niezerowych *nz* w macierzy sfaktoryzowanej przy uporządkowaniu metodą MMD do ilości niezerowych elementów przy uporządkowaniu metodą ND.

W punktach przecięcia krzywych 1, 2 z prostą MMD/ND = 1 porównywalne metody są równoważne. Przy MMD/ND > 1 (powyżej jedynki) przewagę ma metoda ND, ponieważ doprowadzi do wykonania mniejszej ilości operacji arytmetycznych albo do mniejszej ilości zapełnień, a przy MMD/ND < 1 (poniżej jedynki) przewagę ma metoda MMD.

Dla obszaru kwadratowego ($\log_2(k/h)=0 \rightarrow k/h=1$) bardziej skuteczne jest uporządkowanie metodą ND, ponieważ $ops(MMD)/ops(ND)>1 \wedge nz(MMD)/nz(ND)>1$. W przypadku obszarów wyciągniętych algorytm minimalnego stopnia jest bardziej efektywny: $ops(MMD)/ops(ND)<1 \wedge nz(MMD)/nz(ND)<1$.

Algorytm minimalnego stopnia jest zbyt „krótkowzroczny” – dobrze przewiduje dla stosunkowo nie wielu kroków faktoryzacji. Metoda włożonych przekrojów jest zbyt „dalekowzroczna”, ponieważ ma słabe przewidywanie bliskich kroków faktoryzacji.

Powstaje pytanie – czy można stworzyć taką metodę, która by łączyła zalety MMD oraz ND? Okazuje się, że tak, można. **Metody hybrydowe** wykonują względnie nie wiele kroków metodą ND. Powoduje to podzielenie obszaru obliczeniowego na włożone równoważne podobszary relatywnie nie wielkiego rozmiaru. Dalej dla każdego podobszaru jest stosowany algorytm MMD, który na niewielkim podobszarze działa skutecznie.

Dla zadania, przedstawionego na rys. 4.16 (graf przyległości dla macierzy źródłowej – rys. 4.17) wykonamy jeden krok metodą ND (rys. 4.18) i znajdziemy separator S_1={9, 5}. Usuwamy wierzchołki separatora z grafu, który rozpadnie się na podgrafy (rys. 4.19). Podgrafy 4 oraz 8-7 są nierozdzielne, a dla podgrafu 1-2-3-6 zastosowujemy algorytm minimalnego stopnia (kolejność eliminowania jest 6, 1, 2, 3). Tablicę permutacji tworzymy w następujący sposób: w samym końcu umieszczamy separator S_1, a przed nim – uporządkowane wierzchołki podgrafu 1-2-3-6 oraz wierzchołki podgrafów 8-7 i 4. Otrzymujmy Perm = (6, 1, 2, 3, 8, 7, 4, 9, 5). Po wykonaniu faktoryzacji symbolicznej (pozostawiamy to czytnikowi) powstaje niezerowa struktura macierzy sfaktoryzowanej (rys. 4.34).

$$\begin{pmatrix}
6 & & & x & & & & x & x \\
 & 1 & x & & & & & & \\
 & x & 2 & x & & & & & x \\
x & & x & 3 & & & & \# & \# \\
 & & & & 8 & x & & x & x \\
 & & & & x & 7 & & \# & \# \\
 & & & & & & 4 & & x \\
x & & & \# & x & \# & & 9 & \# \\
x & & x & \# & x & \# & x & \# & 5
\end{pmatrix}$$

Rys. 4.34 Niezerowa struktura macierzy sfaktoryzowanej przy uporządkowaniu metodą hybrydową

Na podstawie doświadczeń wykryto, że najczęściej lepsze wyniki dla dowolnego uporządkowania udaje się uzyskać, jeśli używać grafu zgrubionego zamiast grafu szczególnego. Jednym z podejść naturalnych jest użycie grafu dla węzłów modelu obliczeniowego zamiast grafu dla równań, wychodząc z tego, że każdemu węzłowi odpowiada grupa równań.

Do metod hybrydowych należy **METIS** [Karypis and Kumar, 1995]. Algorytm tej metody jest przedstawiony poniżej.

1. Zgrubiamy podany graf przyległości.
2. Na grafie zgrubionym wykonujemy 2^n kroki bisekcji rekursywnych metodą włożonych przekrojów (n jest wartością stosownie nie wielką – metoda włożonych przekrojów nie jest zakończona i znajdujemy 2^n separatory na grafie zgrubionym).
3. Znajdujemy separatory na grafie źródłowym – stosujemy *refinement* dla separatorów odnalezionych.
4. Uporządkowanie w obrębie każdego spójnego podgrafu wykonujemy algorytmem minimalnego stopnia.

W praktyce METIS dla obszarów kwadratowych demonstruje wyniki, bardzo bliskie do metody ND, a dla obszarów wyciągniętych – wyniki, bliskie do algorytmu MDA w wersji MMD. Typowy portret macierzy po takim uporządkowaniu jest podany na rys. 4.15.

5. Rozwiązywanie układów równań liniowych algebraicznych z macierzami rzadkimi symetrycznymi metodami bezpośrednimi.

Skuteczność rozwiązywania układów równań liniowych algebraicznych z macierzami rzadkimi metodami bezpośrednimi istotnie zależy od zdolności metody uporządkowania zmniejszania ilości niezerowych elementów w macierzy sfaktoryzowanej oraz zdolności solwera umieszczania w pamięci głównej dużych tablic danych, a także zabezpieczenia wirtualizacji (podział danych na bloki, z których tylko część znajduje się w pamięci głównej w tym samym czasie, natomiast inne bloki są pobierane z dysku) i zdolności solwera opracowywania bloków danych, znajdujących się w pamięci głównej, na wysokim poziomie wydajności.

Istnieje kilka typów solwerów bezpośrednich dla macierzy rzadkich.

5.1 Solwer skyline

Solwer skyline [George and Liu, 1981] realizuje metodę Gaussa lub Choleskiego, przy czym macierz jest przechowywana w formacie profilowym (rozdział 4.1). Dla osiągnięcia minimalnej szerokości profile przy uporządkowaniu jest stosowana metoda RCM (reverse Cuthill-McKee algorytm [George and Liu, 1981]) albo metoda Sloana [Sloan, 1986] – rozdział 4.2. Ten solwer był bardzo popularny do pierwszej połowy lat 90, ponieważ potrzebował niewielki rozmiar pamięci głównej. Dla dużych zadań ($N > 10\ 000$ równań) wydajność drastycznie spada w skutek powstania znacznej ilości zapełnień. Solwer jest prosty w realizacji, łatwo poddaje się wirtualizacji i potrafi rozwiązywać duże zadania na komputerach z małą pamięcią RAM. Prawdą jest, że czas rozwiązywania takiego zadania może trwać od kilkudziesięciu godzin do kilku dni.

5.2 Solwer frontalny

Solwer frontalny był opracowany przez B. Irons'a [Irons, 1970] dla rozwiązywania układów równań liniowych algebraicznych metody elementów skończonych (MES). Później powstała wersja algebraiczna [Scott, 2006]. Przedstawimy główną idee metody frontalnej, jako solwera MES, typową cechą,

której jest to, że każdy model obliczeniowy składa się z elementarnych podkonstrukcji – elementów skończonych, przy czym fizyczne zachowanie każdego elementu skończonego o numerze *e* jest opisane za pomocą gęstej macierzy $\mathbf{K}_e$. Elementy skończone łączą się pomiędzy sobą w węzłach modelu MES. Stąd wynika, że każdy element skończony zawiera listę węzłów.

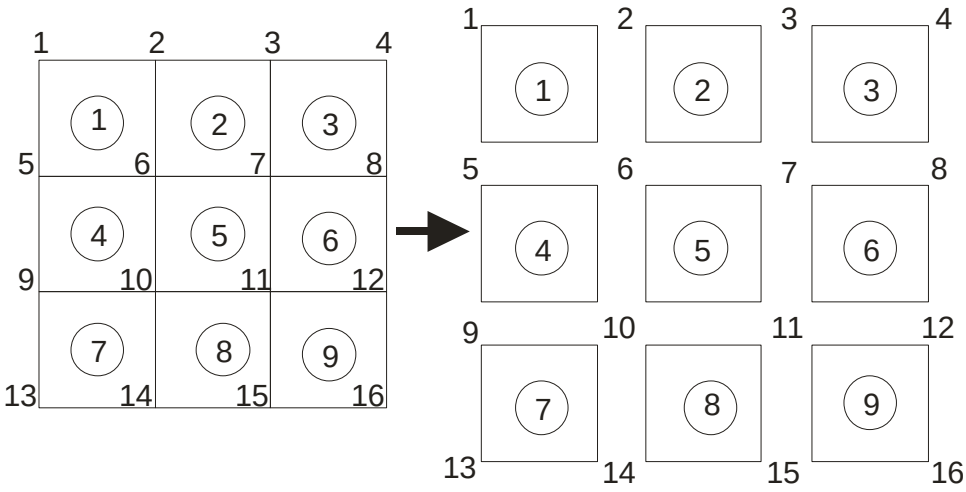
Jeśli połączyć ze sobą wszystkie elementy skończone danego modelu, powstanie globalna macierz MES. Proces łączenia elementów nazywa się agregacją, a globalna macierz jest przedstawiona jako

$$\mathbf{K} = \sum_{e=1}^{Nele} \mathbf{P}_e^T \mathbf{K}_e \mathbf{P}_e \quad , \tag{5.1}$$

gdzie *Nele* – ilość elementów skończonych w modelu MES, $\mathbf{P}_e$ – macierz permutacji o rozmiarze $N \times n_e$, N – rozmiar zadania, n_e – rozmiar macierzy bieżącego elementu skończonego. Operacja $\mathbf{P}_e^T \mathbf{K}_e \mathbf{P}_e$ przestawia elementy macierzy $\mathbf{K}_e$ w odpowiednie pozycje macierzy globalnej.

Klasycznym podejściem do rozwiązania zadania MES z punktu widzenia algebry liniowej jest agregowanie całej macierzy $\mathbf{K}$, jej faktoryzacja $\mathbf{K} = \mathbf{L}\mathbf{L}^T$ oraz wykonanie podstawień bezpośrednich i wstecznych $\mathbf{L}\mathbf{y} = \mathbf{b} \rightarrow \mathbf{y}$, $\mathbf{L}^T\mathbf{x} = \mathbf{y}$, gdzie $\mathbf{b}$ – wektor prawej strony, $\mathbf{x}$ – wektor rozwiązania.

W metodzie frontalnej MES macierz $\mathbf{K}$ nigdy nie agreguje się w całości. Zamiast tego przed rozwiązaniem model MES jest przedstawiony jako zbiór oddzielnych elementów skończonych (rys. 5.1). Poniżej są podane numery węzłów oraz w krążkach – numery elementów skończonych.



Rys. 5.1 Na pierwszym etapie konstrukcja jest przedstawiona jako zbiór oddzielnych elementów skończonych

Dalej zaczniemy pojedynczo dodawać elementy skończone do modelu. Bierzemy element 1. Jego macierz zawiera węzły 1, 2, 6, 5.

$$\mathbf{K}_1 = \begin{array}{c} \\ 1 \\ 2 \\ 6 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 6 & 5 \end{array} \\ \begin{pmatrix} k_{11}^1 & & & \\ k_{21}^1 & k_{22}^1 & & \\ k_{61}^1 & k_{62}^1 & k_{66}^1 & \\ k_{51}^1 & k_{52}^1 & k_{56}^1 & k_{55}^1 \end{pmatrix} \end{array} \tag{5.2}$$

Tu k_{ij}^1 – gęste podmacierze, ponieważ w MES każdy węzeł modelu obliczeniowego zwykłe zawiera tyle równań, ile stopni swobody on ma. Równania, które są związane z węzłem 1, będziemy nazywali kompletnie zebranymi, ponieważ w węźle 1 został połączony tylko element 1. Dodawanie jakichkolwiek pozostałych elementów nie będzie zmieniało współczynników tych równań. Kompletnie zebrane równania można od razu wyeliminować, nie oczekując na następne kroki faktoryzacji frontalnej. Macierze $\mathbf{K}_e$ są macierzami symetrycznymi, dlatego przechowujemy tylko dolną trójkątną część.

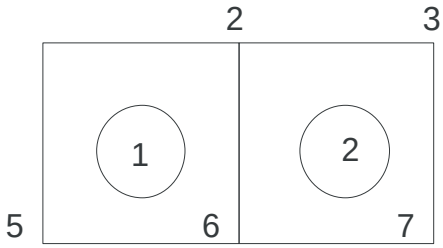
Zastosujemy blokową faktoryzacje Choleskiego (rozdział 2.2.2) dla faktoryzacji macierzy (5.2) w obrębie kompletnie zebranych równań. Dlatego sfaktoryzujemy diagonalny blok k_{11}^1, poprawimy bloki $k_{21}^1, k_{61}^1, k_{51}^1$, umieszczone pod diagonalą pierwszej kolumny blokowej, i obliczymy pozostałą część macierzy, używając obliczenie dopełnienia Schura. Kompletnie sfaktoryzowaną część macierzy – pierwszą kolumnę blokową – umieścimy w buforze dla przechowywania macierzy sfaktoryzowanej. Jest to część ostatecznego wyniku. Pozostała część macierzy, którą nazwiemy niezakończonym frontem, jest przesunięta o szerokość pasma kompletnie zebranych równań w lewo i do góry.

$$\begin{array}{c} \\ 2 \\ 6 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccc} 2 & 6 & 5 \end{array} \\ \begin{pmatrix} \tilde{k}_{22}^1 & & \\ \tilde{k}_{62}^1 & \tilde{k}_{66}^1 & \\ \tilde{k}_{52}^1 & \tilde{k}_{56}^1 & \tilde{k}_{55}^1 \end{pmatrix} \end{array} \tag{5.3}$$

Tilda nad symbolem oznacza, że elementy odpowiedniej podmacierzy uległy zmianie.

Dokładamy element 2, zawierający węzły 2, 3, 7, 6. Podstruktura, odpowiadająca bieżącemu etapowi agregacji, jest przedstawiona na rys. 5.2 i

zawiera węzły 2, 3, 7, 6, 5. Pominięto węzeł 1, ponieważ wszystkie równania dla niego zostały już wyeliminowane.



Rys. 5.2 Drugi etap agregowania

Macierz układu równań dla bieżącej podstruktury ma wygląd

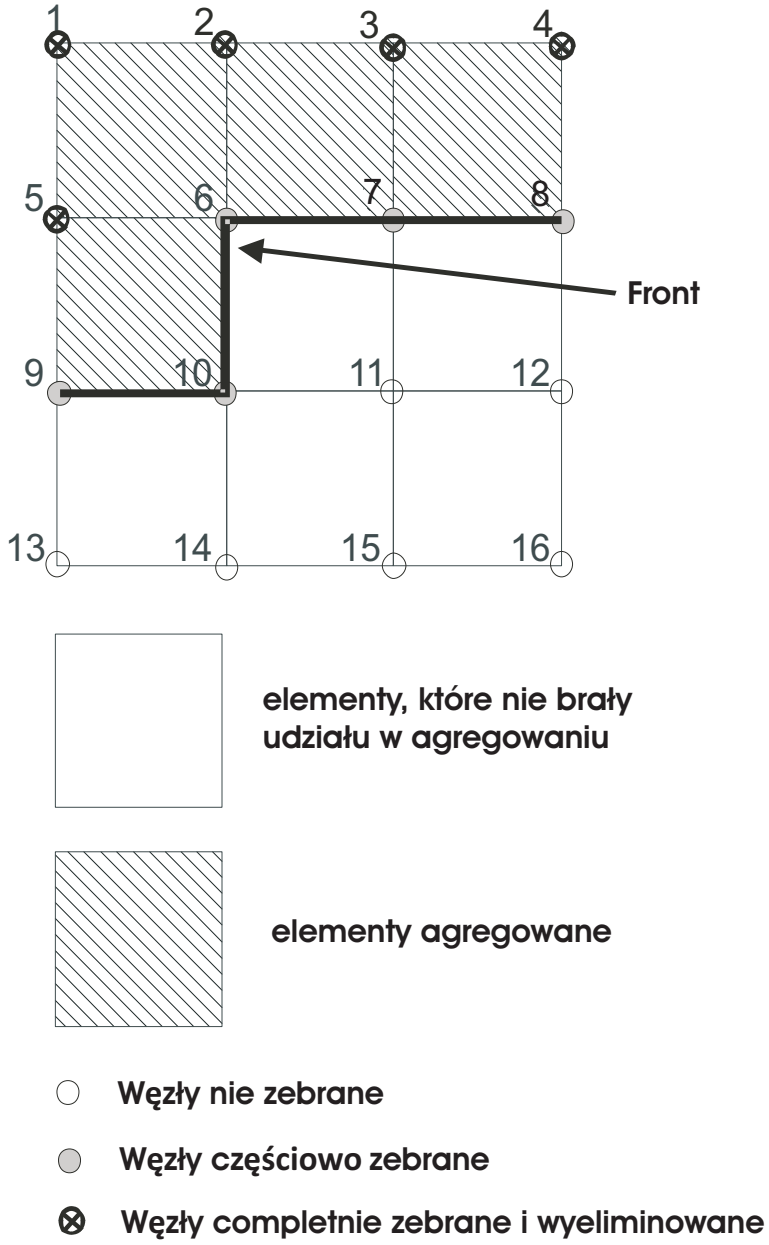
$$\begin{array}{c} \\ 2 \\ 6 \\ 5 \\ 3 \\ 7 \end{array} \begin{array}{c} \begin{array}{ccccc} 2 & 6 & 5 & 3 & 7 \end{array} \\ \begin{pmatrix} \tilde{k}^1_{22}+k^2_{22} & & & & \\ \tilde{k}^1_{62}+k^2_{62} & \tilde{k}^1_{66}+k^2_{66} & & & \\ \tilde{k}^1_{52} & \tilde{k}^1_{56} & \tilde{k}^1_{55} & & \\ k^2_{32} & k^2_{36} & 0 & k^2_{33} & \\ k^2_{72} & k^2_{76} & 0 & k^2_{73} & k^2_{77} \end{pmatrix} \end{array} \tag{5.4}$$

Węzeł 2 jest kompletnie zebrany. Stąd wynika, że kolumna blokowa dla węzła 2 też jest kompletnie zebrana. Wykonujemy odpowiedni krok blokowej faktoryzacji Choleskiego, kompletnie sfaktoryzowaną część macierzy umieszczamy w buforze dla sfaktoryzowanej macierzy, a pozostałe bloki w trakcie wykonania poprawienia przy obliczeniu dopełnienia Schura przesuwamy o szerokość pasma kompletnie zebranych równań w lewo i do góry.

Dalej kontynuujemy ten sam proces. Dokładamy kolejny element skończony, eliminujemy równania dla kompletnie zebranego węzła, umieszczamy kompletnie sfaktoryzowaną część macierzy do buforu i poprawiamy pozostałą część. Przesunięcie pozostałej części w lewo i do góry ogranicza wzrost rozmiaru macierzy gęstej, jaką będziemy nazywali macierzą frontalną.

Jeśli na modelu obliczeniowym wydzielić kompletnie zebrane (już wyeliminowane) węzły, węzły częściowo zebrane, które tworzą niezakończony front, oraz węzły, które jeszcze nie brały udziału w agregowaniu (rys. 5.3), przy czym węzły częściowo zebrane połączyć grubą linią, to w procesie jednoczesnego agregowania – eliminowania okazuje się, że ta linia porusza się po węzłach modelu

obliczeniowego. Powstaje wrażenie, że po konstrukcji propaguje front. Stąd pochodzi nazwa metody.

