Programowanie równoległe

Opracowanie zagadnień

baka1408, IS2016

Treści omawiane na wykładach (źródło : notatki), a czasami nie dostępne w udostępnionych wykładach (.pdf) zaznaczone są kolorem **niebieskim** (kolor ramki zagadnienia).

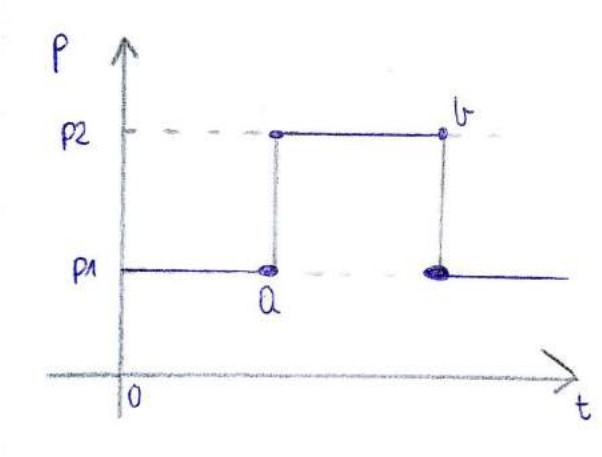
Spis treści :

I. Wstęp do obliczeń równoległych i współbieżnych.

1. Obliczenia współbieżne.

Współbieżność to sytuacja gdy czasy wykonywania procesów nachodzą na siebie. W pewnym przedziale czasu jest wykonywanych wiele procesów jednocześnie.

Jeżeli nie ma możliwości wykonania współbieżnego, wówczas mamy do czynienia z wywłaszczaniem procesów – następuje przełączenie kontekstu w miejscu **a** oraz **b** na poniższym rysunku.



W trakcie **wywłaszczania procesu** musimy zachować informację o jego przestrzeni adresowej, liczniku rozkazów, pamięci rejestrów oraz zarządzanych urządzeniach I/O.

2. Obliczenia równoległe.

Równoległość to sytuacja kiedy dwa lub więcej procesów (wątków) jednocześnie współpracuje (komunikuje się między sobą) w celu rozwiązania pojedynczego zadania.

Architektura von Neumanna – zarówno program jak i dane są umieszczane w pamięci komputera, procesor pobiera rozkazy z pamięci, rozkodowuje pobrany rozkaz i znajduje adresy argumentów, następnie pobiera dane z pamięci i wykonuje na nich operacje. Na końcu zapisuje wynik w pamięci (adresy również muszą zostać zakodowane w rozkazie).

Prawo Moore'a to twierdzenie o podwajaniu się liczby tranzystorów w pojedynczym układzie co 18 miesięcy. Dziś prawo to może być tłumaczone jako podwajanie liczby rdzeni w pojedynczym układzie (defacto liczba tranzystorów wówczas również się podwaja).

Cele obliczeń równoległych to wykorzystanie możliwości współczesnego sprzętu oraz zwiększenie mocy obliczeniowej.

3. Współczesne systemy równoległe.

Współczesne systemy równoległe możemy podzielić na dwie grupy : **procesory wielordzeniowe** oraz **systemy równoległe** (tworzone z procesorów ogólnego przeznaczenia).

W skład procesów wielordzeniowych wchodzą klasyczne procesory multi-core ogólnego przeznaczenia oraz maszyny masowo wielordzeniowe (many-core), czyli np. procesory graficzne lub procesory Xeon Phi.

Równoległość może być realizowana na wielu poziomach :

Równoległość na poziomie pojedynczego rozkazu realizowana jest sprzętowo bez udziału programisty – potokowość, superskalarność.

Równoległość na poziomie sekwencji rozkazów lub pętli realizowana jest przez programistę i twórcę kompilatorów. Model ten wykorzystuje wątki oraz procesy, następuje podział kolejno wykonywanych rozkazów pomiędzy procesory lub podział zadania obliczeniowego na podzadania dla konkretnych procesorów.

Równoległość programów realizowana jest przez system operacyjny i polega na jednoczesnym wykonywaniu wielu programów przez system.

4. Model fork-join.

Funkcja fork() w systemach Unix/Linux służy do wykonywania jednego ciągu rozkazów na w ramach wielu procesów w sposób współbieżny.

Każdy proces musi składać się z własnego ciągu rozkazów (w przypadku tego modelu, jest to ten sam, powielony ciąg rozkazów), stosu, przestrzeni adresowej, oraz elementów dodatkowych takich jak np. kontekst procesu.

Poniżej schemat przedstawiający schemat pamięci podczas wykonania współbieżnego za pomocą funkcji fork() :

W **przestrzeni adresowej** przechowywane są dane adresowe komórek pamięci, dzięki czemu możliwe jest przypisanie zmiennych do danej komórki.

W przypadku ‘kopiowania’ przestrzeni adresowej dla wielu procesów (jak na rysunku) rozkaz typu ‘b = 2’ wykonany będzie na wszystkich procesach – zmienna będzie więc miała wartość równą ‘2’ w dwóch miejscach pamięci.

Polecenia powłoki Linuxa dotyczące modelu fork-join :

- **`pid_t fork(void)`**

- **`int execv(cont char *filename, char *const argv[])`** : Polecenie to w swej istocie przepisuje proces z jednego kodu binarnego na inny.