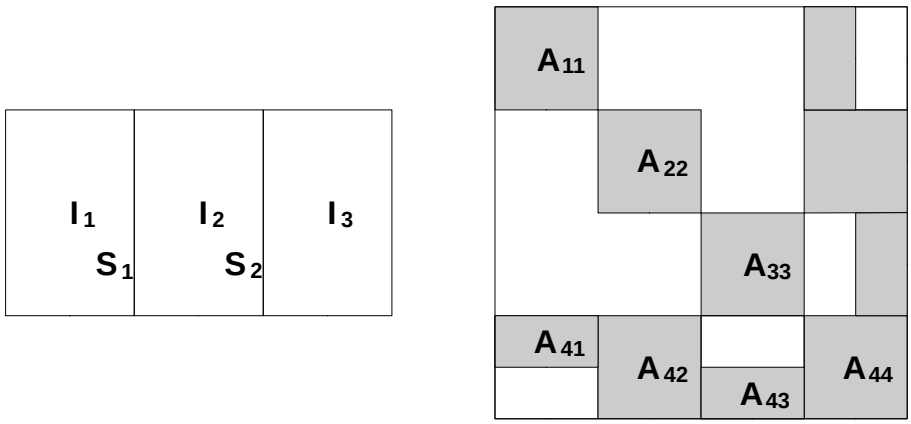


Rys. 5.3 Metoda frontalna

Równania dla węzłów częściowo zebranych tworzą niezakończony front i określają rozmiar macierzy frontalnej, dla zmniejszenia którego najczęściej używają algorytm RCM albo metodę Sloana – rozdział 4.2. Przy tym jest tworzony graf przyległości dla elementów skończonych modelu MES, ale nie dla węzłów, jak jest w metodzie skyline, czyli uporządkowujemy kolejność agregowania elementów skończonych.

5.3 Solwer domain decomposition

Dla przedstawienia głównej idei tej metody rozważmy obszar prostokątny (rys. 5.4).



Rys. 5.4. Podział węzłów na wewnętrzne i separatory dla obszaru prostokątnego. Solwer domain decomposition

Podzielimy ten obszar równoległymi przekrojami na kilka mniejszych podobszarów. Węzły wewnętrzne każdego podobszaru oznaczymy I_1, I_2, I_3. Węzły, łączące podobszary pomiędzy sobą, tworzą separatory S_1, S_2. Uporządkujemy równania w takiej kolejności, która odpowiada kolejności umieszczenia węzłów I_1, I_2, I_3, S_1, S_2. Wtedy macierz układu równań liniowych algebraicznych otrzyma kształt blokowo-diagonalny z obramowaniem (rys. 5.4). Bloki zerowe są oznaczone kolorem białym, a niezerowe – szarym. Zbiory węzłów wewnętrznych I_1, I_2, I_3 tworzą bloki diagonalne $\mathbf{A}_{11}$, $\mathbf{A}_{22}$, $\mathbf{A}_{33}$, ponieważ żaden z węzłów wewnętrznych jednego podobszaru nie jest związany z węzłem wewnętrznym innego podobszaru. Węzły-separatory S_1, S_2 tworzą blok diagonalny $\mathbf{A}_{44}$. Niezerowy blok $\mathbf{A}_{41}$ powstaje w skutek tego, że część węzłów wewnętrznych I_1 jest

związana z węzłami S_1 przez odpowiednie elementy skończone. Blok $\mathbf{A}_{42}$ bierze się z tego, że część węzłów wewnętrznych I_2 jest związana z węzłami S_1, S_2. Blok $\mathbf{A}_{43}$ powstaje w wyniku tego, że część węzłów wewnętrznych I_3 jest związana z węzłami S_2. W pamięci przechowujemy tylko dolną część trójkątną, ponieważ macierz jest symetryczna.

Jeśli równania dla węzłów każdego z bloków wewnętrznych uporządkować algorytmem minimalnego stopnia, to ilość zapełnień wewnątrz bloków diagonalnych istotnie się zmniejsza.

Zaletą takiej struktury macierzy jest to, że w blokach zerowych nie powstają zapełnienia.

Faktoryzacja Choleskiego dla macierzy o takiej strukturze jest przedstawiona wyrażeniem

$$\begin{pmatrix} \mathbf{A}_{11} & 0 & 0 & \mathbf{A}_{41}^T \\ 0 & \mathbf{A}_{22} & 0 & \mathbf{A}_{42}^T \\ 0 & 0 & \mathbf{A}_{33} & \mathbf{A}_{43}^T \\ \mathbf{A}_{41} & \mathbf{A}_{42} & \mathbf{A}_{43} & \mathbf{A}_{44} \end{pmatrix} = \begin{pmatrix} \mathbf{L}_{11} & 0 & 0 & 0 \\ 0 & \mathbf{L}_{22} & 0 & 0 \\ 0 & 0 & \mathbf{L}_{33} & 0 \\ \mathbf{L}_{41} & \mathbf{L}_{42} & \mathbf{L}_{43} & \mathbf{L}_{44} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{L}_{11}^T & 0 & 0 & \mathbf{L}_{41}^T \\ 0 & \mathbf{L}_{22}^T & 0 & \mathbf{L}_{42}^T \\ 0 & 0 & \mathbf{L}_{33}^T & \mathbf{L}_{43}^T \\ 0 & 0 & 0 & \mathbf{L}_{44}^T \end{pmatrix} \tag{5.5}$$

Każdy diagonalny blok sfaktoryzuje się niezależnie od innych:

$$\mathbf{A}_{ii} = \mathbf{L}_{ii} \cdot \mathbf{L}_{ii}^T \rightarrow \mathbf{L}_{ii}\ , \quad i = 1,2,...,n-1 \tag{5.6}$$

Tu n – ilość bloków diagonalnych (ilość podobszarów plus jeden). Po obliczeniu kolejnego bloku diagonalnego $\mathbf{L}_{ii}$ poprawiamy odpowiedni blok w obramowaniu

$$\mathbf{A}_{ni}^T = \mathbf{L}_{ii} \cdot \mathbf{L}_{ni}^T \rightarrow \mathbf{L}_{ni}\ , \quad i = 1,2,...,n-1 \tag{5.7}$$

Dlatego trzeba rozwiązać układ równań liniowych algebraicznych z macierzą dolną trójkątną $\mathbf{L}_{ii}$ i paczką prawych stron $\mathbf{A}_{ni}^T$. Dalej modyfikujemy ostatni blok diagonalny

$$\mathbf{A}_{nn} = \tilde{\mathbf{A}}_{nn} + \sum_{i=1}^{n-1} \mathbf{L}_{ni} \cdot \mathbf{L}_{ni}^T \rightarrow \tilde{\mathbf{A}}_{nn} = \mathbf{A}_{nn} - \sum_{i=1}^{n-1} \mathbf{L}_{ni} \cdot \mathbf{L}_{ni}^T\ . \tag{5.8}$$

Tu jest łatwo obliczyć dopełnienie Schura tuż po obliczeniu odpowiedniego bloku obramowania $\mathbf{L}_{ni}$:

$$\tilde{\mathbf{A}}_{nn}^{(1)} = \mathbf{A}_{nn} - \mathbf{L}_{n1} \cdot \mathbf{L}_{n1}^{T}; \quad \tilde{\mathbf{A}}_{nn}^{(2)} = \mathbf{A}_{nn}^{(1)} - \mathbf{L}_{n2} \cdot \mathbf{L}_{n2}^{T}; \ldots \tag{5.9}$$

W samym końcu sfaktoryzujemy ostatni diagonalny blok

$$\mathbf{A}_{nn} = \mathbf{L}_{nn} \cdot \mathbf{L}_{nn}^{T} \rightarrow \mathbf{L}_{nn} \tag{5.10}$$

Metoda domain decomposition jest prosta oraz łatwa do zrównoleglenia w dowolnej architekturze, jednak przedstawiony typ uporządkowania (metoda równoległych przekrojów [George and Liu, 1981] razem z algorytmem minimalnego stopnia wewnątrz każdego podobszaru) jest skuteczny tylko wtedy, gdy udaje się odnaleźć wąskie separatory. Dlatego ta metoda najczęściej ustępuje w wydajności metodom bardziej zaawansowanym (rozdział 4.2).

5.4 Dekompozycja Choleskiego looking-left

Metoda looking-left (rys. 5.5, rys. 5.6) polega na faktoryzacji kolumn macierzy rzadkiej z lewa na prawo.

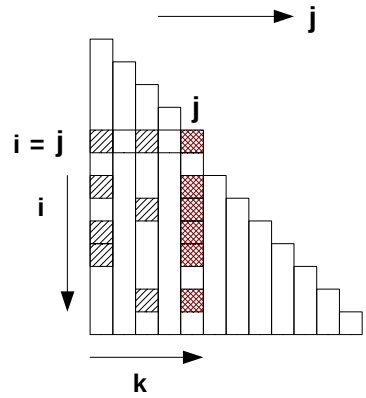
Na kroku faktoryzacji *j* z lewa od kolumny *j* znajdują się kompletnie sfaktoryzowane kolumny macierzy, a z prawa – kolumny macierzy źródłowej. Dla faktoryzacji kolumny *j* trzeba najpierw wykonać jej poprawienie kolumnami, umieszczonymi z lewa. Dla macierzy rzadkich w tym poprawieniu biorą udział tylko takie kolumny, które mają niezerowe elementy w wierszu $i = j$: *do* $k \in List[j]$, gdzie *List*[*j*] – lista kolumn, które poprawiają kolumnę *j*. Lista ta została utworzona w poprzednich krokach faktoryzacji.

Dla każdej kolumny *k* , biorącej udział w poprawieniu kolumny *j*, wykonujemy tą poprawę: *do* $i \in L_k$, gdzie L_k – niezerowa struktura kolumny *k*, przedstawiona formatem skompresowanym przy umieszczeniu elementów macierzy kolumna po kolumnie:

```
a_jk = Space[kfirst];
for(pos = kfirst+1; pos < Pos[k+1]; ++pos)
{
    i = ind[pos];
    a_ik = Space[pos];
    Temp[i] += a_ik*a_jk;
}
```

Temp $\leftarrow 0$; *set zeros to List*[j], $j = 1, 2, \ldots, N$
do $j = 1, N$
  // *update of column* j *by columns, which are located at left*
    do $k \in List[\, j\,]$
      do $i \in L_k$
        $Temp[i] \mathrel{+}= l_{jk} l_{ik}$
      end do
    end do
    do $i \in L_j$
      $\tilde{a}_{ij} = a_{ij} - Temp[i]$
      $Temp[i] \leftarrow 0$
    end do
    // *factorize of column* j
    $l_{jj} = \sqrt{\tilde{a}_{jj}}$
    do $i \in L_j$
      $l_{ij} = \dfrac{\tilde{a}_{ij}}{l_{jj}}$
      $Link[i] \leftarrow j$ // *add* j *to lists of right columns, which are updated by column* j
    end do
end do

Rys. 5.5. Algorytm looking-left faktoryzacji macierzy rzadkiej metodą Choleskiego



Rys. 5.6. Faktoryzacja looking-left macierzy rzadkiej

Tu *kfirst* wskazuje na element a_{jk} kolumny *k*, znajdujący się w wierszu $i = j$. Do pętli *for* będą pobierane tylko niezerowe elementy kolumny *k* ($i \in L_k$). W wektorze *Temp* (jest to tablica o rozmiarze *N*, gdzie *N* – ilość równań), zbieramy poprawki dla kolumny *j*. Po zakończeniu pętli *k* wektor *Temp* zawiera sumę poprawek od wszystkich kolumn $k \in List[j]$.

Przy wykonaniu pętli *do* $i \in L_j$ obliczamy poprawione elementy kolumny *j*. Dalej wykonujemy faktoryzacje kolumny *j*: obliczamy element diagonalny l_{jj} i dzielimy pozostałe niezerowe elementy, umieszczone poniżej diagonali, przez l_{jj}. Szczegóły algorytmu są podane w [George and Liu, 1981].

Dla macierzy o dużym rozmiarze ($N > 10\,000$) ponad 99% obliczeń wykonuje się przy obliczeniu elementów wektora poprawek *Temp*. W pętli wewnętrznej *do i* $\in L_k$ jest zrealizowany algorytm mnożenia wektora przez skalar, a ponieważ element l_{jk} nie zależy od indeksu iteracji *i* , to może być umieszczony w rejestrze procesora i ciągłe tam pozostawać.

Należy zwrócić uwagę na to, że technika macierzy rzadkich prowadzi do następujących optymalizacji:

- Przy obliczaniu wektora poprawek *Temp* pobieramy tylko kolumny, które mają niezerowe elementy l_{jk}.
- Przy obliczaniu elementów wektora poprawek *Temp*[*i*] w pętli po indeksie *i* uwzględniamy tylko niezerowe elementy kolumny *k* ($l_{ik} \neq 0$).

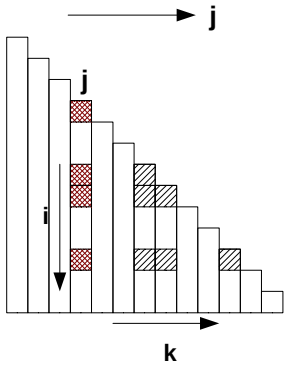
Wraz z możliwością zastosowania skutecznych metod uporządkowania powoduje to istotne zmniejszenie operacji arytmetycznych w stosunku do metody skyline oraz metody frontalnej.

5.5 Dekompozycja Choleskiego looking-right

Metoda ta jest przedstawiona na rys. 5.7 – 5.8. Sfaktoryzujemy kolumny z lewa na prawo – pętle *do* $j = 1, N$. Dla bieżącej kolumny *j* wykonujemy faktoryzację: pobieramy pierwiastek z elementu diagonalnego i dzielimy pozostałe niezerowe elementy przez nowy element diagonalny – pętle do $i \in L_j$. Z lewa od kolumny *j* znajdują się kompletnie sfaktoryzowane kolumny macierzy, a z prawa – kolumny, które będą poprawione przez kolumnę *j*. Przy wykonaniu tego poprawienia indeks *k* , gdzie $k > j$, przyjmuje wartości numerów wierszy elementów niezerowych kolumny *j*, ponieważ kolumna *j* będzie poprawiała tylko takie kolumny *k*, dla których $k = i$ pod warunkiem, że $l_{ij} \neq 0$ (rys. 5.8) – pętle do $k \in L_j \wedge k > j$. W każdej kolumnie $k \in L_j$ będą poprawione tylko takie elementy, które odpowiadają pozycjom niezerowych elementów kolumny *j* przy $i > k$ – pętle *do* $i \in L_j \wedge i \geq k$ – rys. 5.7, rys. 5.8.

```
do j = 1, N
  // factorize column j
  l_jj = sqrt(a_jj)
  piv = 1 / l_jj
  do i ∈ L_j
    l_ij = a_ij · piv
  end do
  // update of columns, which are located at right
  do k ∈ L_j ∧ k > j
    do i ∈ L_j ∧ i ≥ k
      a_ik = a_ik − l_ij · l_kj
    end do
  end do
end do
```

Rys. 5.7. Algorytm looking-right faktoryzacji macierzy rzadkiej metodą Choleskiego



Rys. 5.8. Faktoryzacja looking-right macierzy rzadkiej

Ponad 99% operacji arytmetycznych zostało wykonanych w trakcie poprawienia kolumną *j* kolumn, umieszczonych z prawa.

Ilość operacji arytmetycznych, wykonanych metodą looking right, jest dokładnie taka sama, jak dla metody looking left, jednak wydajność metody looking right jest istotnie niższa. Ten fakt polega na tym, że dla algorytmu looking

left w trakcie poprawienia kolumny *j* kolumnami umieszczonymi z lewa, wektor *Temp* jest utrzymywany w pamięci podręcznej nawet dla dość dużych zadań. Intensywny dostęp do wolnej pamięci głównej powstaje tylko przy odczycie elementów kolumny *k*, $k \in List[j]$.

Dla algorytmu looking right przy wykonaniu poprawienia kolumn, umieszczonych z prawa od kolumny *j*, intensywny dostęp do pamięci głównej wynika tak przy odczycie elementów kolumny *k*, $k \in L_j$, jak i przy ich zapisie po modyfikacji. Czyli ilość transferów danych pamięć główna – pamięć podręczna – pamięć główna dla algorytmu looking left jest około dwa razy mniejsza niż w przypadku algorytmu looking right, co w całości odpowiada wynikam, przedstawionym w [Mateev N,.Menon V., Pingali K., 2000] dla macierzy gęstych.

Metody looking left oraz looking right razem z efektywnym algorytmem uporządkowania bardzo skutecznie eliminują operacje nad elementami zerowymi. Jednak wszystkie operacje arytmetyczne należą do operacji wektorowo-skalarnych, które są wykonane na niskim poziomie wydajności. Drugą wadą tych metod jest zbyt duże obciążenie układu pamięci, co powoduje niskie przyspieszenie na komputerach wielordzeniowych.

W celu podniesienia wydajności oraz osiągnięcia wysokiego speed up przy wielowątkowości trzeba zastąpić operacje wektorowo-skalarne operacjami nad macierzami, które umożliwiają stosowanie algorytmów poziomu 3 BLAS i zmniejszają obciążenie układu pamięci. Takie metody używają blokową faktoryzację i są przedstawione bibliotekami wysokiej wydajności. Jedynym ograniczeniem funkcjonalności tych solwerów, a także klasycznych metod looking left i looking right, jest to, że są one w stanie pracować tylko z pamięcią główną. Jeśli rozmiar zadania przekracza możliwości RAM, to takie zadanie nie będzie rozwiązane. Jest to istotnym ograniczeniem na stosowanie tych solwerów w wielu zagadnieniach technicznych.

5.6 Solwer wielofrontalny

Taki solwer używa dowolnych metod uporządkowania, co daje możliwość istotnego zmniejszenia elementów niezerowych w macierzy sfaktoryzowanej, a także jednocześnie jest przydatny do wirtualizacji – wykorzystania pamięci dyskowej dla dużych zadań, które nie mogą być umieszczone w pamięci głównej. Do niedawna dla dużych zadań był to najbardziej używany solwer, ponieważ łączy on przewagi solwerów sparse direct (looking left, looking right) oraz możliwość wirtualizacji [Gould N. I. M., Hu Y., Scott J. A., 2005].

Główną idee [Amestoy P. R, Duff I. S, L'Excellent J-Y., 2000], [Dobrian F. and Pothen A., 2000] wyjaśnimy na przykładzie [Dobrian F., Kumfert G., Pothen A.,

2000]. Na rys. 5.9 są przedstawieni macierz rzadka po uporządkowaniu wybraną metodą oraz odpowiedni do niej graf przyległości (step 0).

Kroki 1 – 5 prezentują procedurę symbolicznej faktoryzacji, wykonanej w celu wyjaśnienia, w jakich pozycjach pojawią się zapełnienia. W ostatnim kroku powstaje dolna trójkątna macierz **L**. Niezerowe elementy macierzy źródłowej oznaczone jako "×", a zapełnienia – jako "●" .

Po wyjaśnieniu niezerowej struktury macierzy sfaktoryzowanej jest tworzone drzewo eliminacji (rys. 5.10), każdy wierzchołek, którego reprezentuje kolumnę macierzy rzadkiej, a każda krawędź – zależności pomiędzy kolumnami w trakcie faktoryzacji.

Tworzenie drzewa eliminacji polega na oglądaniu każdej kolumny macierzy z lewa na prawo. Dla bieżącej kolumny *j* znajdujemy pierwszy z góry, licząc od elementu diagonalnego, pozadiagonalny niezerowy element $a_{ij} \neq 0$. Nazwiemy jego rodzicem kolumny *j*. Oznacza to, że przy faktoryzacji macierzy kolumna *i* jest pierwszą kolumną, poprawioną przez kolumnę *j*. Na drzewie eliminacji powstaje krawędź $j - i$.

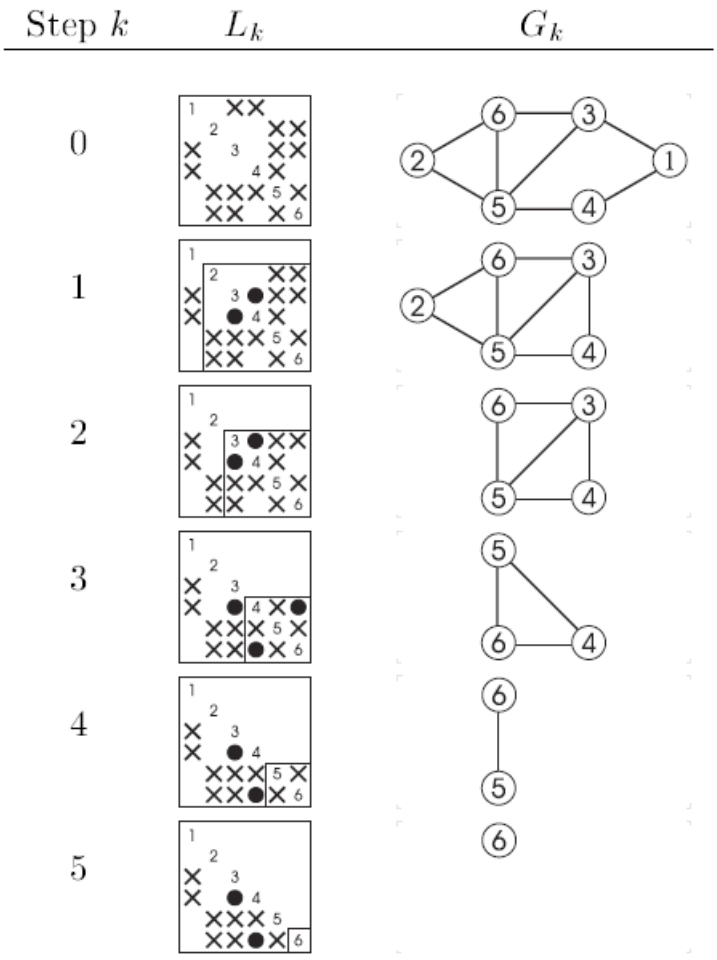
Dla podanego przykładu rodzicem pierwszej kolumny jest 3 (rys. 5.9, step 5). Na drzewie eliminacji (rys. 5.10) powstaje krawędź 1 – 3 . Rodzicem drugiej kolumny jest 5 – na drzewie eliminacji powstaje krawędź 2 – 5. Rodzicem trzeciej kolumny jest 4. Powstaje krawędź 3 – 4. Chociaż dla macierzy źródłowej ten element jest równy zeru, to po sfaktoryzowaniu trzeciej kolumny w tej pozycji powstaje zapełnienie. Dlatego konieczne jest użycie niezerowej struktury macierzy sfaktoryzowanej, a nie źródłowej.

I tak dalej.

Faktoryzacja macierzy kolumna po kolumnie prowadzi do skalarno-wektorowych operacji niskiej wydajności. Kluczowym momentem podniesienia wydajności jest łączenie kolumn macierzy w bloki. Dla macierzy rzadkiej dowolny podział na bloki powoduje obecność dużej ilości elementów zerowych w blokach i doprowadzi do istotnego zwiększenia ilości operacji arytmetycznych. Oznacza to, że trzeba odnaleźć taki podział macierzy rzadkiej na gęste prostokątne bloki, żeby ilość zerowych elementów w blokach była minimalna.

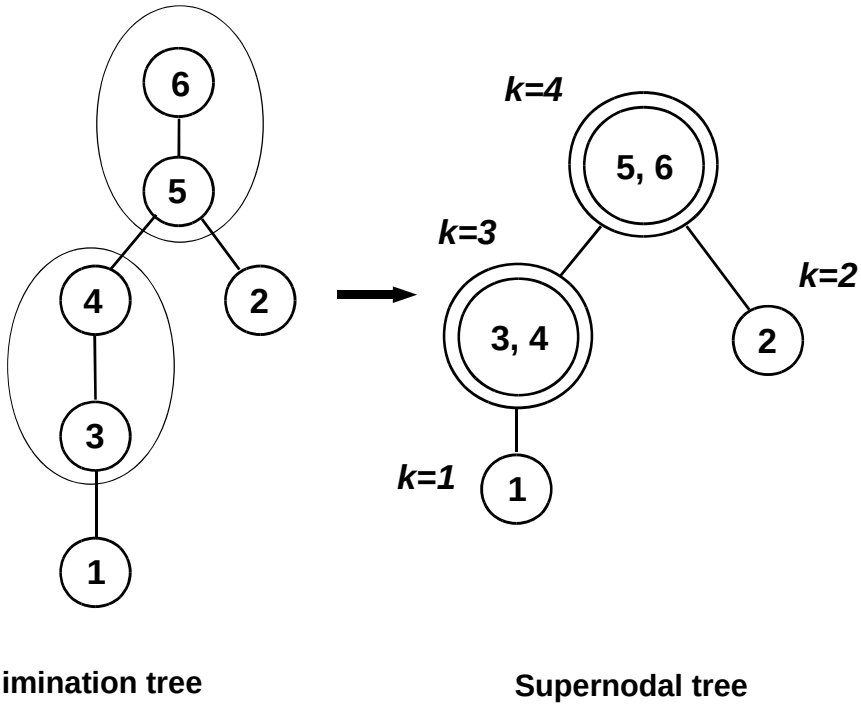
W tym celu stosuje się technikę superwęzłową, która polega na łączeniu wierzchołków drzewa eliminacji w superwęzły na podstawie następujących reguł:

- wierzchołki muszą należeć do tej samej ścieżki drzewa eliminacji oraz
- mieć sekwencyjne numerowanie i ten sam wierzchołek ze starszym numerem.



Rys. 5.9. Symboliczna faktoryzacja macierzy rzadkiej i odpowiednie przetwarzania na grafie przyległości [Dobrian F., Kumfert G., Pothen A., 2000], fig. 1.2

Na tej podstawie tworzymy drzewo superwęzłowe (rys. 5.10). Każdemu superwęzłowi drzewa eliminacji w macierzy rzadkiej odpowiada kolumna blokowa, która ma gęsty blok przy głównej diagonali. Dlatego każdemu niezerowemu elementowi każdej kolumny w bloku w ostatniej kolumnie tego bloku odpowiada element niezerowy, a każdemu zerowemu elementowi ostatniej kolumny w tym bloku odpowiada zerowy wiersz.



Rys. 5.10. Drzewo eliminacji i drzewo superwęzłowe. Nad wierzchołkami drzewa superwęzłowego jest oznaczona kolejność ich sfaktoryzowania [Dobrian F., Kumfert G., Pothen A., 2000]

Proces sfaktoryzowania macierzy rzadkiej został wykonany zgodnie z drzewem superwęzłowym. Zaczynamy z dolnego wierzchołku dowolnej gałęzi. Może to być wierzchołek 1 albo 2. Te wierzchołki można również sfaktoryzować równolegle, ponieważ pomiędzy wierzchołkami różnych gałęzi nie istnieje żadnej zależności.

W naszym przykładzie będziemy sfaktoryzowali wierzchołki w takiej kolejności: 1, 2, {3,4}, {5,6} [Dobrian F., Kumfert G., Pothen A., 2000]. Nie ma możliwości sfaktoryzowania superwęzłą {3,4}, dopóki nie sfaktoryzujemy kolumnę 1, ponieważ kolumna 1 poprawia kolumnę 3, która należy do superwęzła {3,4}. Tak samo nie możemy sfaktoryzować superwęzeł {5,6}, dopóki nie sfaktoryzujemy superwęzeł {3,4} i wierzchołek 2. Drzewo superwęzłowe definiuje kolejność faktoryzacji kolumn macierzy rzadkiej.

Dalej wykonujemy faktoryzację numeryczną (tab. 5.1).

Na kroku $k = 1$ sfaktoryzujemy wierzchołek 1 (rys. 5.10, z prawa). Tworzymy macierz frontalną $\mathbf{F}_1$ i wczytujemy niezerowe elementy pierwszej kolumny macierzy źródłowej. Przypisujemy numery niezerowych wierzy i kolumn – indeksy globalne.

Tabela 5.1

Etapy wykonania faktoryzacji numerycznej.

k	Asemblo-wanie	$\mathbf{F}_k$	$\mathbf{U}_k$	$\mathbf{C}_k$	$\mathbf{L}_k$
1	–	$\begin{array}{c\|ccc} & 1 & 3 & 4 \\ 1 & 1 & x & x \\ 3 & x & & \\ 4 & x & & \end{array}$	$\begin{array}{c\|cc} & 3 & 4 \\ 3 & 3 & \bullet \\ 4 & \bullet & 4 \end{array}$	$\begin{array}{c\|c} & 1 \\ 1 & 1 \\ 3 & x \\ 4 & x \end{array}$	$\begin{pmatrix} 1 & & \\ & & \\ x & & \\ x & & \end{pmatrix}$
2	–	$\begin{array}{c\|ccc} & 2 & 5 & 6 \\ 2 & 2 & x & x \\ 5 & x & & \\ 6 & x & & \end{array}$	$\begin{array}{c\|cc} & 5 & 6 \\ 5 & 5 & \bullet \\ 6 & \bullet & 6 \end{array}$	$\begin{array}{c\|c} & 2 \\ 2 & 2 \\ 5 & x \\ 6 & x \end{array}$	$\begin{pmatrix} 1 & & & \\ & 2 & & \\ x & & & \\ x & & & \\ & x & & \\ & x & & \end{pmatrix}$
3	$+\mathbf{U}_1$	$\begin{array}{c\|cccc} & 3 & 4 & 5 & 6 \\ 3 & 3 & \bullet & x & x \\ 4 & \bullet & 4 & x & \\ 5 & x & x & & \\ 6 & x & & & \end{array}$	$\begin{array}{c\|cc} & 5 & 6 \\ 5 & 5 & \bullet \\ 6 & \bullet & 6 \end{array}$	$\begin{array}{c\|cc} & 3 & 4 \\ 3 & 3 & \\ 4 & \bullet & 4 \\ 5 & x & x \\ 6 & x & \bullet \end{array}$	$\begin{pmatrix} 1 & & & & \\ & 2 & & & \\ x & & 3 & & \\ x & & \bullet & 4 & \\ & x & x & x & \\ & x & x & \bullet & \end{pmatrix}$
4	$+\mathbf{U}_2+\mathbf{U}_3$	$\begin{array}{c\|cc} & 5 & 6 \\ 5 & 5 & x \\ 6 & x & 6 \end{array}$	–	$\begin{array}{c\|cc} & 5 & 6 \\ 5 & 5 & \\ 6 & x & 6 \end{array}$	$\begin{pmatrix} 1 & & & & & \\ & 2 & & & & \\ x & & 3 & & & \\ x & & \bullet & 4 & & \\ & x & x & x & 5 & \\ & x & x & \bullet & x & 6 \end{pmatrix}$

W postaci ogólnej faktoryzacja macierzy frontalnej wygląda następująco:

$$\begin{pmatrix} \mathbf{A}_k & \mathbf{W}_k^T \\ \mathbf{W}_k & \mathbf{B}_k \end{pmatrix} = \begin{pmatrix} \mathbf{L}_k & 0 \\ \tilde{\mathbf{W}}_k & \mathbf{U}_k \end{pmatrix}_k \cdot \begin{pmatrix} \mathbf{L}_k^T & \tilde{\mathbf{W}}_k^T \\ 0 & \mathbf{I} \end{pmatrix}, \tag{5.11}$$

gdzie k – krok faktoryzacji, $\mathbf{A}_k$ – blok diagonalny, $\mathbf{W}_k$ – blok, umieszczony pod diagonalą. Macierz frontalna jest kompletnie zebrana tylko w obrębie pierwszej kolumny blokowej. Blok diagonalny z indeksami $2, 2$ w macierzy frontalnej źródłowej przedstawiamy jako zerową podmacierz. Jednak po dodawaniu odpowiednich macierzy poprawek $\mathbf{U}_k$, które powstały na poprzednich krokach faktoryzacji, może pojawić się niezerowy blok $\mathbf{B}_k$.

W macierzy frontalnej można sfaktoryzować tylko kompletnie zebrane równania, czyli pierwszą kolumnę blokową. Przy tym w górnym bloku diagonalnym powstaje dolna trójkątna macierz $\mathbf{L}_k$, elementy macierzy $\mathbf{W}_k$ ulegną zmianie, a na miejscu bloku $\mathbf{B}_k$ macierzy źródłowej pojawi się macierz poprawek $\mathbf{U}_k$. Etapy faktoryzacji macierzy frontalnej są przedstawione poniżej:

$$\begin{aligned} &1.\quad \mathbf{A}_k = \mathbf{L}_k \cdot \mathbf{L}_k^T \\ &2.\quad \mathbf{L}_k \cdot \tilde{\mathbf{W}}_k^T = \mathbf{W}_k^T \rightarrow \tilde{\mathbf{W}}_k^T \ . \\ &3.\quad \mathbf{U}_k = \mathbf{B}_k - \tilde{\mathbf{W}}_k \cdot \tilde{\mathbf{W}}_k^T \end{aligned} \tag{5.12}$$

Jest to blokowa faktoryzacja Choleskiego – rozdział 2.2.2. Dla dowolnego kroku k kompletnie sfaktoryzowana kolumna blokowa ma wygląd:

$$\mathbf{C}_k = \begin{pmatrix} \mathbf{L} \\ \tilde{\mathbf{W}} \end{pmatrix}_k , \quad k = 1, 2, 3, 4 \ . \tag{5.13}$$

Macierz $\mathbf{C}_k$ zawiera kompletnie sfaktoryzowane kolumny macierzy frontalnej. Jest to część wyniku ostatecznego – sfaktoryzowanej macierzy globalnej. Macierz $\mathbf{C}_k$ jest macierzą gęstą, ponieważ nie zawiera wierszy zerowych. Dla umieszczenia odpowiednich elementów macierzy $\mathbf{C}_k$ w pozycji macierzy globalnej $\mathbf{L}_k$ są używane indeksy globalne.

Przy zastosowaniu (5.12) do macierzy frontalnej $\mathbf{F}_1$, otrzymujemy macierz $\mathbf{C}_1$ i macierz poprawek $\mathbf{U}_1$ (tab. 5.1).

Na kroku $k = 2$ wczytujemy drugą kolumnę macierzy źródłowej, pomijając zerowe elementy. Dostajemy macierz frontalną $\mathbf{F}_2$, po sfaktoryzowaniu, której powstają macierze $\mathbf{C}_2$ i $\mathbf{U}_2$.

Na kroku $k = 3$ sfaktoryzujemy superwęzeł $\{3, 4\}$. Wczytujemy kolumny 3, 4 macierzy źródłowej, które tworzą kolumnę blokową. Zerowe wiersze kolumny blokowej (podany przykład nie zawiera takich wierszy) muszą być pominięte. Powstaje macierz frontalna $\mathbf{F}_{3,4}$, którą trzeba poprawić przy dodawaniu do macierzy poprawek $\mathbf{U}_1$, ponieważ superwęzeł $\{2,3\}$ ma poprzednika 1 (rys. 5.10). Przy takim dodawaniu elementy z jednakowymi indeksami globalnymi sumują się algebraicznie. Po sfaktoryzowaniu poprawionej macierzy frontalnej dostajemy macierzy $\mathbf{C}_3$ i $\mathbf{U}_3$.

Na kroku $k = 4$ wczytujemy kolumny 5, 6 macierzy źródłowej i dodajemy macierzy poprawek $\mathbf{U}_2$, $\mathbf{U}_3$, ponieważ superwęzeł {3, 4} i wierzchołek 2 są poprzednikami superwęzła {5, 6}. Na ostatnim kroku macierz frontalna jest macierzą kompletnie zebraną, dlatego nie powstaje macierz poprawek.

W metodach frontalnej i wielofrontalnej globalna macierz dolna trójkątna **L** fizycznie nie istnieje. Wynik faktoryzacji jest przechowywany w sekwencji macierzy gęstych $\mathbf{C}_1 - \mathbf{C}_4$, które są używane przy wykonywaniu podstawień bezpośrednich i wstecznych.

Przedstawiony algorytm jest łatwy do wirtualizacji. W pamięci głównej trzeba umieścić największą macierz frontalną razem z tablicą indeksów globalnych oraz dwa bufory, jeden z których powinien przechowywać największą macierz $\mathbf{C}_k$, k = 1, 2, ... , a drugi – wektor o rozmiarze N, gdzie N – rozmiar zadania. Przed faktoryzacją macierzy frontalnej kompletnie zebrana kolumna blokowa kopiuje się do pierwszego buforu i w tych adresach pamięci odbywa się jej faktoryzacja. Po faktoryzacji ten bufor zapisuje się na dysk do pliku f1, a w macierzy frontalnej pozostaje tylko macierz poprawek $\mathbf{U}_k$, k = 1, 2, ... , która w przypadku deficytu pamięci RAM może być też wyładowana na dysk do pliku f2. Przy agregacji następnej macierzy frontalnej trzeba odczytać z pliku f2 odpowiednie macierze poprawek. Dlatego jest potrzebny drugi bufor. Dla dużych zadań tablicy indeksów globalnych też mogą być zapisane na dysk w plik f3, a przy wykonaniu podstawień bezpośrednich i wstecznych – odczytane z dysku. Macierz źródłowa również może być zapisana na dysku (plik f0) i odczytywana po częściach do drugiego buforu przed umieszczeniem jej odpowiednich kolumn do macierzy frontalnej.

Sfaktoryzowana macierz znajduje się w pliku f1.

Dla małych i średnich zadań wszystkie dane są umieszczone w pamięci głównej.

Zrobimy krótkie podsumowanie. Rozwiązywanie układu równań liniowych algebraicznych z macierzą rzadką metodą wielofrontalną odbywa się w następującej kolejności:

a) Tworzymy graf przyległości dla podanej macierzy rzadkiej.
b) Uporządkowujemy ten graf wybraną metodą uporządkowania.
c) Na podstawie faktoryzacji symbolicznej dostajemy faktor-graf oraz niezerową strukturę macierzy sfaktoryzowanej, reprezentowanej formatem skompresowanym – tablicy *ind*, *Pos* (tab. 4.4). Tu *ind* – tablica pierwszych indeksów niezerowych elementów pozadiagonalnych, ponieważ na różnice od tab. 4.4 w naszym przypadku macierz jest umieszczona kolumna po kolumnie, a nie wiersz po wierszu.
d) Na podstawie niezerowej struktury macierzy rzadkiej sfaktoryzowanej tworzymy drzewo eliminacji i drzewo superwęzłowe.
e) Wykonujemy faktoryzację numeryczną.
f) Wykonujemy podstawienia bezpośrednie i wsteczne.

Procedury a – d tworzą etap analizy macierzy rzadkiej.

Przedstawiony powyżej solwer wielofrontalny jest metodą algebraiczną, ponieważ jedyną wejściową informacją jest macierz źródłowa, podana w formacie skompresowanym. W metodzie elementów skończonych (MES) często jest stosowany solwer wielofrontalny MES, dla którego wejściową informacją są graf przyległości węzłów modelu MES, reprezentujący topologie modelu obliczeniowego oraz macierze sztywności każdego elementu skończonego, najczęściej przedstawiane w postaci procedury, wywołanie której powoduje tworzenie macierzy sztywności dla podanego elementu skończonego, a także listy węzłów dla każdego elementu skończonego. Takie solwery mają bardziej wąskie zastosowanie, jednak przy odpowiednim opracowaniu wymagają mniejszej objętości pamięci RAM oraz są bardziej wydajne w stosunku do solwerów algebraicznych.

Rozważmy jeden z takich solwerów – blokową wielofrontalną metodę podkonstrukcji BSMFM (block substructure multifrontal method) [Fialko S., 2004], [Fialko S., 2008], [Fialko S., 2009, CT], [Fialko S., 2009, M], [Fialko S., 2009, CAMES].

5.7 Blokowa wielofrontalna metoda podkonstrukcji

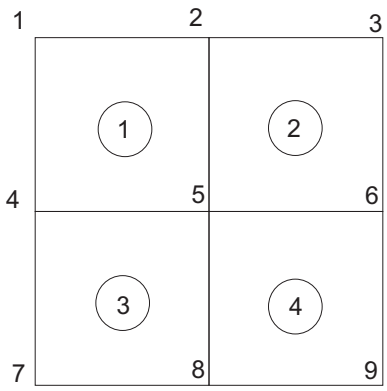
Stosowanie grafu przyległości dla węzłów modeli MES zamiast grafu przyległości dla macierzy rzadkiej ma następujące zalety:

- Istotnie zmniejsza objętość danych, ponieważ graf przyległości dla węzłów modeli MES zawiera mniej wierzchołków i krawędzi od grafu dla macierzy rzadkiej.
- W skutek tego szybciej działają metody uporządkowania, co jest istotne dla dużych zadań (ponad 1 000 000 równań).
- Najczęściej metody uporządkowania działają skuteczniej na grafach zgrubionych, czyli stosowanie grafu przyległości dla węzłów doprowadza do mniejszej ilości zapełnień niż w przypadku stosowania grafu dla macierzy rzadkiej.
- Powstaje łączenie równań w grupy, wynikające z fizycznego ustawienia problemu, ponieważ w każdym węźle modeli MES istnieje kilka równań. Przy uporządkowaniu oddzielnych równań te grupy mogą być rozłączone, a przy uporządkowaniu węzłów modeli MES grupy pozostają. Powoduje to łatwiejsze skupienie równań w duże bloki.