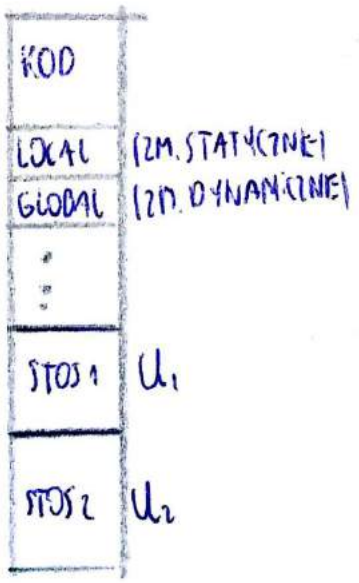
- **`pid_t wait(pid_t pid, int *status)`** : Polecenie nakazuje zawiesić wykonanie podane procesu dopóki jeden z wątków potomnych nie zostanie zakończony (lub wszystkie, zależnie od argumentu).

5. Pojęcie wątku.

Pojęcie wątku zostało wprowadzone dla zwiększenia szybkości przełączania kontekstu dzięki połączeniu przestrzeni adresowych, która jest dla wszystkich wątków wspólna. Ułatwia to również komunikację przy użyciu pamięci wspólnej.

Licznik rozkazów, stos oraz rejestr są unikalne dla każdego z wątków. Poniżej schemat struktury pamięci każdego z wątków :

Oraz schemat całej struktury pamięci, z uwzględnieniem wielu stosów dla wielu wątków :



Proces a wątek – każdy proces posiada własną przestrzeń adresową, ciąg rozkazów wątku głównego i ewentualnie innych wątków, stos wątku głównego oraz dodatkowe elementy tworzące min. kontekst procesu. Wątki z kolei posiadają własne ciągi rozkazów, stosy oraz niektóre z elementów tworzących kontekst procesu.

Tworzenie wątku w systemach Linux odbywa się poprzez funkcję :

```
int clone(int (*fn)(void*), void *child_stack, int flags, void *arg)
```

Argumenty to kolejno : funkcja, którą ma wykonać wątek, lokacja stosu wykorzystywanego przez tworzony wątek, flagi zawierające ilość wysyłanych sygnałów zakończenia do wątku macierzystego oraz argumenty przesyłane do wątku.

II. Biblioteka Pthreads – niskopoziomowe mechanizmy równoległości.

1. Wątki Pthreads.

Bibliotekę Pthreads specyfikuje standard POSIX.

Możliwe operacje na wątkach w ramach biblioteki to :

- określanie atrybutów (odłączalność, adres i rozmiar stostu, czas życia i inne)
- tworzenie i uruchamianie
- porównywanie identyfikatorów (*pthread_equal*, *pthread_self*)
- zabijanie i przesyłanie sygnałów (*pthread_cancel, pthread_kill*)
- odłączanie od procesu potomnego (*pthread_detach*)
- oczekiwanie na zakończenie (*pthread_join*)

2. Komunikacja między wątkami.

SPMD (single program, multiple data) to podstawowy paradygmat programowania równoległego. Każdy wątek realizuje ten sam program (pozyskany z tego samego pliku binarnego). Mechanizm tworzenia wątków musi zapewnić natomiast dla każdego procesu jego własne dane. Paradygmat ten zakłada udostępnienie programiście unikatowego identyfikatora dla każdego procesu, dzięki któremu wątek może być identyfikowany w celu otrzymania własnych danych.

Komunikacja między wątkami odbywa się za pomocą pamięci wspólnej (w ramach zmiennych globalnych).

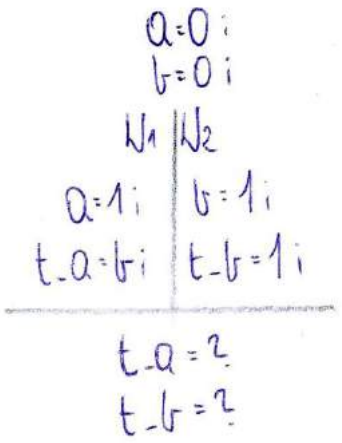
W celu realizacji modelu SPDM można do wielu wątków wykonujących ten sam program przesłać inne dane jako argumenty procedury. Częstym przypadkiem jest również przesyłanie ID wątku, dzięki któremu można określić zakres obowiązków danego wątku.

W modelu Pthreads **zaleca się nie korzystać** z wartości zwracanych przez funkcje wykonywane przez wątki.

Komunikacja poprzez pamięć wspólną ma **charakter rozgłaszania** – zmiana wartości globalnych przez pojedynczy wątek oznacza możliwość jej odczytania przez inne wątki. Potrzebna jest więc synchronizacja działania wątków. W obliczeniach sekwencyjnych zagadnienie synchronizacji jest problemem głównie wydajności, w przypadku równoległych dochodzą jeszcze kwestie poprawności danych.

3. Modele spójności pamięci.

Spójność sekwencyjna gwarantuje, że wszystkie operacje wykonywane będą z taką samą kolejnością i ich lokalne uporządkowanie powinno być zachowane w każdym z procesów. Paradoksem jest fakt, że współczesne procesory nie spełniają nawet takiego prostego modelu spójności. Dzieje się tak z uwagi na optymalizacje, która uniemożliwia deterministyczne stwierdzenie kolejności wykonywania operacji. Przykład poniżej :



W przypadku kiedy t_a = 0 oraz t_b = 0, wówczas rozwiązanie jest niedeterministyczne – out of order execution. W innych przypadkach (jeżeli model spójności sekwencyjnej jest spełniony – założenie) wykonanie jest prawidłowe.

4. Problemy współbieżności.

Podstawowym problemem jest **wyścig** (race 8onditio) w dostępie do danych. Ma on miejsce kiedy dwa lub więcej wątków rywalizują o dostęp do danych, w skutek czego nie możliwe

jest deterministyczne określenie rezultatu – jeden z wątków może napisać zmienną zanim drugi zdąży odczytać lub zmodyfikować jej wartość.