Podstawą metody jest eliminacja węzłów modeli MES. Wyeliminowanie węzła oznacza wyeliminowanie wszystkich równań, skojarzonych z tym węzłem. Procedura faktoryzacji polega na wyeliminowaniu węzłów krok po kroku. Ilość kroków jest równa ilości węzłów. Jeśli w podanym węźle brakuje równań w skutek

zadania podpór, bieżący krok eliminacji jest pusty. Jeśli istnieje możliwość łączenia grup w większe bloki, to za jednym razem będzie wykonano tyle kroków eliminacji, ile węzłów zostało zjednanych w podany blok.

Rozważmy prosty przykład – płyta kwadratowa z siatką 2×2 (rys. 5.11).



Rys. 5.11. Płyta kwadratowa z siatką 2×2

Zakładamy, że uporządkowanie grafu przyległości węzłów modeli MES jest wykonane na podstawie algorytmu minimalnego stopnia z tablicą permutacji *Perm* = {1, 3, 7, 9, 2, 6, 8, 4, 5}.

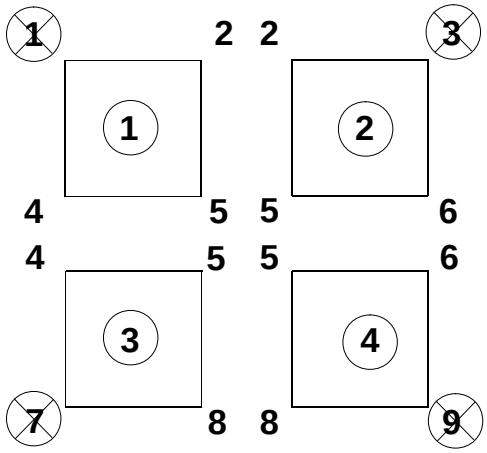
Rozdzielimy modele MES na oddzielne elementy skończone (rys. 5.12). Uzyskujemy taką kolejność agregowania elementów skończonych, żeby zabezpieczyć podaną kolejność eliminowania węzłów (tab. 5.2). Dlatego, żeby wyeliminować węzeł, trzeba połączyć wszystkie elementy skończone, zawierające ten węzeł. Wtedy węzeł będzie kompletnie zebrany.

Tabela 5.2

Uzyskanie kolejności agregowania elementów skończonych

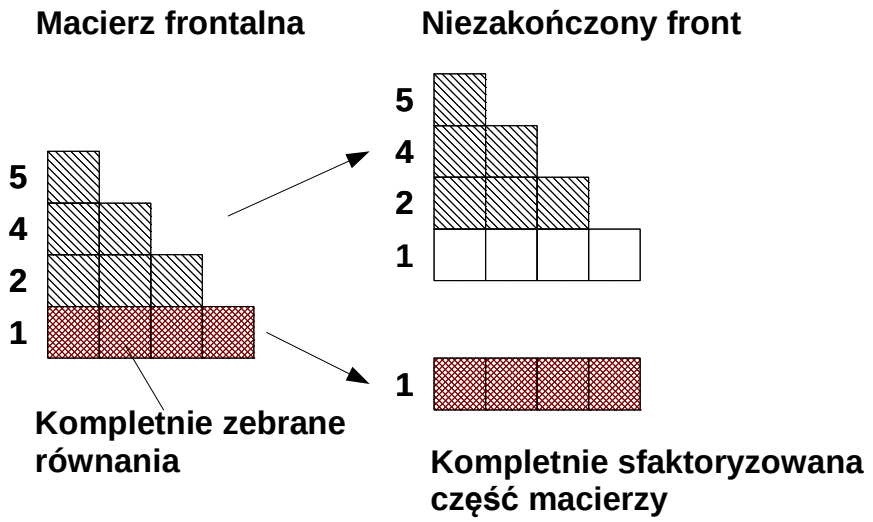
Numer węzła	1	3	7	9	2	6	8	4	5
Numer elementu	1	2	3	4	1, 2	2, 4	3, 4	1, 3	1,2,3,4

Każdy element może być agregowany tylko jeden raz. Elementy, które już byli agregowane wcześniej, oznaczamy w kolorze szarym. Na przykład, żeby zebrać węzeł 2, trzeba agregować elementy 1, 2, przecież te elementy już byli agregowane.



Rys. 5.12. Płyta kwadratowa z siatką 2×2. Wyeliminowane węzły są przekreślone

Węzły 1, 3, 9, 7 są kompletnie zebrane już na poziomie oddzielnych elementów skończonych. Dla wyeliminowania węzła 1 rozważmy macierz elementu 1. Jest to macierz frontalna, w której grupa równań dla węzła 1 jest kompletnie zebrana (rys. 5.13).



Rys. 5.13. Macierz frontalna dla pierwszego kroku eliminacji. Eliminujemy węzeł 1

Przy agregowaniu macierzy frontalnej umieszczamy węzły w kolejności, odwrotnej do podanej w tablice *Perm*. Wtedy grupa kompletnie zebranych równań

zawsze okazuje się w dolnej części macierzy frontalnej. Kopiujemy tą część macierzy do buforu dla kompletnie zebranych równań i faktoryzujemy w adresach tego bufora (fully decomposed part). Niezakończony front (incomplete front) jest obliczany na podstawie dopełnienia Schura. Jeśli grupa kompletnie zebranych równań jest umieszczona w dolnej części macierzy frontalnej, nie trzeba wykonywać przesunięcia elementów niezakończonego frontu w górną część macierzy. Jest to istotne dla podniesienia wydajności i speed up przy wielowątkowych obliczeniach, ponieważ dla komputerów z pamięcią wspólną procedura kopiowania należy do procedur o niskiej wydajności, które w trybie wielowątkowym są przyspieszane słabo.

Faktoryzacja macierzy frontalnej wygląda następująco:

$$\begin{pmatrix} \mathbf{C} & \mathbf{W} \\ \mathbf{W}^T & \mathbf{D} \end{pmatrix} = \begin{pmatrix} \tilde{\mathbf{C}} & \tilde{\mathbf{W}} \\ 0 & \mathbf{L} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{I} & 0 \\ 0 & \mathbf{I}_s \end{pmatrix} \cdot \begin{pmatrix} \mathbf{I} & 0 \\ \tilde{\mathbf{W}}^T & \mathbf{L}^T \end{pmatrix}, \tag{5.14}$$

gdzie $\mathbf{D}$ – diagonalny blok kompletnie zebranych równań, $\begin{pmatrix} \mathbf{W}^T & \mathbf{D} \end{pmatrix}$ tworzy dolny pasek kompletnie zebranych równań, $\mathbf{L}$ – dolna trójkątna macierz po sfaktoryzowaniu diagonalnego bloku $\mathbf{D}$, $\tilde{\mathbf{C}}$ – niezakończony front, $\mathbf{I}_s$ – diagonal znaków, która daje możliwość sfaktoryzowania macierzy nieokreślonych. Przy zastosowaniu blokowej faktoryzacji, otrzymujemy:

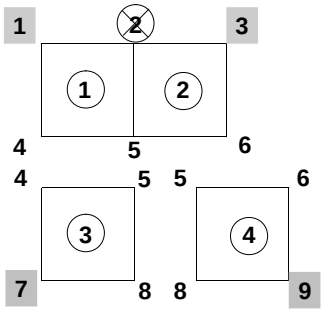
$$\begin{aligned} &\mathbf{D} = \mathbf{L} \cdot \mathbf{I}_S \cdot \mathbf{L}^T \quad \rightarrow \quad \mathbf{L}, \mathbf{I}_S \\ &\mathbf{W}^T = \mathbf{L} \cdot \mathbf{I}_S \cdot \tilde{\mathbf{W}}^T \rightarrow \quad \tilde{\mathbf{W}} \\ &\mathbf{C} = \tilde{\mathbf{C}} + \tilde{\mathbf{W}} \cdot \mathbf{I}_S \cdot \tilde{\mathbf{W}}^T \rightarrow \tilde{\mathbf{C}} = \mathbf{C} - \tilde{\mathbf{W}} \cdot \mathbf{I}_S \cdot \tilde{\mathbf{W}}^T \end{aligned} \tag{5.15}$$

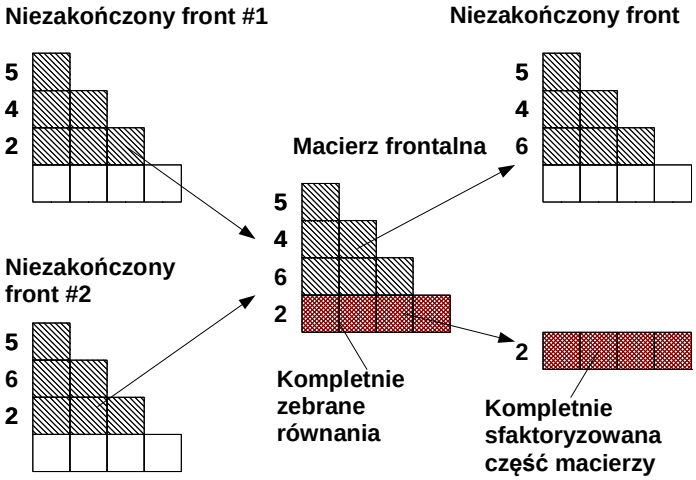
W taki sam sposób będą wyeliminowane węzły 3, 7, 9 na poziomie oddzielnych elementów skończonych 2, 3, 4.

Dla eliminacji węzła 2 trzeba zagregować elementy skończone 1, 2 (rys. 5.14). W związku z tym, że elementy te już były agregowane wcześniej, agregujemy niezakończone fronty, zawierające węzeł 2. Itd.

Proces agregowania – eliminowania węzłów (faktoryzacje macierzy) jest przedstawiony w postaci struktury danych *Deskryptor Procesu* (tab. 5.3).

Front należy rozumieć jako obiekt klasy C++, który zawiera numer eliminowanego węzła, listę węzłów, tworzących front, listę frontów poprzednich oraz listę agregowanych elementów skończonych. Każdy front jest przedstawiony wierszem tabeli 5.3. Eliminowany węzeł zawsze znajduje się na ostatnim miejscu listy węzłów. Każdy z frontów poprzednich może być użyty do agregacji tylko jeden raz.





Rys. 5.14. Agregowanie niezakończonych frontów 1, 2. Węzeł 2 jest kompletnie zebrany. Węzły 1, 3, 9, 7 są wyeliminowane na poprzednich krokach

Tabela 5.3

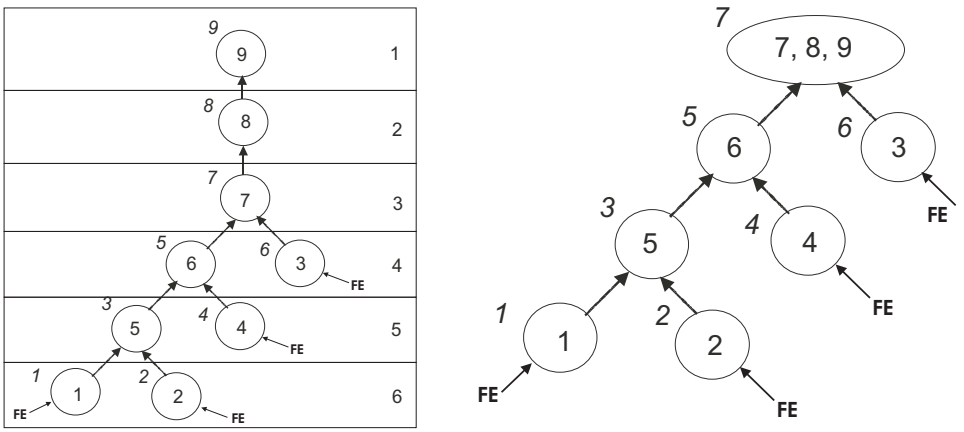
Deskryptor procesu

Krok eliminacji	Eliminowany węzeł	Lista węzłów, tworzących macierz frontalną	Lista poprzednich frontów	Lista agregowanych elementów
1	1	5, 4, 2, **1**	-----	1
2	3	5, 6, 2, **3**	-----	2
3	7	5, 4, 8 , **7**	-----	3
4	9	5, 8, 6, **9**	-----	4
5	2	5, 4, 6, **2**	1,2	----
6	6	5, 4, 8, **6**	5,4	----
7	8	5, 4, **8**	6,3	----
8	4	5, **4**	7	----
9	5	**5**	8	----

Strukturę poziomów drzewa frontalnego tworzymy poprzez oglądanie tab. 5.3 z dołu do góry. Ostatnim okazuje się front 9 – ustawiamy jego na górę, jako poziom 1. Poprzednikiem frontu 9 jest front 8, który trafi na drugi poziom. Poprzednikiem frontu 8 jest front 7 – ustawiamy jego na poziom 3. Poprzednikami frontu 7 są fronty 6 i 3 – ustawiamy ich na poziom 4. Itd.

W taki sposób powstaje drzewo frontalne (rys. 5.15, z lewa). Każdy wierzchołek tego drzewa odpowiada pewnej podstrukturze, która ma swój zbiór węzłów niewyeliminowanych oraz swoją macierz frontalną. Krawędzi tego drzewa ustalają kolejność eliminacji frontów. Przy agregowaniu frontów 1 – 4 są używane elementy skończone 1 – 4. Na rysunku oznaczamy to symbolem FE ze strzałką.

Żeby zmniejszyć rozmiar danych dla przechowywania niezakończonych frontów wprowadzamy renumeracje drzewa frontalnego tak, jak jest to podane na rys. 5.15. Nowe numery są oznaczone nad krążkami. W takiej kolejności będziemy faktoryzowali odpowiednie macierze frontalne.



Rys. 5.15. Struktura poziomów drzewa frontalnego (z lewa) i drzewa scalonego (z prawa)

Na drzewie frontalnym wydzielamy wierzchołki początkowe, sekwencyjne i węzłowe. Wierzchołki sekwencyjne mają tylko jednego poprzednika, węzłowe – kilka poprzedników, a początkowe nie mają frontów poprzednich. Pamięć dla macierzy frontalnej będzie alokowana tylko dla wierzchołków początkowych oraz węzłowych, a także będzie agregowana odpowiednia macierz frontalna. Dla wierzchołków sekwencyjnych przekazujemy wskaźnik do macierzy frontalnej od jego poprzednika. Daje to możliwość uniknięcia operacji kopiowania danych z jednego frontu do innego. Jeżeli na tym kroku istnieją agregowane elementy skończone, dodajemy algebraicznie elementy macierzy tych elementów skończonych do elementów macierzy frontalnej, które mają takie same indeksy globalne. Jeśli dla frontów sekwencyjnych na podanym kroku żaden element

skończony nie będzie agregowany, oznacza to, że w macierzy frontalnej są kompletnie zebrane równania z poprzednich kroków. To daje możliwość łączyć grupy kompletnie zebranych równań frontów sekwencyjnych w większe bloki w celu podniesienia wydajności. W naszym przykładzie fronty 7, 8, 9 są frontami sekwencyjnymi, niepobierającymi żadnych elementów skończonych. Stąd powstaje scalone drzewo frontalne z superwierzchołkiem {7, 8, 9} (rys. 5.15, z prawa).

Numeryczna faktoryzacja jest wykonana w kolejności, odpowiadającej numerowaniu wierzchołków drzewa frontalnego scalonego po przenumerowaniu. Faktoryzacja każdej macierzy frontalnej odbywa się zgodnie z (5.15). Obliczenie dopełnienia Schura oraz zrównoleglenie na podstawie OpenMP jest podane w rozdziale 3.6. Mnożenie diagonali znaków $\mathbf{I}_s$ przez blok $\tilde{\mathbf{W}}^T$ (5.15) jest wykonane przy przepakowaniu bloku macierzy $\hat{\mathbf{W}}^T = \mathbf{I}_s \cdot \tilde{\mathbf{W}}^T$ (rys. 3.23).

Jako przykład, rozważmy sześciościan, podzielony siatką 50×50×50 na objętościowe elementy skończone. Rozmiar zadania wynosi 397 941 równań. W tab. 5.4 w celu oszacowania wniosków różnych technik przyspieszenia są podane czasy rozwiązywania dla różnych wersji tej metody.

Tabela 5.4

Trwałość faktoryzacji numerycznej, s.

Wersja, rok	Platforma	Ilość procesorów (rdzeni)		
		1	2	4
2002	ia32	4 520	–	–
2004	ia32	2 000	–	–
2008	ia32	2 000	1 142	684
2009	ia32	827	504	365
2009	×64	584	322	213
ANSYS v.11.0	ia32	1 610	882	544

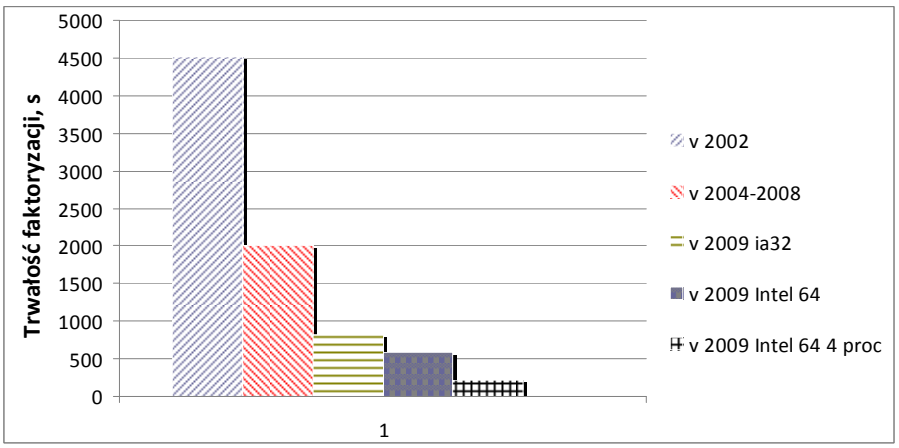
W wersji z 2002 roku przy faktoryzowaniu macierzy frontalnej była stosowana **LU** faktoryzacja Gaussa przy umieszczeniu elementów macierzy frontalnej wiersz po wierszu w górnej części trójkątnej [Fialko S., 2004]. Rejestry XMM nie są używane, a także brakuje wspierania wielowątkowości, blokowania pamięci podręcznej oraz blokowania rejestrów (metoda naiwna). Ta wersja była zaimplementowana do programu Robot Millenium v. 15 (www.robobat.com) oraz do programu SCAD v. 31 (www.scadsoft.com) [Карпиловский В. С. и др., 2004]. Są to oprogramowania MES dla obliczeń konstrukcji budowlanych.

W wersji z 2004 roku dla faktoryzacji macierzy frontalnej była opracowana blokowa metoda $\mathbf{L}\cdot\mathbf{S}\cdot\mathbf{L}^T$, gdzie **S** – diagonal znaków. Rozmiar bloku był równy ilości stopni swobody w węźle modelu MES (lb = 3 dla podanego problemu), brakowało wspierania wielowątkowości, łączenia grup równań w większe bloki, użycia XMM rejestrów oraz ich blokowania (SCAD v. 11).

W wersji z 2008 roku zostało dodane wspieranie wielowątkowości (SCAD v.11.1). W 2009 roku było zrealizowane łączenie grup równań w większe bloki, stosowanie rejestrów XMM oraz ich blokowanie, przepakowanie danych i rozwijanie pętli wewnętrznej.