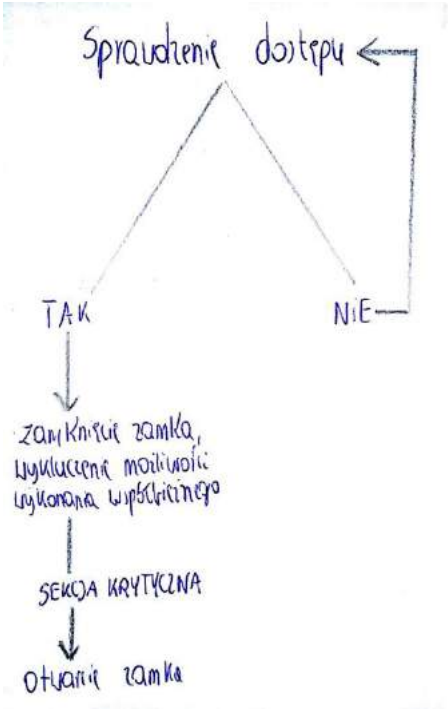
Synchronizacja można być realizowana programowo (bariera, sekcja krytyczna, operacje atomowe) lub sprzętowo (komputery macierzowe, architektura Sindi – obliczenia wektorowe).

Wzajemne wykluczanie to sposób na osiągnięcie synchronizacji poprzez stosowanie sekcji krytycznej z protokołami wejścia i wyjścia a w przypadku Pthreads realizowane przez mechanizm zamków.

Do pożądanych skutków synchronizacji należą uczciwość podziału czasu, bezpieczeństwo oraz żywotność programów, natomiast negatywne skutki to możliwość zakleszczenia lub zagłodzenia.

5. Wzajemne wykluczanie wątków.

Schemat realizacji sekcji krytycznej :



Podstawowy zamek programowy (abstrakcyjny – nie związany z żadnym mechanizmem równoległości) można przedstawić jako :

```
int lock = 0;
while (lock != 0) {/*aktywne czekanie*/ }
lock = 1;   // otwieramy zamek
critical_section_task();      // sekcja krytyczna
lock = 0;         // zamykamy zamek
```

W tym przypadku głównym problemem jest brak bezpieczeństwa procedury, zapewniania żywotności a także zużycie zasobów spowodowane aktywnym czekaniem. Jak rozwiązać te problemy? Poprzez **bardziej rozbudowane algorytmy** wejścia do sekcji krytycznej, **wsparcie sprzętowe** (odpowiednie rozkazy procesora), **wsparcie systemowe** lub **odpowiednie API** (które wykorzystuje zaimplementowane procedury systemowe).

6. Mechanizmy wzajemnego wykluczania.

W przypadku Pthreads problem rozwiązują tak zwany **mutexy** od (mutual exclusion), czyli mechanizmy zamków. Należy (jeśli to możliwe) minimalizować użycie zamków poprzez modyfikację algorytmów a jeżeli zakładamy zamki, musimy to robić we właściwej kolejności, to znaczy tak by unikać zakleszczeń. Każdy zamek musi być też we właściwym miejscu otwierany.

Alternatywą do stosowania sekcji krytycznej jest projektowanie **współbieżnych struktur danych**, czyli takich do których dostęp jest bezpieczny z poziomu wielu wątków. Mechanizm ten wykorzystywany jest szczególnie w przypadku obiektowych, wysokopoziomowych języków, których biblioteki posiadają gotowe struktury umożliwiające dostęp współbieżny. Przykładem takiego podejścia jest również model **pamięci transakcyjnej**, gdzie dostęp do danych wymaga rozpoczęcia transakcji. Jeżeli inny wątek w trakcie jej trwania ukończył swoją transakcję, to ten pierwotny wraca do stanu początkowego (tz. od nowa rozpoczyna transakcję i daną operację).

Kolejnym mechanizmem są **semafory** – teoretyczna konstrukcja umożliwiająca poprawne rozwiązanie problemu wzajemnego wykluczania. Jest to obiekt (zmienna globalna) na, której można dokonywać dwóch niepodzielnych, wykluczających się operacji, które mają charakter atomowy. Zwyczajowo są one nazywane P (wait) oraz V (signal).

Przykładowa abstrakcyjna implementacja semafora :

```
void P(int miejsca){ if (miejsca > 0) miejsca--; else uśpij_wątek(); }
void V(int miejsca){ if (ktoś_śpi()) obudź_wątek(); else miejsca++; }
```

Uczciwość dostępu zależy od implementacji funkcji V, odpowiadającej na warunek wybudzenia wątku. Zmienna miejsca opisuje ilość pozostałych dostępów do zasobu (dla sekcji krytycznej równa 1 na początku).

Działanie semaforów można przedstawić na przykładzie problemu **ucztujących filozofów** lub **czytelników i pisarzy.** Więcej o tym zagadnieniu można znaleźć w rozdziale 4.

7. Proces tworzenia oprogramowania równoległego.

Podczas tworzenia programów równoległych istnieją dwa istotne kroki – **wykrycie dostępnej współbieżności** w trakcie realizacji programu oraz **określenie koniecznej synchronizacji** (Pthreads) lub wymiany komunikatów (MPI) pomiędzy wątkami lub procesami. Jednym z najważniejszych wymagań stawianych programom równoległym **jest przenośność oraz skalowalność** – możliwość uruchomienia na maszynie o dowolnej liczbie procesorów.