W ostatnim wierszu tab. 5.4 są przedstawione czasy obliczeń tego zadania przez wielofrontalny solwer ANSYS v. 11.0 na platformie 32-bitowej. Okazało się, że BSMFM w wersji z 2009 roku nie ustępuje wielofrontalnemu solwerowi wiadomego programu MES przynajmniej dla tego zadania testowego.

Na rys. 5.16 jest przedstawiony diagram trwałości faktoryzacji numerycznej dla różnych wersji tej metody.



Rys. 5.16. Trwałość faktoryzacji numerycznej dla różnych wersji metody

5.8 PARDISO

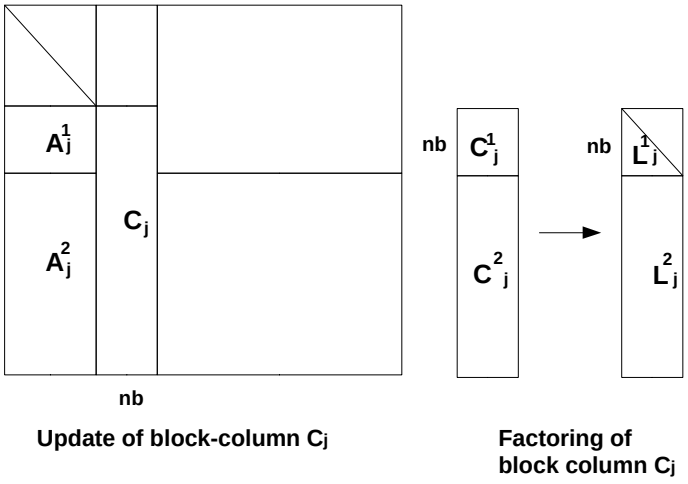
PARDISO (parallel direct solver) [Schenk O, Gartner K., 2002] znajduje się w składzie procedur biblioteki Intel MKL i demonstruje wysoką wydajność oraz dobry speed up na komputerach z pamięcią wspólną. Podstawą dla wysokiej wydajności jest podział macierzy rzadkiej na gęste prostokątne bloki, do których następnie stosują się procedury poziomu 3 BLAS.

W tym celu stosuje się tworzenie drzewa eliminacji i drzewa superwęzłowego (rozdział 5.6).

Przedstawione w tym rozdziale wzory dla macierzy symetrycznych powstały z bardziej ogólnego podejścia [Schenk O, Gartner K., 2002] dla macierzy niesymetrycznych.

Algorytm sekwencyjny, zrealizowany w PARDISO, dla macierzy symetrycznej wygląda tak, że przy faktoryzacji bieżącej kolumny blokowej $\mathbf{C}_j$ (rys. 5.17) najpierw jest wykonane jej poprawienie – modyfikacja zewnętrzna:

$$\mathbf{C}_j = \mathbf{C}_j - \begin{pmatrix} \mathbf{A}_j^1 \\ \mathbf{A}_j^2 \end{pmatrix} \left(\mathbf{A}_j^1\right)^T \tag{5.16}$$



Rys. 5.17. PARDISO: poprawienie kolumny blokowej *j* oraz jej faktoryzacja

Następnie trzeba sfaktoryzować poprawioną kolumnę *j* – etap wewnętrznej faktoryzacji. Dlatego sfaktoryzujemy diagonalny blok $\mathbf{C}_j^1 = \mathbf{L}_j^1 \cdot \left(\mathbf{L}_j^1\right)^T$ – powstaje dolna trójkątna macierz $\mathbf{L}_j^1$. Dalej obliczamy blok $\mathbf{L}_j^2$:

$$\mathbf{L}_j^1 \left(\mathbf{L}_j^2\right)^T = \left(\mathbf{C}_j^2\right)^T \rightarrow \mathbf{L}_j^2 \tag{5.17}$$

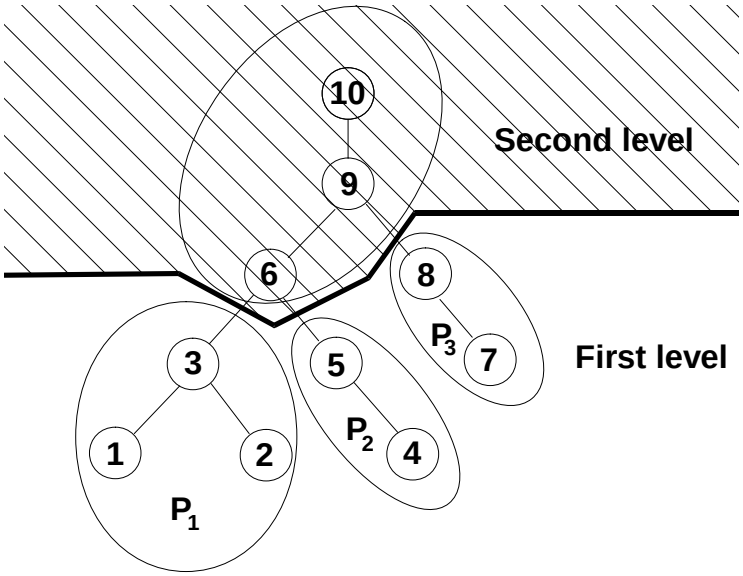
Dlatego rozwiązujemy układ równań liniowych algebraicznych z macierzą dolną trójkątną $\mathbf{L}_j^1$ i paczką prawych stron $\left(\mathbf{C}_j^2\right)^T$, gdzie $\mathbf{C}_j^2$ – część kolumny blokowej $\mathbf{C}_j$, umieszczonej pod jej blokiem diagonalnym $\mathbf{C}_j^1$,

$$\mathbf{C}_j = \begin{pmatrix} \mathbf{C}_j^1 \\ \mathbf{C}_j^2 \end{pmatrix} . \tag{5.18}$$

Jest to podobne do metody looking left, tylko uogólnione dla wersji blokowej.

W trybie wielowątkowym PARDISO używa dwupoziomowy left-right looking algorytm faktoryzacji macierzy rzadkich [Schenk O, Gartner K., 2002].

Pierwszy poziom jest związany z tym, że drzewo superwęzłowe (rys.5.18) ma wiele gałęzi, superwęzły każdej z nich można sfaktoryzować niezależnie od innych. Daje to możliwość równolegle sfaktoryzować kolumny blokowe, reprezentowane na drzewie eliminacji wierzchołkami, należącymi do różnych gałęzi (first level – rys. 5.18).



Rys. 5.18. Rzutowanie wierzchołków drzewa eliminacji na 3 procesory P_1, P_2, P_3 w dwóch poziomowym left-right looking algorytmie PARDISO

Wierzchołkom górnej części drzewa eliminacji odpowiadają kolumny blokowe o dużej szerokości *nb* (rys. 5.17). Dlatego duża ilość pracy obliczeniowej przypada na górną część drzewa eliminacji.

Dwupoziomowy algorytm faktoryzacji left-right looking jest przedstawiony na rys. 5.19.

Kolejka gałęzi Q_s drzewa eliminacji jest tworzona w trakcie wykonania analizy macierzy rzadkiej. Dla przykładu, podanego na rys. 5.18, $Q_s = \{Q_1, Q_2, Q_3\}$, gdzie $Q_1 = \{1, 2, 3\}$, $Q_2 = \{4, 5\}$, $Q_3 = \{7, 8\}$.

Algorytm faktoryzacji zaczyna się z pierwszego równoległego regionu *First level*, w którym są wykonane pętle *while* dopóki wszystkie wierzchołki niezależnych gałęzi drzewa eliminacji nie zostaną sfaktoryzowane.

First level:

while (*not local independent subtrees processed*)
    lock 1
        remove one subtrees from subtree queue Q_s
    unlock 1
    factorize local subtree
    lock 2a
        perform right looking: subtree is factorized
    unlock 2a
end while (no wait)

Second level:

while (*not all supernodes factorized in root node queue* Q_r)
    lock 2b
        remove new supernode j from root node queue Q_r
    unlock 2b
    if (*supernode j receives updates from previous supernodes k*)
        if (*thread has to wait*)
            lock 2c
                pushback supernode j to root node queue Q_r
            unlock 2c
            goto Second level
        else
            perform external factoring of supernode j with supernode k
        end if
    end if
    compute internal supernode factoring
    lock 2d
        perform right-looking: supernode j is computed
    unlock 2d
end while

Rys. 5.19. Dwóch poziomowy algorytm faktoryzacji left-right looking [Schenk O, Gartner K., 2002]

W sekcji krytycznej 1 odbywa się rzutowanie bieżącej gałęzi drzewa eliminacji na procesor. Z kolejki Q_s jest pobierany i od razu wykreślony najbliższy element. Dalej każda z równoległych pętli *while* wykonuje sekwencyjną looking left faktoryzację kolumn blokowych, odpowiadających wierzchołkom bieżącej gałęzi Q_i (i = 1, 2, 3 dla podanego przykładu). Jak tylko odpowiednia kolumna blokowa j zostanie sfaktoryzowana, wykonuje się etap looking right, który polega na przeanalizowaniu niezerowej struktury kolumny blokowej L_j i uzupełnieniu $List[k] \leftarrow j$, gdzie $k \in L_j \wedge k > j$. Jest to blokowane sekcją krytyczną 2a, żeby uniemożliwić modyfikacje tej samej listy $List[k]$ jednocześnie przez kilka różnych wątków.

Osiągnięcie zrównoważenia obciążenia wątków (load balance) na pierwszym poziomie jest trudne, ponieważ ciężko przewidzieć ilość operacji, wykonanych dla każdej gałęzi –drzewo eliminacji zwykłe jest słabo zbalansowane. Dlatego przy zakończeniu swojej pracy w pierwszym równoległym regionie żaden wątek nie oczekuje na inny, lecz od razu wchodzi w drugi równoległy region *Second level* (opcja *no wait*).

W drugim równoległym regionie znów działają pętle *while*, w każdej z nich z kolejki dla korzeniowej gałęzi drzewa eliminacji Q_r (dla naszego przykładu Q_r = {6, 9, 10}) jest pobierany i od razu wyeliminowany najbliższy element. Odbywa się to w sekcji krytycznej 2b.

Jeśli bieżąca kolumna blokowa (superwęzeł) *j* powinna być poprawiona poprzednimi kolumnami blokowymi *k*, gdzie $k \in List[j]$ ($List[j]$ nie jest zbiorem pustym) i żadna kolumna z innego wątku nie poprawia w tym momencie kolumny *j*, wykonujemy zewnętrzną faktoryzację – poprawienie kolumny *j* przez kolumnę *k*. Jeśli kolumna innego wątku już poprawia kolumnę *j*, indeks *j* umieszczamy w koniec kolejki Q_r (sekcja krytyczna 2c), a wątek bieżący przekazuje sterowanie na początek równoległego regionu (*goto* Second level), pobiera z kolejki Q_r numer innej kolumny blokowej i próbuje ją poprawić.

Jak tylko kolumna blokowa *j* będzie poprawiona przez wszystkie kolumny $k \in List[j]$ (lista $List[j]$ staje się pusta), w sekcji krytycznej 2d wykonujemy etap looking right – na podstawie niezerowej struktury kolumny blokowej *j* uzupełniamy odpowiednie listy kolumn, umieszczonych z prawa.

Pętle *while* działają dopóki wszystkie superwęzły korzeniowej gałęzi drzewa eliminacji nie zostaną sfaktoryzowane.

Formalnie PARDISO wspiera tryb faktoryzacji przy użyci dysku – tryb wirtualizacji OOC (out of core). Jak pokazują wielokrotne testy, wykonane autorem, w trybie OOC PARDISO jest w stanie rozwiązywać tylko stosunkowo nie wielkie zadania, a dla dużych zadań ten tryb nie działa. Będzie to pokazane poniżej. Dlatego my odnosimy PARDISO do klasy solwerów, które działają tylko na pamięci głównej. Jest to poważne ograniczenie dla stosowania tej metody.

5.10 PARFES

PARFES (parallel finite element solver) [Fialko S., 2010] jest metodą rozwiązywania układów równań liniowych algebraicznych z macierzami rzadkimi symetrycznymi, które powstają przy zastosowaniu MES do zadań mechaniki konstrukcji oraz mechaniki ciała stałego.

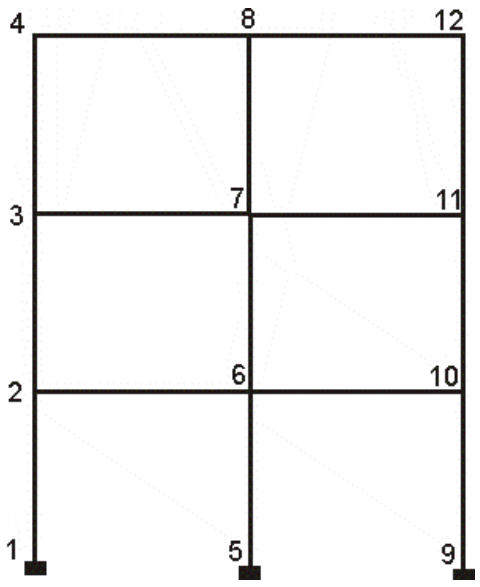
Podstawą dla opracowania tej metody posłużyło to, że solwery wielofrontalne bardzo oszczędnie wykorzystują pamięć główną, łatwo ulegają wirtualizacji, jednak na komputerach z pamięcią wspólną wykazują znacznie mniejszą

wydajność w porównaniu z PARDISO oraz innymi metodami z bibliotek wysokiej wydajności, ponieważ mają zbyt dużo transferów danych pomiędzy różnymi buforami pamięci RAM albo pamięcią RAM i dyskiem. Odbywa się to przy umieszczeniu macierzy poprawek (niezakończonego frontu) w buforze pamięci RAM oraz przy pobraniu tych danych przy agregowaniu kolejnej macierzy frontalnej. Operacje kopiowania pamięć – pamięć należą do operacji niskiej wydajności i przy zwiększeniu ilości procesorów są bardzo słabo przyspieszane na komputerach z pamięcią wspólną.

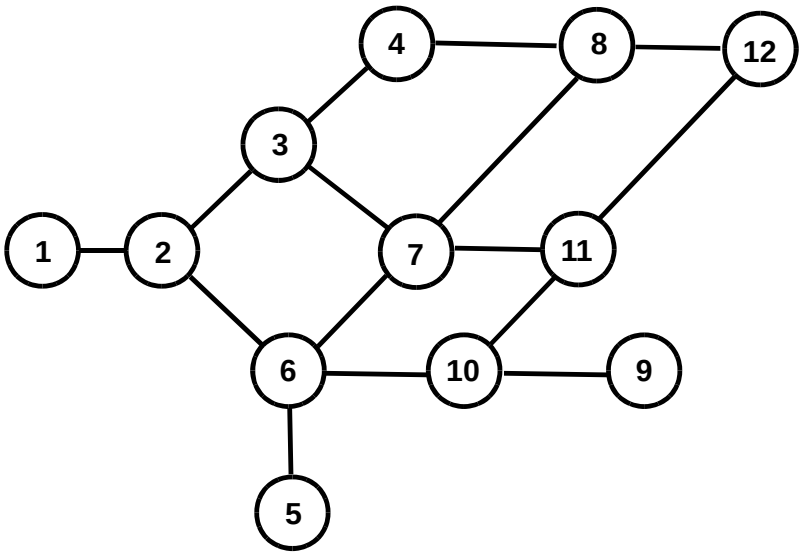
Z drugiej strony, PARDISO ta inne metody z bibliotek wysokiej wydajności, demonstrują wysoką wydajność, dobre przyspieszenie przy zwiększeniu ilości procesorów, jednak, jeśli rozmiar zadania przekroczy możliwości pamięci RAM, to takie zadanie nie będzie rozwiązane.

Stąd i powstała idea opracowania takiej metody, która by wykazywała wysoką wydajność oraz dobry speed up, jeśli zadanie się mieści w pamięci głównej (tryb core mode – CM), oraz była by w stanie podłączyć dysk i rozwiązać zadanie, rozmiar którego jest zbyt duży dla umieszczenia w pamięci RAM (tryb out of core OOC). Przy tym ta metoda w trybie OOC powinna wykazywać stabilny speed up, chociaż bardziej niski, niż w trybie CM.

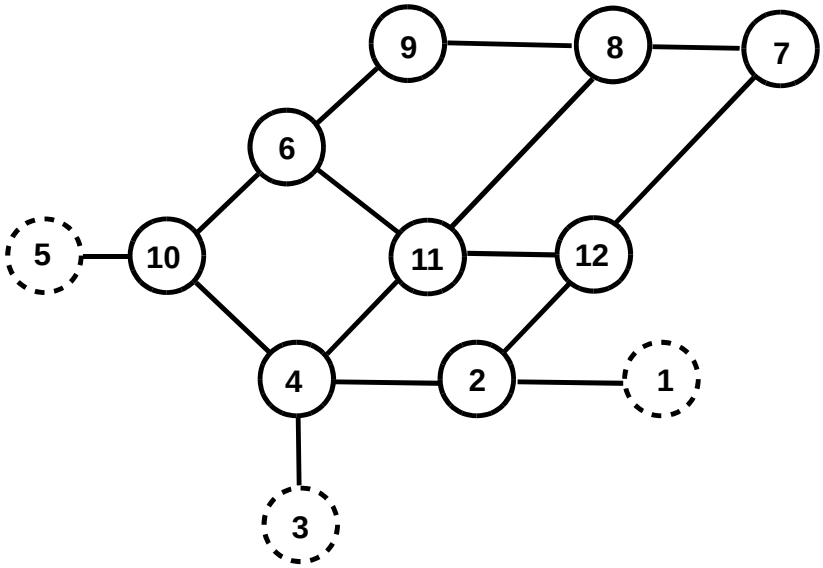
Rozważmy przykład ramy płaskiej (rys. 5.20). Graf przyległości jest podany na rys. 5.21. Po zastosowaniu uporządkowania algorytmem minimalnego stopnia otrzymamy graf przyległości z nowymi numerami węzłów (rys. 5.22).



Rys. 5.20. Przykład ramy płaskiej



Rys. 5.21. Graf przyległości dla węzłów modelu, przedstawionego na rys. 5.21



Rys. 5.22. Graf przyległości po uporządkowaniu

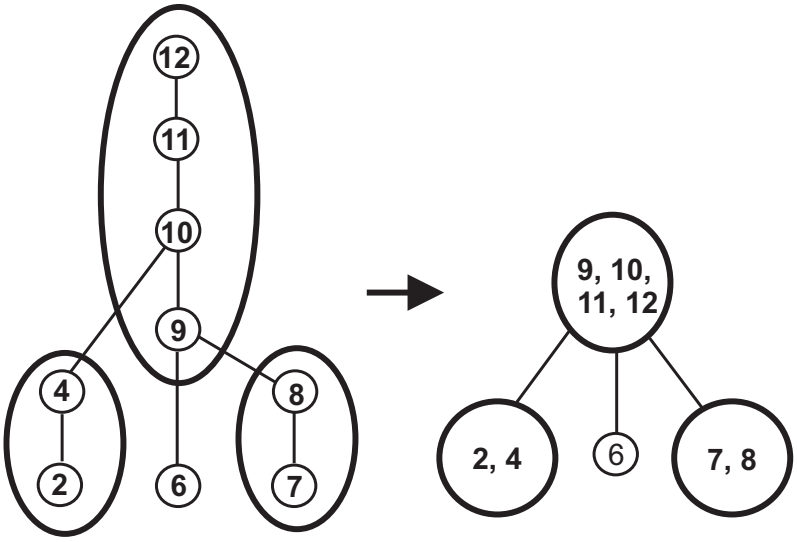
Węzły 1, 3, 5 w nowym układzie numerowania odpowiadają węzłom 9, 5, 1 w układzie początkowym. Są to węzły podparte, nie produkują one żadnego równania i dlatego oznaczone są linią przerywaną.

Po wykonaniu faktoryzacji symbolicznej otrzymujemy niezerową strukturę macierzy rzadkiej po uporządkowaniu (rys. 5.23). Brakuje tu kolumn 1, 3, 5, ponieważ nie tworzą one równań.