Podstawowe zadania do realizacji w przypadku tworzenia programów równoległych :

- **dekompozycja** (podział) zadania obliczeniowego na podzadania
- **odwzorowanie zadań** na procesy i wątki (idąc dalej na węzły, multi-procesory, rdzenie)
- **dystrybucja** (przydział) danych pomiędzy elementy pamięci związane z elementami przetwarzania
- **określenie koniecznej wymiany** pomiędzy procesami (wątkami) i odwzorowanie jej na sieć połączeń między procesorami (rdzeniami)
- **określenie koniecznej synchronizacji** pomiędzy zadaniami

Usystematyzowaną **metodologią** tworzenia oprogramowania równoległego jest **PCAM**. Kolejne kroki wywodzą się od liter : P (partition), C (communicate), A (agglomerate – analiza wariantów podziału), M (map – uwzględnienie ostatecznego odwrozrowania). Pierwsze dwa kroki stanowią etapy tworzenia, natomiast pozostałe to etapy optymalizacyjne.

Wyróżniamy też wiele **wzorców** architektonicznych wykorzystywanych w procesie tworzenia **oprogramowania równoległego** : master-slave/manager-worker divide and conquer/recursive splitting/fork-join, loop parallesim, pipelining (przetwarzanie potokowe), software agents/actors/discrete event, map-reduce oraz wiele innych. *Więcej o wzorcach programowania równoległego można znaleźć w dodatku dołączonym do tego opracowania.* Są to ogólne wskazówki rozwiązania problemu dekompozycji zadania na pod-zadania, przydzielane każdemu wątkowi/procesorowi.

Możliwe rodzaje dekompozycji (podziału zadania na pod zadania) :

- Podział ze względu na **funkcje** (functional decomposition) : Każdy wątek otrzymuje jakąś funkcję do wykonania, która realizuje zupełnie odmienne zagadnienie pod względem logicznym. Przykładem są nowoczesne, złożone aplikacje, gdzie jeden wątek realizuje połączenie z serwerem, drugi odpowiada za interfejs, trzeci za obliczenia itp.

- Podział **struktury danych** (data decomposition) : Każdy wątek otrzymuje ‘część’ tablicy, listy lub innej sturktury danych (najczęściej poprzez otrzymanie indeksów początkowych oraz końcowych). Wówczas każdy wątek równolegle dokonuje obliczeń na podanej części, a po zakończeniu obliczeń wątek główny po odczekaniu na zakończenie prac przez wszystkie wątki (mechanizm bariery) scala wyniki i podaje końcowy rezultat. Przykładem są zagadnienia sortowania tablic czy rozwiązywania układów równań.

- Podział **w dziedzinie problemu** (domain decomposition) : Polega na podziale zadania w oparciu o wiedzę o dziedzinie rozwiązywanego problemu. Każdy wątek otrzymuje część zadania w oparciu o podział zadania na ‘naturalne podzadania’. Przykładem może być podział geometryczny w przypadku symulacji zjawisk fizycznych, rozwiązywanie układów równań, lub też (bardziej przyziemnie) podział obliczeń związanych z ruchami gracza w przypadku gry wieloosobowej – każdy wątek otrzymuje identyfikator gracza i wykonuje obliczenia związane tylko z tym konkretnym obiektem.

- Podział **złożony** : Podział taki ma cechy zarówno podziału funkcjonalnego jak i podziału danych, przykładem jest przetwarzanie potokowe sekwencji obrazów. Zależnie od danych przydzielanych do wątku, różne są również wykorzystywane przez niego funkcje.

8. Metodologie programowania równoległego.

SIMD (Single instruction, multiple data) – przetwarzanych jest wiele strumieni danych w oparciu o pojedynczy strumień rozkazów. Architektura charakterystyczna dla komputerów wektorowych.

MIMD (Multiple instruction, multiple data) – przetwarzanie równoległe zarówno na poziomie danych jak i instrukcji. W przypadku super-komputerów wiele procesorów SIMD połączonych jest w klaster obliczeniowy MIMD. W przypadku wielordzeniowych procesorów domowych, każdy rdzeń realizuje osobny potok instrukcji.

Ze względu na mechanizmy programowe rozróżniamy następujące **metodologie programowania równoległego** :

- **SPMD** : Każdy proces realizuje ten sam program (ten sam plik binarny). Model jest bardzo uniwersalny, znacznie częściej wykorzystywany od MPMD. Dla architektur SIMD oznacza wykonywanie tego samego kodu z synchronizacją sprzętową natomiast dla MIMD oznacza, że każdy wątek może realizować ten sam kod otrzymując przydzieloną porcję danych, na jakich operuje.

- **MPMP** : Każdy proces otrzymuje różny kod (z różnych plików). Konieczna jest komunikacja pomiędzy procesami a implementacja najczęściej dokonywana jest w modelu z pamięcią rozproszoną (MPI). Architektura sprzętu opiera się zawsze na tym modelu.

Kolejny możliwy podział, ze względu na konkretne mechanizmy programowe, niejako tożsamy z rodzajami dekompozycji – równoległość zadań (jak dekompozycja funkcji), równoległość wykonania pętli (jak dekompozycja struktur), równoległość sterowania (na podstawie swojego indeksu wątek ustala, którą ścieżkę w programie powinien realizować), oraz równoległość danych (np. w języku HPC).

Alternatywami dla modelu programowania równoległego mogą być : programowanie sekwencyjne ze zrównolegleniem niejawnym (poprzez układ procesora superskalarnego lub automatyczny kompilator zrównoleglający), programowanie w językach równoległości danych, programowanie w modelu z pamięcią wspólną, programowanie z przesyłaniem komunikatów (MPI), programowanie w modelu przerzucenia obliczeń na układ wspomagający (koprocesor, akcelerator) czy programowanie w modelach hybrydowych.

9. Monitory i zmienne warunku.

Problem producentów i konsumentów – jedna grupa procesów produkuje dane, druga je konsumuje. Pytanie – jak zagwarantować sprawny (bez zakleszczeń i zagłodzeń) przebieg tej procedury?

Problem czytelników i pisarzy – jedna grupa produkuje dane, druga je odczytuje. Produkować jednocześnie może tylko jeden proces, natomiast odczytywać może wiele procesów na raz.

Zmienne warunku (condition variables) są zmiennymi wspólnymi dla wątków, które służą do identyfikacji uśpionych wątków. Można wykonywać na nich operacje :

- init : tworzenie zmiennej warunku.
- wait : usypianie wątku w miejscu identyfikowanym przez zmienną warunku, wątek czeka do momentu gdy jakiś inny wątek wyśle odpowiedni sygnał budzenia.
- signal : sygnalizowanie budzenia pierwszego w kolejności wątku, który oczekuje na podanej zmiennej warunku.
- broadcast : sygnalizowanie budzenia wszystkich wątków oczekujących na zmiennej warunku.

Monitor to moduł posiadający stan (atrybuty – stałe i zmienne) oraz zachowanie (metody – procedury i funkcje). Metody monitora dzielą się na publiczne oraz prywatne. Podstawową zasadą działania monitora jest realizowanie sekcji krytycznej na poziomie obiektu – jeżeli jakiś proces rozpoczął realizację publicznej metody monitora to żaden inny proces nie może rozpocząć realizacji żadnej innej publicznej metody tego monitora. Wewnątrz monitora może znajdować się tylko jeden proces/wątek.

Za realizację wykluczania dostępu do monitora odpowiada środowisko w ramach, którego funkcjonuje monitor. Wątek wywołujący publiczną metodę monitora otrzymuje do niego dostęp, jeżeli nie ma żadnego innego wątku wewnątrz monitora, lub jest ustawiany w kolejce uśpionych oczekujących wątków. Po opuszczeniu monitora przez wątek, system wznawia działanie pierwszego wątku w kolejce (jest tylko jedna kolejka).

Można w dużym uproszczeniu powiedzieć, że **monitor jest obiektem**, do którego dostęp może uzyskać jednocześnie jedynie jeden wątek/proces. Wzajemne wykluczanie istnieje więc

na poziomie całej klasy. W środowiskach obiektowych (np. Java) zdarza się, że każdy obiekt jest monitorem i posiada właściwe dla niego funkcje.

Poniżej **przykład abstrakcyjnej realizacji klasy monitora** dla problemu czytelników i pisarzy :

Jeżeli ktokolwiek czeka w kolejce, lub w bibliotece jest pisarz, wówczas wątek jest usypiany. Po obudzeniu lub jeżeli powyższy warunek nie został spełniony czytelnik przystępuje do czytania i zwiększa liczbę aktywnych czytelników. Po wyjściu, jeżeli nie ma żadnego czytelnika w kolejce, to budzony jest pisarz.

Od implementacji protokołu wyjścia pisarza zależy uczciwość algorytmu, można nadać priorytet bądź to pisarzom bądź czytelnikom.

```
Monitor
{
      czytelnicy = 0;
      pisarze = 0;

      chce_czytać()
      {
            if (pisarze > 0 lub kolejka_czytelników > 0)
                  uśpij(kolejka_czytelników);
            czytelnicy++;
            obudź(kolejka_czytelników);
      }

      kończe_czytanie()
      {
            czytelnicy--;
            if (czytelnicy = 0)
                  obudź(kolejka_pisarzy);
      }

      chce_pisać()
      {
            if (pisarze + czytelnicy > 0)
                  uśpij(kolejka_pisarzy);
            pisarze++;
      }

      kończe_pisanie()
      {
            pisarze--;
            if (kolejka_czytelników > 0)
                  obudź(kolejka_czytelników);
            else
                  obudź(kolejka_pisarzy)
      }
}
```

W standardzie POSIX istnieją również **programowe zamki odczytu i zapisu**, które mogą rozwiązać problem czytelników i pisarzy. Można więc zezwolić danych wątkom tylko na odczyt lub tylko na modyfikację zmiennych w sekcji krytycznej.

III. Równoległość w językach obiektowych.

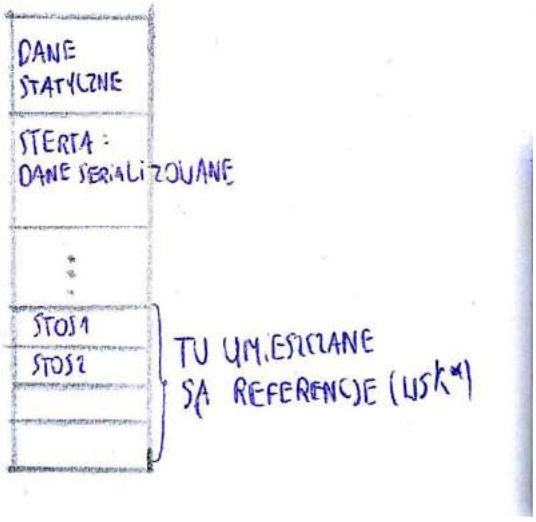
1. Model współbieżności w Java.

Współbieżność w Javie opiera się na **zarządzaniu wątkami**, **operacjach atomowych**, **mechanizmach synchronizacji** oraz **klasach kontenerowych** przystosowanych do używania współbieżnego (BlockingQueue itp.).

W językach obiektowych ryzyko i problemy dotyczące atrybutów obiektów są takie same jak w przypadku zmiennych globalnych w językach niższego poziomu – metody obiektów mogą być wywoływane przez różne wątki. Wymaganiem dla każdej projektowanej klasy, której obiekty mogą być wykorzystywane przez wiele wątków jest 'bezpieczeństwo wątkowe'.