Dalej trzeba podzielić macierz rzadką na gęste bloki prostokątne tak, żeby dołożyć minimalną ilość elementów zerowych. Dlatego stosujemy technikę superwęzłową. Tworzymy drzewo eliminacji i łączymy jego wierzchołki w superwęzły (rys. 5.24).

$$\begin{pmatrix}
2 & & & & & & & & \\
x & 4 & & & & & & & \\
 & & 6 & & & & & & \\
 & & & 7 & & & & & \\
 & & & x & 8 & & & & \\
 & & x & & x & 9 & & & \\
 & x & x & & & x & 10 & & \\
 & x & x & & x & x & x & 11 & \\
x & x & & x & x & x & x & x & 12
\end{pmatrix}$$

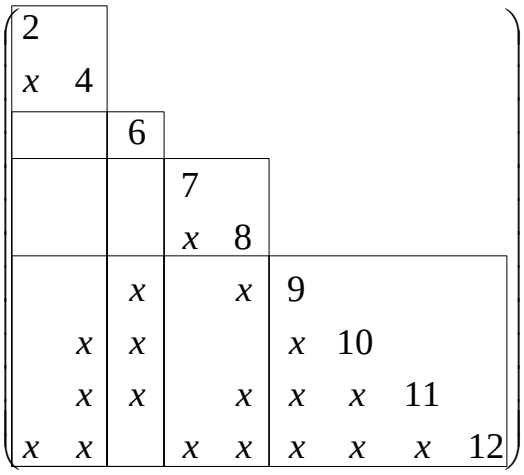
Rys. 5.23. Niezerowa struktura macierzy rzadkiej po uporządkowaniu



Rys. 5.24. Drzewo eliminacji oraz drzewo superwęzłowe

Przy tworzeniu drzewa eliminacji w ścieżce 2 – 4 trzeba fikcyjnie dopisać pominięte wierzchołki 1, 3. Wtedy będziemy mieli sekwencyjne numerowanie wierzchołków tej samej gałęzi, a to daje podstawę połączyć wierzchołki 2 – 4 w superwęzeł.

Podział macierzy na bloki prostokątne według podanego drzewa superwęzłowego jest przedstawiony na rys. 5.25.

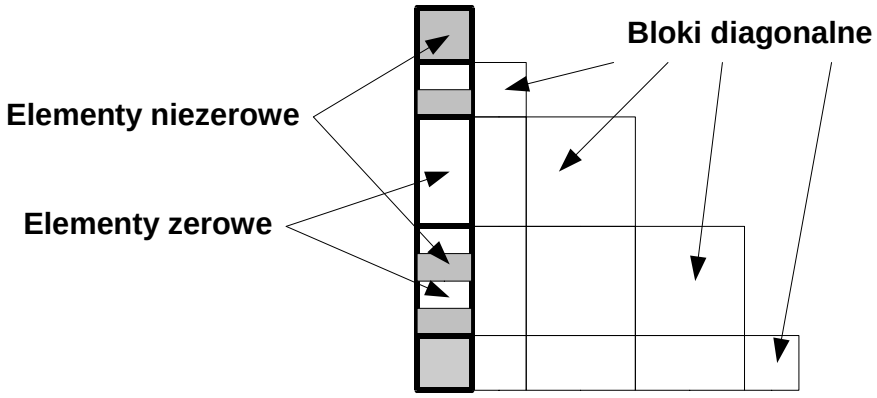


Rys. 5.25. Podział macierzy rzadkiej na bloki

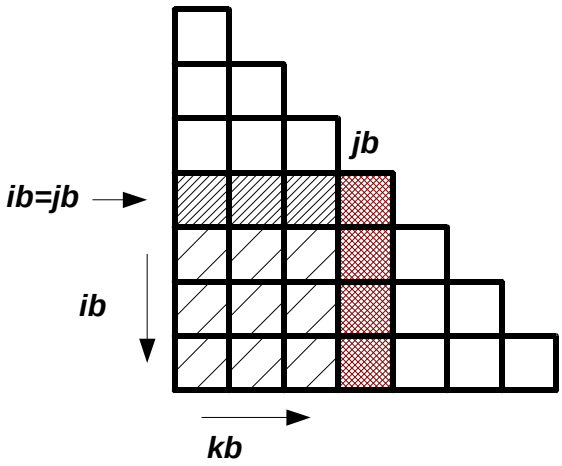
Bloki diagonalne są zawsze kompletnie wypełnione. Bloki pozadiagonalne mogą być kompletnie wypełnionymi (w podanym prostym zadaniu takich bloków niema), częściowo wypełnionymi (są to bloki z indeksami $\mathbf{A}_{41}$; $\mathbf{A}_{42}$; $\mathbf{A}_{43}$) oraz blokami pustymi ($\mathbf{A}_{21}$, $\mathbf{A}_{31}$, $\mathbf{A}_{32}$). Dla bloków pustych pamięć nie jest alokowana. Dla bloków kompletnie wypełnionych umieszczamy elementy macierzy kolumna po kolumnie, a dla bloków częściowo wypełnionych pamięć jest alokowana tylko dla niezerowych wierszy. Wierz uważamy za niezerowy, jeśli przynajmniej jeden element tego wiersza jest różny od zera. W naszym przykładzie dla bloku $\mathbf{A}_{41}$ przechowujemy wierszy 2, 3, 4, dla bloku $\mathbf{A}_{42}$ – wierszy 1, 2, 3, a dla bloku $\mathbf{A}_{43}$ – wierszy 1, 3, 4. W ogólnym przypadku wygląd kolumny blokowej jest przedstawiony na rys. 5.26.

Dla faktoryzacji macierzy rzadkiej stosujemy blokową metodę Choleskiego left-looking. Dla macierzy gęstej algorytm faktoryzacji jest przedstawiony na rys. 5.27, 5.28. Faktoryzujemy kolumny blokowe kolejno z lewa na prawo. Przy faktoryzacji kolumny *jb* trzeba najpierw wykonać jej poprawienie przez kolumny, umieszczone z lewa (pętle *do kb* = 1, *jb*-1). Każda kolumna blokowa o indeksie *kb* będzie

mnożona przez blok $\mathbf{A}_{jb,kb}^{T}$, znajdujący się w wierszu blokowym o indeksie *ib* = *jb* . Dla obliczenia iloczynu dwóch macierzy stosujemy procedurę DGEMM z Intel Math Kernel Library. To daje możliwość wykonywać obliczenia na szczycie wydajności procesora.



Rys. 5.26. Typowa struktura kolumny blokowej



Rys. 5.27. Faktoryzacja macierzy gęstej blokową metodą Choleskiego left-looking

do $jb = 1, N_b$
  // *update of column jb by columns, are located at left*
  do $kb = 1, jb - 1$
    do $ib = jb, N_b$
      $\mathbf{A}_{ib,\,jb} = \mathbf{A}_{ib,\,jb} - \mathbf{A}_{ib,kb} \cdot \mathbf{A}^T_{jb,kb}$
    end do
  end do
  // *Factoring of column jb*
  $\mathbf{A}_{jb,\,jb} = \mathbf{L}_{jb,\,jb}\mathbf{L}^T_{jb,\,jb}$
  do $ib = jb + 1, N_b$
    $\mathbf{L}_{jb,\,jb}\mathbf{L}_{ib,\,jb} = \mathbf{A}_{ib,\,jb} \rightarrow \mathbf{L}_{ib,\,jb}$
  end do
end do

Rys. 5.28. Algorytm faktoryzacji macierzy gęstej blokową metodą Choleskiego left-looking

Po poprawieniu kolumny blokowej *jb* wszystkimi kolumnami, umieszczonymi z lewa, sfaktoryzujemy diagonalny blok i obliczamy bloki, umieszczone pod blokiem diagonalnym. Dla każdego takiego bloku rozwiązujemy układ równań liniowych algebraicznych z macierzą dolną trójkątna – wykonujemy podstawienie bezpośrednie (pętla *do ib = jb*+1, N_b).

Dla macierzy rzadkich przy poprawieniu kolumny *jb* kolumnami, umieszczonymi z lewa, będą brali udział tylko takie kolumny *kb*, dla których blok, znajdujący się w wierszu *ib = jb*, jest blokiem niezerowym: *kb* ϵ *List*[*jb*] . Oprócz tego indeks *ib* przyjmuje tylko takie wartości, które odpowiadają niezerowym blokom $\mathbf{A}_{ib,kb}$: *ib* ϵ *L*, gdzie *L* – niezerowa struktura macierzy sfaktoryzowanej. Pozwala to obejść mnożenie przez bloki zerowe.

Algorytm blokowy looking left dla macierzy rzadkiej jest przedstawiony na rys. 5.29

1. **if**(CM)
   prepare block column $jb \in [1, Nb]$
2. **do** $jb = 1, Nb$
3. **if**(OOC ∨ OOC1)
   prepare block column jb
4. parallel update of column jb:
   if(OOC1)
   read block column kb from disk.

$$\mathbf{A}_{ib,jb} = \mathbf{A}_{ib,jb} - \sum_{kb \in List[jb]} \mathbf{A}_{ib,kb} \cdot \mathbf{S}_{kb} \cdot \mathbf{A}^{T}{}_{jb,kb}; \quad ib \geq jb, ib \in L$$

5. factoring of block column jb

$$\mathbf{A}_{jb,jb} = \mathbf{L}_{jb,jb} \cdot \mathbf{S}_{jb} \cdot \mathbf{L}^{T}{}_{jb,jb}$$

$$\mathbf{L}_{jb,jb} \cdot \mathbf{S}_{jb} \cdot \mathbf{L}^{T}{}_{ib,jb} = \mathbf{A}^{T}{}_{ib,jb} \quad \rightarrow \quad \mathbf{L}_{ib,jb}; \quad ib > jb, ib \in L$$

   if(OOC ∨ OOC1)
   write block column jb to disk and free RAM
   for blocks in row $ib = jb$.
6. add jb to $List[lb]$, $lb > jb$, if block column jb updates the block column lb.
7. **end do**

Rys. 5.29. Algorytm faktoryzacji macierzy rzadkiej blokową metodą Choleskiego left-looking

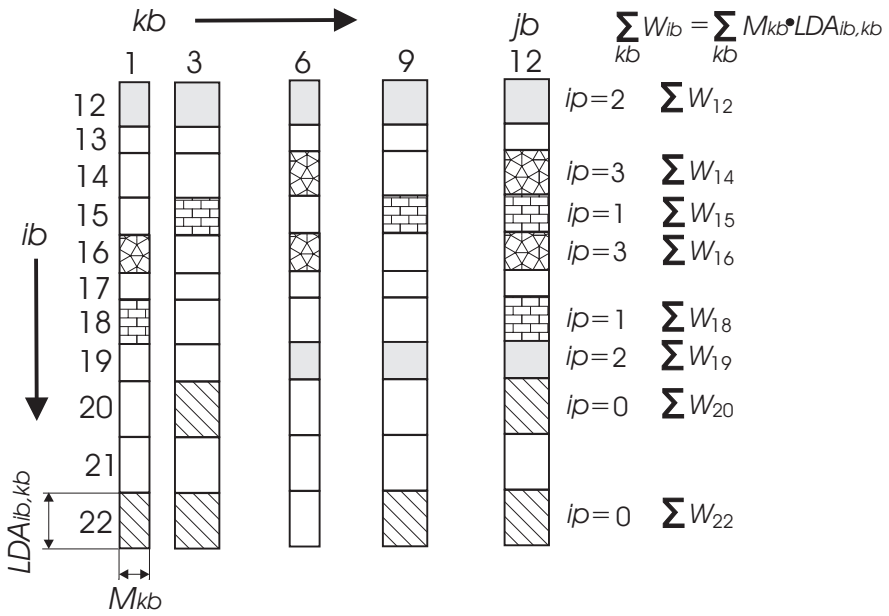
Tu OOC, OOC1 oznacza tryb wymiany danych z dyskiem. Będzie o tym mowa później. Diagonal znaków **S** powstaje przy uogólnieniu metody Choleskiego na macierze nieokreślone.

Ponad 99% operacji arytmetycznych wykonuje się na etapie 4. Dlatego ten etap powinien być zrównoleglony w pierwszej kolejce.

Schemat zrównoleglenia jest przedstawiony na rys. 5.30 [Fialko S., 2010].

W podanym przykładzie kolumna blokowa o numerze 12 jest poprawiona przez kolumny 1, 3, 6, 9. Obliczenia są wykonane na czterech wątkach ($ip \in [0, 3]$, gdzie ip – numer watku). Dla uniknięcia stosowania synchronizacji, która drastycznie zmniejsza wydajność przy wykonywaniu takich zadań w trybie wielowątkowym, będziemy wykonywali obliczenia dla każdego wiersza blokowego na tym samym wątku. Wtedy sytuacja, kiedy dwa różnych wątki dokonują zapis w ten samy blok kolumny 12, jest kompletnie wykluczona.

Dla zabezpieczenia zrównoważenia obciążenia pomiędzy wątkami dla każdego wiersza blokowego liczymy ilości niezerowych elementów (wagi $\sum W_{ib}$). Dalej sortujemy wiersze blokowe w porządku malejącym według ich wag i po kolei przypisujemy te wiersze do procesorów fizycznych (wątków). Przy tym na każdym kroku takiego przypisania bieżący wiersz będzie przypisany do tego procesora, który ma minimalną sumę wag (najmniejszą objętość pracy obliczeniowej).



Rys. 5.30. Schemat zrównoleglenia obliczenia poprawek do kolumny *jb*

Matematyczna formalizacja tego algorytmu jest przedstawiona poniżej.

a. Dla każdego wiersza blokowego wyznaczamy sumę wag $\sum W_{ib} = \sum_{kb \in List[\,jb\,]} M_{kb} \cdot LDA_{ib,kb}$.

b. Sortujemy kolumny blokowe w porządku malejącym według ich wag.

c. $\forall ip \in \lfloor 0,1,\ldots,n_p - 1 \rfloor \quad sumofweights[ip] = 0$

$\forall ib \in L_{jb} \qquad (loop\ ib)$

d. $Find \quad \min\_ip \in [0,\ n_p - 1]\big|_{sumofweights[ip]\ is\ \min}$

$sumofweights[\min\_ip] += \sum W_{ib}; \quad thread\_numb[ib] = \min\_ip;$

$end\ loop\ ib$

e.
$\forall kb \in Link[jb]$ (*loop kb*)
$\forall ib \in L_{kb}$ (*loop ib*)
$Q[thread\_numb[ib]] \leftarrow \{\mathbf{A}_{ib,kb}, \mathbf{A}_{jb,kb}, kb\}$ (*put to queue Q*[*ip*])
end loops ib, kb

W punkcie *c* tablica *sumofweights*, która przechowuje sumę wag dla każdego wątku $ip \in [0, n_p\text{-}1]$, jest wyzerowana. Tu n_p – ilość wątków. W punkcie *d* algorytm znajduje taki numer wątku *min_ip*, który zawiera minimalną sumę wag, przypisuje bieżący wiersz blokowy *ib* do tego wątku, poprawia jego sumę wag i umieszcza numer tego wątku w tablicę *thread_numb*, która przechowuje dla każdego wiersza blokowego numer wątku, który będzie wykonywał obliczenia w tym wierszu. W punkcie *e* dla każdego wątku tworzymy kolejki *Q*[*ip*]. Każdy element kolejki *Q*[*ip*] zawiera wskaźniki do odpowiednich bloków macierzy oraz numer indeksu *kb*.

Po przygotowaniu kolejek wykonujemy równoległe poprawianie kolumny blokowej *jb*.

 # pragma omp parallel
while$(Q[ip]$ *is not empty*$)$
$\{\mathbf{A}_{ib,kb}, \mathbf{A}_{jb,kb}, kb\} \leftarrow (Q[ip] / \{\mathbf{A}_{ib,kb}, \mathbf{A}_{jb,kb}, kb\})$
$\mathbf{A}_{ib,\,jb} = \mathbf{A}_{ib,\,jb} - \mathbf{A}_{ib,kb} \cdot \mathbf{S}_{kb} \cdot \mathbf{A}^{T}{}_{jb,kb};$
end while
end parallel region

W równoległym regionie wykonujemy pętlę *while* dopóki kolejka, odpowiadająca wątkowi $ip \in [0, n_p - 1]$, nie będzie pusta. Przy każdej iteracji pętli dla wątku *ip* pobieramy odpowiedni element kolejki i od razu wykreślamy go $(Q[ip] / \{\mathbf{A}_{ib,kb}, \mathbf{A}_{jb,kb}, kb\})$. Otrzymujemy wskaźniki do odpowiednich bloków macierzy i wykonujemy obliczenia, używając procedury DGEMM z Inlet MKL.

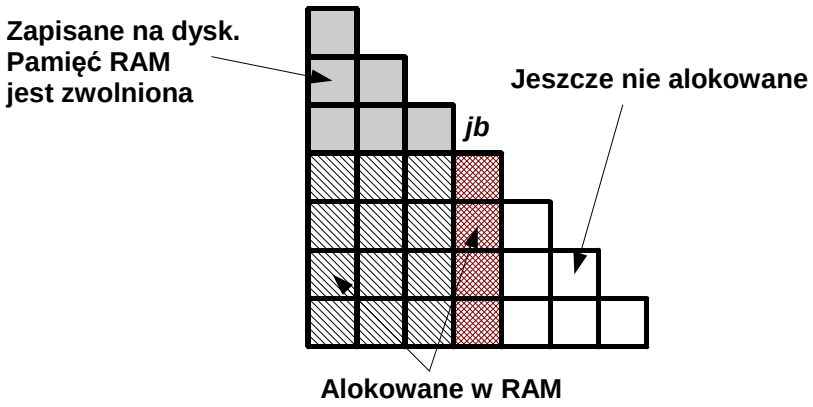
Zrównoleglenie na etapie 5 (rys.5.29) jest wykonywane przy zastosowaniu odpowiednich równoległych procedur z biblioteki Inlet MKL.

Jeśli zadanie nie mieści się w pamięci głównej, będzie użyty jeden z dwóch trybów wirtualizacji metody.

Tryb OOC zawiera minimalną ilość odwołań do dysku, ponieważ potrzebuje znacznie więcej pamięci głównej, niż tryb OOC1. W trybie OOC1 w skutek znacznie większej ilości odwołań do dysku wydajność metody zmniejsza się, jednak to daje możliwość rozwiązywania dużych zadań na komputerach z małą pamięcią główną.

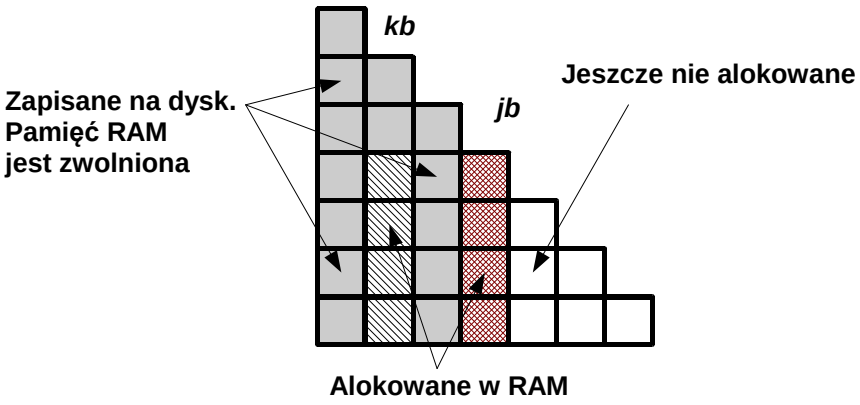
W trybie OOC przy poprawieniu kolumny *jb* w pamięci głównej są utrzymywane bloki, oznaczone na rys. 5.31. Przed obliczeniem kolumny *jb* będzie alokowana pamięć dla jej bloków i te bloki będą wczytane z dysku. Po obliczeniu

kolumny *jb* jej bloki są zapisywane na dysk. Pamięć dla bloków wszystkich kolumn, umieszczonych z lewa, które nie będą potrzebne dalej, będzie zwolniona.



Rys. 5.31. Utrzymywane w RAM bloki macierzy w trybie OOC

W trybie OOC1 przy poprawieniu kolumny *jb* w pamięci głównej są utrzymywane kolumna *jb* oraz potrzebne bloki kolumny *kb* (rys. 5.32).