Wątki w Javie są odwzorowywane na wątki systemowe poprzez maszynę wirtualną Javy (JVM). Odwzorowanie to nie jest jednak bezpośrednie, nie mamy kontroli nad dokładną liczba i atrybutami wątków systemowych tworzonych przez JVM.

Poniżej struktura pamięci zarządzanej przez JVM podczas wykonania programu Javy :



Pośrednim rozwiązaniem na ochronę współbieżności może być również wykorzystanie klas niezmiennych (immutable). W przypadku takich klas, atrybuty ich są niezmienne, klasy te są więc bezpieczne wątkowo. Podejście takie nie jest jednak często możliwe.

2. Synchronizacja w Java.

Najprostszym sposobem na synchronizowanie działania dowolnej metody danego obiektu jest oznaczenie jej jako **synchronizowaną** (synchronised). Wówczas jej realizacja przez jakiś wątek uniemożliwia jej wywołanie przez inne wątki (są ustawianie w kolejce). Mechanizm ten opiera się na istnieniu wewnętrznych zamków monitorowych, zwanych monitorami – wątek rozpoczynający wywołanie metody zamyka taki zamek i otwiera go dopiero po ukończeniu pracy z obiektem.

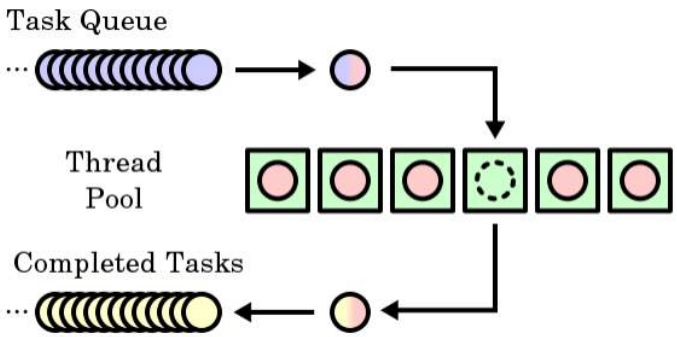
W podobny sposób działa **blok synchronised**, który ogranicza wywołanie sekcji krytycznej wewnątrz bloku poprzez pojedynczy mutex. Blok taki jako argument posiadać musi dowolny obiekt, którego mutex (wewnętrzny zamek) będzie używany jako 'blokada'.

Kolejna metoda synchronizacji wykorzystuje klasę z pakietu java.util.concurrency, a konkretniej interfejs **Lock**, który posiada standardowe funkcje mutexa i kilka specyficznych implementacji, a także wzbogaca zamki o możliwość użycia funkcji *trylock()* niedostępnej w poprzednio wymienionych implementacjach.

3. Obliczenia masowo równoległe.

Przedstawione powyżej metody są optymalne dla przetwarzania wielowątkowego z małą liczba zadań, dla masowej równoległości Java posiada specjalne mechanizmy : pule wątków oraz interfejsy wykonawców.

Konkretna pula wątków (czyli implementacja interfejsu puli) realizuje zarządzanie dostarczonymi wątkami (implementującymi interfejs Runnable lub Callable), które obejmuje uruchamianie, zatrzymywanie, sprawdzanie stanu oraz inne niezbędny operacje. Taki sposób zarządzania wątkami nakłada mniejszy narzut, niż 'ręczne' tworzenie wątków i ich uruchamianie za pomocą klasy Thread.



Powyższy przykład ilustruje działanie puli, zależnie od ustanowionej aktywnej puli wątków (np. 6) mechanizm automatycznie uruchamia i kończy działanie zadań (wątków) tak, by przez cały czas działała ich określona z góry ilość.

Różnica pomiędzy interfejsami **Runnable** oraz **Callable** polega na tym, że pierwszy nie daje możliwości zdefiniowania zadań zwracających wynik, a drugi tak. Do przekazywania wyników oraz sprawdzania czasu zakończenia wykonania obiektów Runnable służą obiekty **Futures**, które będąc sparametryzowane typem zwracanego wyniku (to znaczy, że z góry należy określić jakiego typu dane będą one przechowywać, tak samo jak w przypadku obiektów klas kontenerowych – *ArrayList*, *HashMap* i tym podobnych) udostępniają funkcję do anulowania, sprawdzania zakończenia oraz zwrotu wyniku po zakończeniu działania zadania.

Work stealing pozwala na efektywne dzielenie dużej liczby zadań pomiędzy wątki – gdy jeden z wątków jest 'wolny', czyli nie posiada żadnych zadań do realizacji, zabiera on zadania innym wątkom, które mają ich aktualnie nadmiar (zakolejkowane). Decyzja o takim przydziale podejmowana jest tylko wtedy, gdy wątek 'kradnący' miałby pozostać bezczynny.

IV. Programowanie systemów z pamięcią wspólną – OpenMP.

1. Cechy OpenMP.

OpenMP bazuje na dyrektywach kompilatora, jest więc przenośne pomiędzy systemami. Obowiązuje serializacja wykonania, kod sekwencyjny jest zrównoleglony. Model zrównoleglenia to *fork-join()*. OpenMP udostępnia prócz dyrektyw dla kompilatora również zmienne środowiskowe oraz funkcje biblioteczne.

Poniżej rysunek przedstawiający model zrównoleglenia OpenMP :

Najważniejszymi dyrektywami są **dyrektywy podziału pracy**, występujące w obszarze równoległym i stosowane do podziału poleceń dla poszczególnych wątków.