Rys. 5.32. Utrzymywane w RAM bloki macierzy w trybie OOC1

Dlatego są potrzebne dwa bufory, alokowane w pamięci głównej. Jeden bufor mieści kolumnę *jb*, a drugi – bloki kolumny *kb*. Przed obliczeniem kolumny *jb* z dysku są wczytane jej bloki dla macierzy źródłowej. Przy każdym

inkrementowaniu indeksu *kb* (rys. 5.29, punkt 4) potrzebne bloki kolumny *kb* będą odczytane z dysku. Ilość odwołań do dysku istotnie wzrasta w porównaniu z trybem OOC, jednak objętość używanej pamięci głównej jest znacznie mniejsza. Po sfaktoryzowaniu kolumny *jb* jej bloki będą zapisane na dysk.

Porównamy czasy obliczeń różnymi metodami dużego zadania *schema_new_1*, wziętego z zestawu zadań firmy SCAD Soft [SCAD], przedstawionego na rys. 1.5 i zawierającego 3 198 609 równań liniowych algebraicznych. Będziemy porównywali PARFES, PARDISO i BSMFM. Ostatnia metoda może być potraktowana jako dobra realizacja solwera wielofrontalnego (tab. 5.4, wiersze 5 i 7) i dlatego wybrana dla takiego porównania.

W tab. 5.5 są podane czasy obliczeń, otrzymane na komputerze z procesorem Intel® Core™2 Quad CPU Q6600 @2.40 GHz, cache L1: 4x32 KB, L2: 4096 KB, RAM DDR2 333.7 MHz, 8 GB, chipset Intel P35/G33/G31, OS Windows Vista™ Business (64-bit), Service Pack 2.

W kolumnie "NonZero(L)" są rozmiary sfaktoryzowanej macierzy w MB. Dla każdej z metod był użyty algorytm uporządkowania METIS [Karypis, Kumar , 1995]. W kolumnie "Analysis" została przedstawiona trwałość etapu analizy, a w kolumnie "Solution phase" – trwałość podstawień bezpośrednich i wstecznych na jednym i na czterech procesorach.

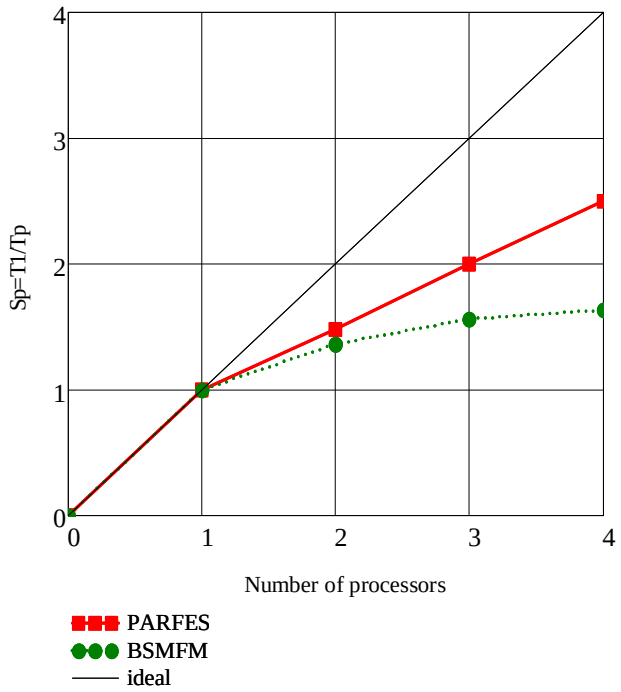
Każda z metod działa w trybie OOC. PARDISO skończył wykonanie zadania awaryjnie z błędem -11. PARFES na 4 procesorach na etapie faktoryzacji numerycznej wykazał czas obliczeń, trzy razy mniejszy od trwałości metody BSMFM. BSMFM demonstruje najmniejszy czas wykonania podstawień bezpośrednich oraz wstecznych, ponieważ prowadzi do mniejszego rozmiaru macierzy sfaktoryzowanej.

Tabela 5.5

Trwałość etapów rozwiązania zadania *schema_new_1* (3 198 609 równań) na komputerze z procesorem Intel® Core™2 Quad CPU Q6600 @2.40 GHz [Fialko S., 2010].

Method	NonZero(L) MB	Ana-lysis s	Numerical factorization, s				Solution phase, s	
			Number of processors				Number of proc.	
			1	2	3	4	1	4
PARFES (OOC)	12 186	23.6	1 190	802	594	475	804	526
PARDISO(OOC)	10 662	61.4	Numerical factorization phase: error = -11					
BSMFM (OOC)	10 869	9.0	2 011	1 482	1 286	1 232	497	

Na rys. 5.33 jest przedstawiony speed up dla metod PARFES i BSMFM [Fialko S., 2010].

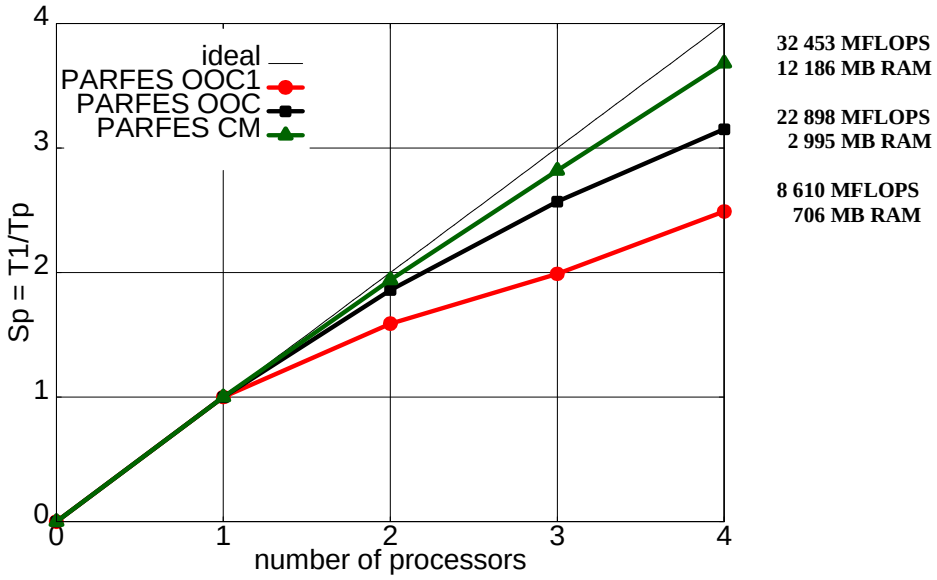


Rys. 5.33. Zadanie *schema_new_1* (3 198 609 równań). Przyspieszenie etapu faktoryzacji numerycznej w trybie OOC na komputerze z procesorem Core™2 Quad

Nawet w trybie OOC PARFES demonstruje stabilny speed up.

Na rys. 5.34 dla metody PARFES w trybach CM, OOC, OOC1 jest podany speed up na etapie faktoryzacji numerycznej zadania *schema_new_1* (3 198 609 równań) na komputerze z czterordzeniowym procesorem AMD Phenom™ II x4 995 3.2 GHz; L1: 4x64 KB, L2: 4x512 KB, L3: 6 MB; RAM: DDR3 1066 MT/s, 16 GB core memory, Chipset: AMD 790X, OS: Windows VistaTM Business (64-bit), Service Pack 2.

W trybie CM (core mode) PARFES demonstruje największą wydajność (32 453 MFLOPS na czterech rdzeniach) oraz najlepszy speed up. Jednak dla rozwiązania zadania jest potrzebne 12 186 MB RAM. W trybie OOC wydajność i speed up są mniejsze (22 898 MFLOPS na czterech rdzeniach), natomiast dla rozwiązania tego zadania wystarczy 2 995 MB RAM. W trybie OOC1 wydajność zmniejsza się do 8 610 MFLOPS, speed up jest niski, jednak dla rozwiązania zadania wystarczyło tylko 706 MB RAM, czyli takie duże zadanie może być rozwiązane aplikacją 32-bitową na zwykłym notebooku.



Rys. 5.34. Zadanie *schema_new_1* (3 198 609 równań). PARFES: przyspieszenie etapu faktoryzacji numerycznej w trybach CM, OOC i OOC1 na komputerze z procesorem AMD Phenom™ II x4 995

W tab. 5.6 są przedstawione trwałości etapów analizy oraz faktoryzacji numerycznej dla metod PARFES, PARDISO i BSMFM na stacji roboczej DELL z dwoma sześciordzeniowymi procesorami Intel Xeon X5660 @ 2.8 GHz /3.2 GHz, z pamięcią RAM DDR3, 24 GB. Ogólna ilość rdzeni wynosi 12. Wszystkie metody działają w trybie core mode (CM).

Czasy obliczeń etapu faktoryzacji numerycznej solwerów PARFES i PARDISO są bardzo bliskie. Największa wydajność zostaje osiągnięta przy 11 wątkach. Wydajność BSMFM jest 10 razy mniejsza.

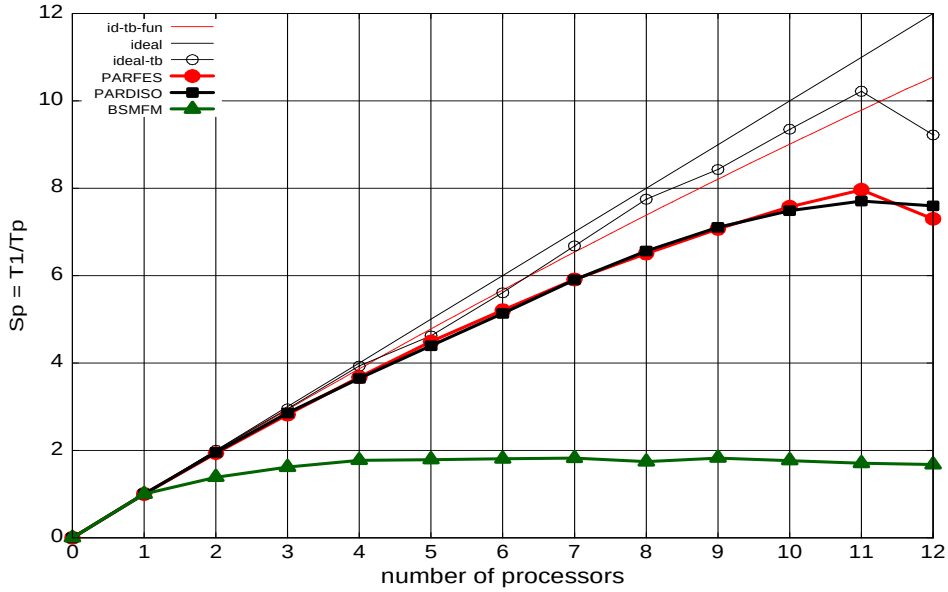
Na rys. 5.35 są przedstawione przyspieszenia PARFES, PARDISO i BSMFM. Procesory Intel Xeon X5660 @ 2.8 GHz /3.2 GHz wspierają tryb *turbo boost* (rozdział 3.4). Krzywa „ideal” odpowiada przyspieszeniu idealnemu, krzywa „ideal-tb-fun” – aproksymacja idealnego przyspieszenia parabolą kwadratową, a krzywa "ideal-tb" jest wynikiem testowania programem, który intensywnie liczy funkcje trygonometryczne i praktycznie nie obciąża układu pamięci.

Krzywe "id-tb-fun" i "ideal-tb" są bliskie, stąd wynika, że mogą oni przedstawiać przyspieszenie idealne dla podanego sprzętu.

Tabela 5.6

Trwałość etapów rozwiązania zadania *schema_new_1* (3 198 609 równań) na stacji roboczej DELL z dwoma sześciordzeniowymi procesorami Intel Xeon X5660 @ 2.8 GHz /3.2 GHz.

Ilość procesorów	PARFES		PARDISO		BSMFM	
	Anal., s	Num. Fakt., s	Anal., s	Num. Fakt., s	Anal., s	Num. Fakt., s
1	16.9	654	31.1	596	13	1406
2	16.9	337.8	23.6	305.3	13	1015
3	16.9	232.1	25.3	208.6	13	869
4	16.9	177.9	23.3	163.3	13	793
5	16.9	145.7	23.8	135.7	13	786
6	16.9	125.6	25.7	116	13	777
7	16.9	110.6	23.1	100.9	13	772
8	16.9	100.5	23.4	90.8	13	807
9	16.9	92.5	24	83.9	13	770
10	16.9	86.3	24	79.6	13	796
11	16.9	82.1	29.9	77.4	13	825
12	16.9	87.5	28.6	78.5	13	839



Rys. 5.35. Zadanie *schema_new_1* (3 198 609 równań). Przyspieszenie etapu faktoryzacji numerycznej w trybie CM na stacji roboczej DELL z dwoma sześciordzeniowymi procesorami Intel Xeon X5660 @ 2.8 GHz /3.2 GHz

Speed up dla PARFES i PARDISO okazał się bardzo bliski i znacznie lepszy od przyspieszenia metody BSMFM.

Ten sam problem pozostał rozwiązany na zwykłym notebooku Toshiba Sattelite z procesorem Intel Pentium Dual CPU T3200 @ 2.00 GHz, cache: L1: 32 KB, L2: 1024 KB, RAM: DDR2, 4 GB; chipset: Intel GL40 rev. 07, OS – Windows Vista™ Business (64-bit), Service Pack 2. PARFES był używany, jako aplikacja 32-bitowa. Oznacza to, że dla solwera było dostępne około 900 MB pamięci RAM. Zadanie było rozwiązane w trybie OOC1 na dwóch wątkach. Analiza macierzy trwała 26 s, agregowanie macierzy – 3 min 16 s, faktoryzacja numeryczna – 51 min 51 s, podstawienia bezpośrednie i wsteczne – 19 min 54 s. Całkowity czas rozwiązania zadania wynosi 75 min 32 s (1 godzina 15 minut). Ani PARDISO, ani BSMFM nie potrafili rozwiązać tego zadania na podanym komputerze.

LITERATURA

Amdahl, Gene, 1967. *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*. AFIPS Conference Proceedings (30): 483–485.

Amestoy P. R, Duff I. S, L'Excellent J-Y., 2000. *Multifrontal parallel distributed symmetric and unsymmetric solvers*. Comput. Meth. Appl. Mech. Eng., 184: 501–520.

Ashcraft C., Liu J. W-H., 1996. *Robust Ordering of Sparse Matrices Using Multisection. Technical Report CS 96-01*, Department of Computer Science, York University, Ontario, Canada.

Demmel J. W., 1997. *Applied Numerical Linear Algebra*. Philadelphia: SIAM.

Dobrian F. and Pothen A., 2000. *Oblio: a sparse direct solver library for serial and parallel computations*. Technical Report describing the OBLIO software library. URL: http://www.cs.odu.edu/~pothen/papers.html . Accessed 5 December 2008.

Dobrian F., Kumfert G., Pothen A., 2000. *The Design of Sparse Direct Solvers using Object-Oriented Techniques*, In: Bruaset A M, Langtangen H P, Quak E (eds.) Modern Software Tools in Scientific Computing. Springer-Verlag, pp 89–131.

Dongarra J. J., Mayes P., di Brozolo J.R., 1989. *The IBM RISK System/6000 ana Linear Algebra Operations*. University of Tennessee Computer Science, Tech Report: CS-90-122.

Duff I. S. and Reid J. K., 1983. *The multifrontal solution of indefinite sparse symmetric linear systems*. ACM Transactions on Mathematical Software, 9, 302–325.

Fialko S., 2004. *Stress-Strain Analysis of Thin-Walled Shells with Massive Ribs*. Int. App. Mech, 40, N4, p. 432–439.

Fialko S., 2008 *A Sparse Shared-Memory Multifrontal Solver in SCAD Software*, Proceedings of the International Multiconference on Computer Science and Information Technology; October 20-22, 2008, Wisła, Poland, 3 (2008), ISSN 1896-7094, ISBN 978-83-60810-14-9, IEEE Catalog Number CFP0864E-CDR, pages 277–283. (http://www.proceedings2008.imcsit.org/pliks/47.pdf)

Fialko S., 2009, CT. Fialko S., 2009. *Blokowa wielofrontalna metoda podstruktur do rozwiązywania dużych układów równań MES*. Czasopismo techniczne, 1-NP, 175 – 188

Fialko S., 2009, M. Fialko S., 2009. *Прямые методы решения систем линейных уравнений в современных МКЭ-комплексах*. М.: СКАД СОФТ. The direct methods for solution of linear equation sets in modern FEA software. Moscow, SCAD SOFT.

Fialko S., 2009, CAMES. Fialko S., 2009. *A block sparse shared-memory multifrontal finite element solver for problems of structural mechanics*. Computer Assisted Mechanics and Engineering Sciences, 16, 2009. p. 117 – 131.

Fialko S., 2010. *PARFES: A method for solving finite element linear equations on multi-core computers*. Advances in Engineering Software, 41, 1256–1265.

George A., Liu J. W. H., 1981. *Computer solution of sparse positive definite systems*. New Jersey : Prentice-Hall, Inc. Englewood Cliffs.

George A., Liu J. W-H., 1989. *The Evolution of the Minimum Degree Ordering Algorithm*, SIAM Rev. 31, 1-19.

Golub G. H., Van Loan C. F, 1996. *Matrix Computations*. Third edition. John Hopkins University Press.

Goto K, Van De Geijn R. A., 2008. *Anatomy of High-Performance Matrix Multiplication*. ACM Transactions on Mathematical Software. V 34, 3, 1–25.

Gould N. I. M., Hu Y., Scott J. A., 2005. *A numerical evaluation of sparse direct solvers for the solution of large sparse, symmetric linear systems of equations*. Technical report RAL-TR-2005-005, Rutherford Appleton Laboratory.

Grama A., Gupta A., Karypis G., Kumar V., 2003. *Introduction to parallel computing*. Second edition. Addiso–Wesley.

Intel – heap. Avoiding Heap Contention Among Threads. 2011. URL: http://software.intel.com/en-us/articles/avoiding-heap-contention-among-threads/

Intel – HT. Intel® Hyper-Threading Technology (Intel® HT Technology). 2011. http://www.intel.com/technology/platform-technology/hyper-threading/index.htm

Intel – HT 1. Developing Multithreaded Applications: A Platform. Consistent Approach. V. 2.0, February 2005. Copyright © Intel Corporation 2003-2005 . 2011. URL: http://cache-www.intel.com/cd/00/00/05/15/51534_developing_multithreaded_applications.pdf

Intel TB. Intel® Turbo Boost Technology 2.0. 2011. URL: http://www.intel.com/technology/turboboost/index.htm

Intel MKL, 2011. URL: http://software.intel.com/en-us/articles/intel-math-kernel-library-documentation/

Irons B. M., 1970. *A frontal solution program for finite-element analysis*. International Journal for Numerical Methods in Engineering, 2, 5 –32.

Karypis G., Kumar V. A., 1995. *Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs*. Technical Report TR 95-035, Department of Computer Science, University of Minnesota, Minneapolis.

Mateev N,.Menon V., Pingali K., 2000. *Left-looking to Right-looking and vice versa: An Application of Fractal Symbolic Analysis to Linear Algebra Code Restructuring.* Technical report: Cornell University Ithaca, NY, USA ©2000.

MSDN – Microsoft Developer Network. 2011. URL: http://msdn.microsoft.com/en-us/library/ms123401.aspx

Prefetch, 2011. URL: http://www.docstoc.com/docs/284306/MMCodec_amd64 .

Richter J., 1997. *Advanced Windows*. Third Edition. Microsoft Press.

Rose D. J., 1972. *A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations*. In “Graph theory and computing”, edited by R.C. Read. Academic Press, New York.

SCAD – Интегрированная система прочностного анализа и проектирования конструкций Structure CAD Office. URL: http://www.scadsoft.com/

Schenk O, Gartner K., 2002. *Two-level dynamic scheduling in PARDISO: Improved scalability on shared memory multiprocessing systems*. Parallel Computing, 28, 187–197.

Scott J. A., 2006. *A frontal solver for the 21st century*. Communications in Numerical Methods in Engineering, 22, 1015-1029.

Sloan S. W., 1986. *An algorithm for profile and wavefront reduction of sparse matrices*. International Journal for Numerical Methods in Engineering, 23, 1315–1324.