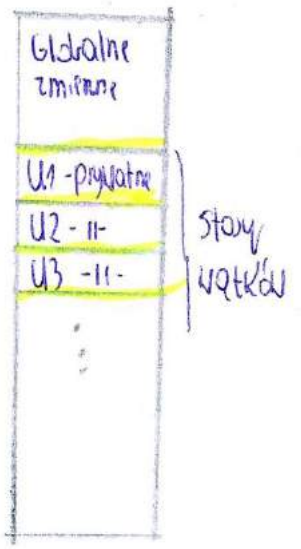
Każda dyrektywa posiada zestaw dopuszczalnych klauzul, które określają sposób traktowania zmiennych przez wątki w obszarze równoległym. Składnia każdej z dyrektyw wygląda następująco :

```
#pragma omp nazwa_dyrektywy lista_klauzul
```

Wątki systemowe tworzone są w momencie uruchomienia programu i zadania są im przedzielane dopiero po wejściu sterowania do sekcji równoległej. Liczba wątków domyślnie ustanowiona jest przez zmienną środowiskową '*OMP_NUM_THREADS*'. Można dokonać zmiany tej zmiennej, albo zadeklarować liczbę wątków w programie za pomocą klauzuli lub wywołania procedury bibliotecznej (klauzula ma najwyższy priorytet). Istnieje również możliwość dynamicznego ustalania liczby wątków.

Domyślnie wszystkie wątki są równoważne, nie ma więc wątku głównego (master) oraz potomnych (slave). Jednak sytuacja zmienia się, kiedy mamy do czynienia z zagnieżdżonym obszarem równoległym – wówczas wątki wywołane w jednym z wątków obszaru równoległego są potomne względem niego. Sytuacje tą możemy zaobserwować na prezentowanym już wykresie.

Poniżej schemat struktury zmiennych w modelu OMP :



2. Zmienne w OpenMP.

W OpenMP mamy do wyboru kilka klauzul dotyczących współdzielenia zmiennych, zmienne mogą być więc deklarowane jako :

- *shared*, zmienna wspólna dla wszystkich wątków (wymagana synchronizacja)

- *private*, zmienna prywatna dla wątków, należy zainicjalizować przed użyciem

- *fistprivate*, zmienna prywatna, inicjowana wartością początkowa (sprzed wejścia do obszaru równoległego)

- *lastprivate*, zmienna prywatna z wartością końcową równą wartości przy wykonaniu sekwencyjnym (zostaje jako wynik pracy ostatniego z wątków)

- *threadprivate*, zmienna istnieje tylko jako prywatna zmienna wątku i jej zakres ważności jest taki sam jak zakres obszaru równoległego, ale jej wartość jest zachowywana pomiędzy obszarami równoległymi. Oznacza to, że zmienna *threadprivate* zadeklarowana w pierwszym obszarze równoległym i zainicjowana wartością ‘5’ będzie miała taką samą wartość w n-tym obszarze równoległym, dla tego konkretnego wątku (pod warunkiem, że w innych obszarach równoległych nie zostanie zmodyfikowana co do wartości).

Zmienne deklarowane bezpośrednio w obszarze równoległym są zmiennymi prywatnymi, 'jednorazowymi' wątku. Zmienne są wspólne jeżeli istnieją przed obszarem równoległym i nie zostają 'dotknięte' dyrektywami prywatności.

3. Zależności.

Zależności to wzajemne uzależnienie instrukcji nakładające ograniczenia na kolejność ich realizacji. Wyróżniamy następujące zależności :

- **Zależności zasobów**, kiedy wiele plików usiłuje uzyskać dostęp do wybranego zasobu (pliku, gniazda, urządzenia).

- **Zależności sterowania**, kiedy wykonanie danej instrukcji zależy od rezultatów poprzedzających instrukcji warunkowych.

- **Zależności danych** (przepływu), najistotniejsza pod względem analizy równoległego wykonania grupa zależności, kiedy instrukcje wykonywane w bezpośrednim sąsiedztwie czasowym operują na tych samych danych i choć jedna z instrukcji dokonuje zapisu.

Wyróżniamy następujące typy zależności danych :

- **Write after Write** (zależność wyjścia), nie jest to rzeczywisty problem, ponieważ przemianowanie zmiennych lub eliminacja instrukcji pozwala uniknąć tej zależności.

```
a = 4;
a = 12;
printf(a); // Wynik nie znany

(przemianowanie zmiennych)

a = 4;
a1 = 12;
printf(a); // Wynik '4'
```

- **Write after Read** (anty zależność), w tym przypadku również przemianowanie zmiennych pozwala na usunięcie zależności.

```
a = 4;

a = 12;
b = a;
printf(b); // Wynik nie znany

(przemianowanie zmiennych)

a = 4;

a1 = 12;
```

```
b = a;
printf(b); // Wynik '4'
```

- **Read after Write** (zależność rzeczywista), taki typ zależności uniemożliwia zrównoleglenie algorytmu, należy go przeformułować lub zastosować mechanizm synchronizacji.

```
a = 4;

a = 6;
b = a;
printf(b); // Wynik nie znany, nie da się przemianować zmiennej
```

Aliasing to pokrywanie się wartości wskaźników o różnych nazwach, przez co na pierwszy rzut oka zależność nie jest widoczna. Przykład – do wskaźników *wsk1 oraz *wsk2 przypisano adres zmiennej 'b' a następnie dokonano tożsamej jak powyżej operacji z zależnością Read after Write.

Możemy mieć również do czynienia z **zależnościami przenoszonymi w pętli** – jeżeli iteracyjnie wykonujemy przykładowo operację : **tab[i]** = tab2[i] * **tab[i-1]**, to nie mamy pewności co do kolejności wykonania iteracji, tak więc przenoszona jest zależność dla i=1,2,3…,n. Przemianowanie zmiennej *tab* na *tab1* rozwiązuje problem.