SSE2 – Streaming SIMD extension 2: 2008. URL: http://ru.wikipedia.org/wiki/SSE2

TBB – Intel Threading Building Blocks. 2011. URL:http://threadingbuildingblocks.org/ .

Yotov K., Roeder T., Pingali K., Gunnels J., Gustavson F., 2007. *An Experimental Comparison of Cache-oblivious and Cache-conscious Programs*. SPAA'07, June 9–11, 2007, San Diego, California, USA.

Карпиловский В. С. [и др.], 2004. SCAD Office. *Вычислительный комплекс SCAD*.– М: Изд-во АСВ.

Касперский К., 2003. *Техника оптимизации программ. Эффективное использование памяти*. "БХВ-Петербург", Санкт-Петербург.

Шагин И. М. Высокопроизводительные вычисления: курс лекций, 2003. URL: http://www.exelenz.ru/learning/parallel-lections/

Załączniki

Załącznik 1. Kod programu MemTest_1.

```
#include "stdafx.h"
#include<iostream>
#include <cmath>
#include <emmintrin.h>
#include "windows.h"
using namespace std;


int _tmain(int argc, _TCHAR* argv[])
{
	int N, ntimes, i, it;
	DWORD t_s;
	double *X = NULL, t_elaps;
	register double dot, dot1;
	const int L1 = 4096;  //ilosc slow double w cache L1 = 32 KB
#ifdef _DEBUG
	cout << "VERSION DEBUG\n";
#else
	cout << "VERSION RELEASE\n";
#endif
	cout << "Input: N, ntimes\n";
	cin >> N >> ntimes;

	N = N/L1;
	N = N*L1;
	//teraz N jest wielokrotne L1 - dzieli sie bez reszty na
	//4, 8 ,..., 2*exp(k) <= 4096

	cout << "N = " << N << " ntimes = " << ntimes << endl;

	try
	{
		X = (double *)_aligned_malloc(N*sizeof(double), 16);
		if(!X)
			throw 10;
	}
	catch(int)
	{
		cout << "Memory allocation error\n";
		exit(1);
	}
```

```
	for(i=0; i<N; i++)
	{
		X[i] = sqrt((double)(i+1));
	}

	//--------Odczyt klasyczny (kod naiwny)
	t_s = GetTickCount();
	for(it=0; it<ntimes; it++)
	{
		dot = 0.0;
		for(i=0; i<N; i++)
		{
			dot += X[i]*X[i];
		}
	}
	t_elaps = (double)(GetTickCount()-t_s);

	cout << "classical access:                          " <<
t_elaps << " msek" << endl;

	//--------Rozwijanie petli x 4
	t_s = GetTickCount();
	for(it=0; it<ntimes; it++)
	{
		dot = 0.0;
		for(i=0; i<N; i+=4)
		{
			dot += X[i]*X[i]+X[i+1]*X[i+1]+
				X[i+2]*X[i+2]+X[i+3]*X[i+3];
		}
	}
	t_elaps = (double)(GetTickCount()-t_s);

	cout << "unrolled loop x 4:                         " <<
t_elaps << " msek" << endl;

	//--------Rozwijanie petli x 8
	t_s = GetTickCount();
	for(it=0; it<ntimes; it++)
	{
		dot = 0.0;
		for(i=0; i<N; i+=8)
		{
			dot += X[i]*X[i]+X[i+1]*X[i+1]+X[i+2]*X[i+2]+
				X[i+3]*X[i+3]+X[i+4]*X[i+4]+
				X[i+5]*X[i+5]+X[i+6]*X[i+6]+X[i+7]*X[i+7];
		}
	}
	t_elaps = (double)(GetTickCount()-t_s);

	cout << "unrolled loop x 8:                         " <<
t_elaps << " msek" << endl;
```

```
	//--------Rozwijanie petli z krokiem 8, prfetch - z krokiem 8
	//Krok 8 - to jest ilosc slow double w linie cache L1
	t_s = GetTickCount();
	//Ilosc slow double w linii cache o rozmiarze 64B jest 8
	for(it=0; it<ntimes; it++)
	{
		dot1 = 0.0;
		for(i=0; i<N; i+=8)
		{
		    _mm_prefetch((const char *)(&X[i+8]), _MM_HINT_T0);
		    dot1 += X[i]*X[i]+X[i+1]*X[i+1]+X[i+2]*X[i+2]+
			    X[i+3]*X[i+3]+X[i+4]*X[i+4]+X[i+5]*X[i+5]+
			    X[i+6]*X[i+6]+X[i+7]*X[i+7];
		}
	}
	t_elaps = (double)(GetTickCount()-t_s);

	cout << "prefetching loop: unroll x 8  prefetch x 8  " <<
t_elaps << " msek" << endl;

	//---------------SSE2----------------------//
	__m128d c1, c2, c3, c4, sum;
	__declspec(align(16)) double res[2];

	t_s = GetTickCount();
	// Ilosc slow double w linii cache o rozmiarze 64B jest 8
	for(it=0; it<ntimes; it++)
	{
		sum = _mm_setzero_pd();

		for(i=0; i<N; i+=8)
		{
		    _mm_prefetch((const char *)(&X[i+8]), _MM_HINT_T0);
		    //load X[i], X[i+1]   to c1
		    c1 = _mm_load_pd(&X[i]);
		    //load X[i+2], X[i+3] to c2
		    c2 = _mm_load_pd(&X[i+2]);
		    //load X[i+4], X[i+5] to c3
		    c3 = _mm_load_pd(&X[i+4]);
		    //load X[i+6], X[i+7] to c4
		    c4 = _mm_load_pd(&X[i+6]);
		    //c1 <- c1*c1
		    c1 = _mm_mul_pd(c1, c1);
		    //sum = sum + c1*c1
		    sum = _mm_add_pd(sum, c1);
		    //c2 <- c2*c2
		    c2 = _mm_mul_pd(c2, c2);
		    //sum = sum + c2*c2
		    sum = _mm_add_pd(sum, c2);
		    //c3 <- c3*c3
```

```
                c3 = _mm_mul_pd(c3, c3);
                //sum = sum + c3*c3
                sum = _mm_add_pd(sum, c3);
                //c4 <- c4*c4
                c4 = _mm_mul_pd(c4, c4);
                //sum = sum + c4*c4
                sum = _mm_add_pd(sum, c4);
            }
            _mm_store_pd (res, sum); //unload res <- sum
            dot1 = res[0]+res[1]; //finally: dot1 = res[0]+res[1];
      }
      t_elaps = (double)(GetTickCount()-t_s);

      cout << "SSE2: x4  prefetch x 8                    " <<
t_elaps << " msek" << endl;

      if((dot-dot1)*(dot-dot1) >= 1.0e-16)
      {
            cout << "Error: dot = " << dot << "  dot1 = " << dot1
<< endl;
      }

      _aligned_free(X);
      X = NULL;

      system("pause");

      return 0;

}
```

Załącznik 2. Kod programu DGEMM 1989 (J. Dongarra)

```
#define BLOCKSIZE 64

static int MIN(int i, int j);

void dgemm(int M, int N,int K, double *A, double *B, double *C)
/*============================================================

 A - N x K
 B - K x M
 C - N x M

 A, B, C są umieszczone kolumna po kolumnie

 ============================================================*/
{
      int i, idepth, ii, ilen, ispan, j, jdepth, jj;
      int jlen, jspan, l, ll, lspan;
```

```
int mb, nb, kb;
mb = BLOCKSIZE;
nb = mb;
kb = BLOCKSIZE;

register double t11, t12, t21, t22;
double CH[BLOCKSIZE][BLOCKSIZE];

int ind_A, ind_B, ind_C;

for(l=1; l<=K; l+=kb)
{
       lspan = MIN(kb, K-l+1);
       for(i=1; i<=M; i+=mb)
       {
              idepth = 2;
              ispan = MIN(mb,M-i+1);
              ilen = idepth*(ispan/idepth);

              //kopiowanie bloku macierzy A i jego
              // przepakowanie
              for(ii=i; ii<=i+ispan-1; ii++)
              {
                     for(ll=l; ll<=l+lspan-1; ll++)
                     {
                            ind_A = N*(ll-l)+(ii-1);
                            CH[ll-l+1][ii-i+1] = A[ind_A];
              }
              }

              for(j=1; j<=N; j+=nb)
              {
                jdepth = 2;
                jspan = MIN(nb,N-j+1);
                jlen = jdepth*(jspan/jdepth);
                for(jj=j; jj<=j+jlen-1; jj+=jdepth)
                {
                  for(ii=i; ii<=i+ilen-1; ii+=idepth)
                  {
                     t11 = t12 = t21 = t22 = 0.0;
                     for(ll=l; ll<=l+lspan-1; ll++)
                     {
                       //point to B[ll, jj]
                       ind_B = K*(jj-1)+(ll-1);
                       t11 += CH[ll-l+1][ii-i+1]*B[ind_B];
                       t21 += CH[ll-l+1][ii-i+2]*B[ind_B];
                       t12 += CH[ll-l+1][ii-i+1]*B[ind_B+1];
                       t22 += CH[ll-l+1][ii-i+2]*B[ind_B+1];
                     }

                     ind_C = N*(jj-1)+(ii-1);
                     C[ind_C] += t11;
```

```
      C[ind_C+1]   += t21;
      C[ind_C+N]   += t12;
      C[ind_C+N+1] += t22;
    }//koniec pętli ii
    if(ilen < ispan)
    {
      for(ii=i+ilen; ii<=i+ispan-1; ii++)
      {
        t11 = t12 = 0.0;
        for(ll=l; ll<=l+lspan-1; ll++)
        {
          //point to B[ll, jj]
          ind_B = K*(jj-1)+(ll-1);
          t11 += CH[ll-l+1][ii-i+1]*B[ind_B];
          t12 += CH[ll-l+1][ii-i+1]*B[ind_B+K];
        }
        ind_C = N*(jj-1)+(ii-1);
        C[ind_C]   += t11;
        C[ind_C+N] += t12;
      }
    }
  }// koniec pętli jj
  if(jlen < jspan)
  {
    for(jj=j+jlen; jj<=j+jspan-1; jj++)
    {
      for(ii=i; ii<=i+ilen-1; ii+=idepth)
      {
        t11 = t21 = 0.0;
        for(ll=l; ll<=l+lspan-1; ll++)
        {
          // point to B[ll, jj]
          ind_B = K*(jj-1)+(ll-1);
          t11 += CH[ll-l+1][ii-i+1]*B[ind_B];
          t21 += CH[ll-l+1][ii-i+2]*B[ind_B];
        }

        ind_C = N*(jj-1)+(ii-1);
        C[ind_C]  += t11;
        C[ind_C+1]+= t21;
      }
      if(ilen < ispan)
      {
        for(ii=i+ilen; i<=i+ispan-1; ii++)
        {
          t11 = 0.0;
          for(ll=l; ll<=l+lspan-1; ll++)
          {
            //point to B[ll, jj]
            ind_B = K*(jj-1)+(ll-1);
            t11 += CH[ll-l+1][ii-i+1]*B[ind_B];
```

```
                                }

                                ind_C = N*(jj-1)+(ii-1);
                                C[ind_C] += t11;
                              }// koniec pętli ii
                            }// koniec bloku if
                          }// koniec pętli jj
                        }// koniec bloku if
                      }// koniec pętli j
                }// koniec pętli i
        }// koniec pętli l
}

int MIN(int i, int j)
{
        if(i < i)
                return i;
        else
                return j;
}
```

Załącznik 3. Przykład tworzenia wątków. Sekcji krytyczne.

```
// MultThreads.cpp : Defines the entry point for the console
application.
//          Przykład tworzenia wątków i funkcji wątku
//          Synchronizacja na podstawie sekcji krytycznych
//          synchronizuje dostep wątków do wspolnego resursu
//          - strumienia stdout.

#include "stdafx.h"
#include <iostream>
#include "windows.h"
using namespace std;

#define MAX_THREAD_NUMB 64

//prototyp funkcji wątku
DWORD WINAPI ThreadFunc(void *lpParam);

//dane przekazywane do wątku, pakujemy w strukturze
struct THREAD_DATA
{
        int ip;
};

//globalna deklaracja obiektu sekcji krytycznych
CRITICAL_SECTION cs;
```

```
int _tmain(int argc, _TCHAR* argv[])
{
	HANDLE hThread[MAX_THREAD_NUMB];
	DWORD ThreadID[MAX_THREAD_NUMB];
	THREAD_DATA tDat[MAX_THREAD_NUMB];

	int nThreads;
	int iThread;

	cout << "Input number of threads\n";
	cin >> nThreads;

	//inicjowanie obiektu sekcji krytycznych
	InitializeCriticalSection(&cs);

	for(iThread=0; iThread<nThreads; iThread++)
	{
		//wypelniamy dane, przekazywane do wątku ip
		tDat[iThread].ip = iThread;
		//tworzymy wątek potomny ip
		hThread[iThread] = CreateThread(NULL, 0, ThreadFunc,
					(void *)&tDat[iThread], 0,
					&ThreadID[iThread]);
		if(hThread[iThread] == NULL)
		{
			//Niepowodzenie - zamykamy program
			cout << "create thread error: iThread = " <<
iThread << endl;
			system("pause");
			DWORD ExitCode;
			BOOL Ret =
			GetExitCodeProcess(GetCurrentProcess(),
					&ExitCode);
			ExitProcess(ExitCode);
		}
	}

	//Wątek pierwotny powinien czekać dopuki wszystkie wątki
	//potomnę nie skończą działanie.
	//Jeśli tą instrukcje usunąć, wątek pierwotny
	//może skończyć działanie wcześniej
	//od zakończenia działania wątków potomnych.
	if(WaitForMultipleObjects(nThreads, hThread, TRUE, INFINITE)
				==  WAIT_FAILED)
	{
		cout << "WaitFor... problem "<< endl;
		system("pause");
		DWORD ExitCode;
		BOOL Ret = GetExitCodeProcess(GetCurrentProcess(),
					&ExitCode);
		ExitProcess(ExitCode);
```

```
	}

	//Wątki potomne są zakończone - niszczymy ich handles
	for(iThread=0; iThread<nThreads; iThread++)
		CloseHandle(hThread[iThread]);

	//deinicjowanie obiektu sekcji krytycznych
	DeleteCriticalSection(&cs);

	cout << "Hello from master thread\n";
	system("pause");

	return 0;
}

DWORD WINAPI ThreadFunc(void *lpParam)
/*====================================================================
Funkcja wątku
====================================================================*/
{
	THREAD_DATA *pDat = (THREAD_DATA *)lpParam;
	int ip = pDat->ip;

	//usunięcie synchronizacji powoduje niepoprawne działanie
	//funkcji biblioteki run time C\C++
	//Zakomentuj EnterCriticalSection(&cs) oraz
	//LeaveCriticalSection(&cs) i zobacz co wyjdzie
	EnterCriticalSection(&cs);    //wejscie do sekcji krytycznej
		//wyprowadzamy w strumien stdout pierwsza czesc zdania
		printf("Hello from thread ip");
		//wymuszamy odlaczenie wątku od procesora -
		//prowokujemy błąd
		Sleep(0);
		//wyprowadzamy drugą czesc zdania
		printf(" = %d\n", ip);
	LeaveCriticalSection(&cs);  //wyjscie z sekcji krytycznej

	//Wstrzymujemy wykonanie wątku o 1000 ms .
	Sleep(1000);
	return 0;
}
```

Załącznik 4. Kod programu dot_prod_tbb.

```
// dot_prod_tbb_2.cpp : Defines the entry point for the console
application.
//
```

```
#include "stdafx.h"
#include<iostream>
#include<cmath>
#include <vector>
#include "windows.h"
#include "tbb/parallel_reduce.h"
#include "tbb/blocked_range.h"
#include "tbb/task_scheduler_init.h"
#include "tbb/concurrent_vector.h"

//----------------type of data:
#define CONCURR_VECTOR
//#define VECTOR_DOUBLE
//#define DOUBLE

using namespace tbb;
using namespace std;

//klasa-szablon dla procedury "parallel reduce" z TBB
template <typename T>
class DotProd
{
public:
	double dot;
	T &a;
	T &b;
	DotProd(T &aa, T &bb):a(aa),b(bb) { dot=0.0; }
	DotProd(DotProd &dp, split):a(dp.a),b(dp.b) { dot=0.0; }
	void operator() (const blocked_range<int>& range)
	{
		double tmp = dot;
		blocked_range<int>::const_iterator ita;

		for(ita = range.begin(); ita != range.end(); ++ita)
		{
			tmp += a[ita]*b[ita];
		}
		dot = tmp;
	}

	void join(DotProd<T> &rhs) {dot += rhs.dot;}
};


//funkcja-szablon wspiera różne typy danych
template <typename T>
double ParallelDotProd(T &x, T &y, size_t n, size_t np)
/*============================================================
x - wector x
y - wector y
n - rozmiar zadania
np - ilość wątków
```

```
============================================================*/
{
	//konstruowanie obiektu dla procedury "parallel reduce" z TBB
	DotProd<T> ob(x, y);
	//Deklarowanie range dla procedury "parallel reduce" z TBB
	blocked_range<int> Rr(0, n, 1);
	//wywołanie procedury "parallel reduce" z TBB, która
	//przetworza object ob
	parallel_reduce(Rr, ob);
	return ob.dot;
}

int _tmain(int argc, _TCHAR* argv[])
{
#ifdef _DEBUG
	cout << "DEBUG VERSION\n";
#else
	cout << "RELEASE VERSION\n";
#endif

	int np = 0;
	size_t N;
	size_t ntimes, it;
	size_t ind;
	DWORD dwTime;
	double dot_prod;

	cout << "Input N, np, ntimes\n";
	cin >> N >> np >> ntimes;

	size_t loc_N = N/np;
	N = loc_N*np;

	cout << "N = " << N << endl;

	task_scheduler_init Init(np);

#ifdef CONCURR_VECTOR
	concurrent_vector<double> a;
	concurrent_vector<double> b;
	cout << "CONCURRENT_VECTOR" << endl;

#endif
#ifdef VECTOR_DOUBLE
	vector<double> a;
	vector<double> b;
	cout << "vector<double>" << endl;
#endif
#ifdef DOUBLE
	double *a = NULL;
	double *b = NULL;
	cout << "double" << endl;
```

```
#endif

#ifdef DOUBLE
	try
	{
		a = new double [N+1];
		b = new double [N+1];
		for(ind=0; ind<N; ind++)
		{
			a[ind] = 1.0;
			b[ind] = (double)(ind);
		}
	}
	catch(bad_alloc)
	{
		cout << "mem_alloc_err\n";
		system("pause");
		exit(1);
	}
#else
	//przygotowanie wektorów x, y, umieszczonych odpowiednio w
	//obiektach a, b
	a.reserve(N+1);
	b.reserve(N+1);

	for(ind=0; ind<N; ind++)
	{
		a.push_back(1.0);
		b.push_back((double)ind);
	}
#endif

	dwTime = GetTickCount();

	//call parallel reduction and repeat it ntimes times
	for(it=0; it<ntimes; it++)
		dot_prod = ParallelDotProd(a, b, N, np);

	cout << GetTickCount()-dwTime << " ms\n";

#ifdef DOUBLE
	delete [] a; a = NULL;
	delete [] b; b = NULL;
#else
	a.clear();
	b.clear();
#endif

	//porównywanie wyniku z wynikiem obliczeń sekwencyjnych
	double dot_prod_ctrl = 0;
	double *x, *y;
	try
```

```
	{
		x = new double [N];
		y = new double [N];
	}
	catch (bad_alloc e)
	{
		cout << "mem_alloc_err\n";
		system("pause");
		exit(1);
	}

	for(ind=0; ind<N; ind++)
	{
		x[ind] = 1.0;
		y[ind] = (double)ind;
	}

	for(ind=0; ind<N; ind++)
		dot_prod_ctrl += x[ind]*y[ind];

	if(fabs(dot_prod_ctrl-dot_prod) > 1.0e-8)
	{
		cout << "error!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n";
	}

	delete [] x;
	delete [] y;
	system("pause");
	return 0;
}
```