Jeżeli operowano cały czas na *wsk1, to nie widać, by *wsk2 w ogóle był zależny od *wsk1 czy zmiennej 'a', ale jednak wartością jest równy *wsk1, więc również obarczony jest zależnością rzeczywistą.

Kompilatory zrównoleglające posiadają szereg metod heurystycznych (przybliżających, szacujących) analizujących kod pod względem zależności, jednak dla praktycznie występujących problemów analiza może przekraczać możliwości sprzętowe – problem jest NP-trudny. W takim przypadku jedynym gwarantem uniknięcia zależność i uzyskania wydajnego programu jest sam programista.

Przykład analizy zależności :

```
1. x = 2y + w;
2. y = 2z – cos(w)
3. z = log(w) + y
4. x+= sin(w)
```

Rozważmy zależności poszczególnych zmiennych :

- X : zależność RaW oraz WaW. Nie możemy jej usunąć, wobec czego punkty 1 oraz 4 muszą być wykonywane przez ten sam wątek.
- Y : zależność RaW oraz WaR. Nie możemy jej usunąć, wobec czego punkty 2 oraz 3 muszą być wykonywane przez ten sam wątek.

- Z : zależność WaR w punktach 2 oraz 3, możemy przemianować zmieniając w punkcie trzecim zmienną na 'z1'.
- W : brak zależności, nie ma zapisu tej zmiennej więc zależność nie może występować.

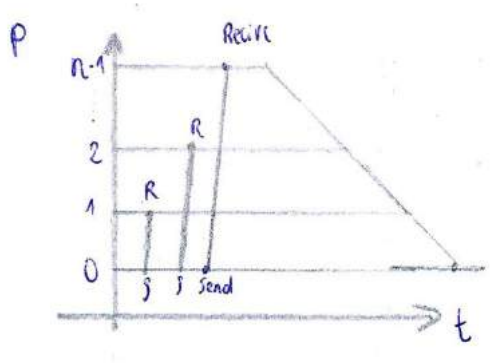
W skutek analizy ustalono, że punkty 1 i 4 oraz 2 i 3 muszą być wykonywane na dwóch, osobnych wątkach, aby możliwe było wykonanie bez mechanizmów synchronizacji.

V. Grupowe przesyłanie komunikatów – MPI.

1. Model komunikacji z przesyłaniem komunikatów.

W modelu MPI nie mamy pamięci wspólnej dla wszystkich procesów, gdyż nie ma wspólnej przestrzeni adresowej. Istnieje możliwość lokalnego debugowania MPI za pomocą komunikacji między procesami a nie maszynami, jednakże nawet wtedy każdy proces zarządza własnym obszarem pamięci.

W modelu z rozproszoną pamięcią nie ma zjawiska wyścigu, istnieje jednak konieczność poważnej modyfikacji (adaptacji) istniejących algorytmów. Poniżej przykład działania tego modelu na wykresie czasowym :



MPI realizuje paradygmat **"send-receive"**, który opiera się na wysyłaniu komunikatu oraz odbieraniu komunikatu. Model jest uniwersalny i zapewnia nieograniczoną skalowalność (rzędu milionów rdzeni)**.** Programy mogą być realizowane zarówno w metodologii SPMD jak i MPMD. Konkretne środowiska programowania w MPI zawierają także narzędzia do kompilacji i uruchamiania programów.

MPI wymaga uruchomienia deamona na maszynie (w warstwie pośredniej) oraz oprogramowania MPIexec, które przygotowuje środowisko do pracy z przesyłaniem komunikatów.

MPI do przesyłania komunikatów musi znać konfigurację interfejsu sieciowego – wymagane informacje do adres internetowy (IP) oraz liczba procesów.

2. Przesyłanie dwupunktowe (point-to-point).

Komunikator to grupa procesów zawierająca informację pozwalającą na komunikację między nimi. Można tworzyć klika różnych grup procesów, ale predefiniowany komunikator, zrzeszający wszystkie procesy to 'MPI_COMM_WORLD'.

Ranga procesu to inaczej jego identyfikator i nie ma on nic wspólnego z priorytetami czy ważnością danego procesu.

Procedury **przesyłania dwupunktowego** gwarantują :

- Postęp przy realizacji przesyłania (po prawidłowym zainicjowaniu pary send-recive co najmniej jedna z nich musi zostać ukończona).

- Zachowanie porządku przyjmowania komunikatów z tego samego źródła, o tym samym ID i w ramach jednego komunikatora. Przy odbieraniu komunikatów z różnych źródeł nie ma gwarancji co do uczciwości.

- W trakcie realizacji procedur może wystąpić błąd związany z przekroczeniem limitu dostępnych zasobów.

Wyróżniamy **procedury przesyłania blokującego** – procedura nie odda sterowania dopóki przesyłanie nie zostanie zakończone a bufory ze zmiennymi (rezultaty albo argumenty) nie zostaną bezpiecznie zwolnione oraz **procedury przesyłania nieblokującego** – procedura natychmiast zwraca sterowanie dalszym instrukcjom (wówczas wymagany jest dodatkowy argument do funkcji, pełniący role flagi statusu przesyłania).

Tryby komunikacji w środowisku MPI :

- **buforowany** (*MPI_Bsend, MPI_Ibsend)* – istnieje jawnie określony bufor używany do przesyłania komunikatów, komunikacja kończy się w momencie skopiowania komunikatu do bufora.

- **synchroniczny** (*MPI_Ssend, MPI_Issend)* – komunikacja jest zakończona kiedy zakończona zostaje procedura odbierania po drugiej stronie (w procesie odbierającym).

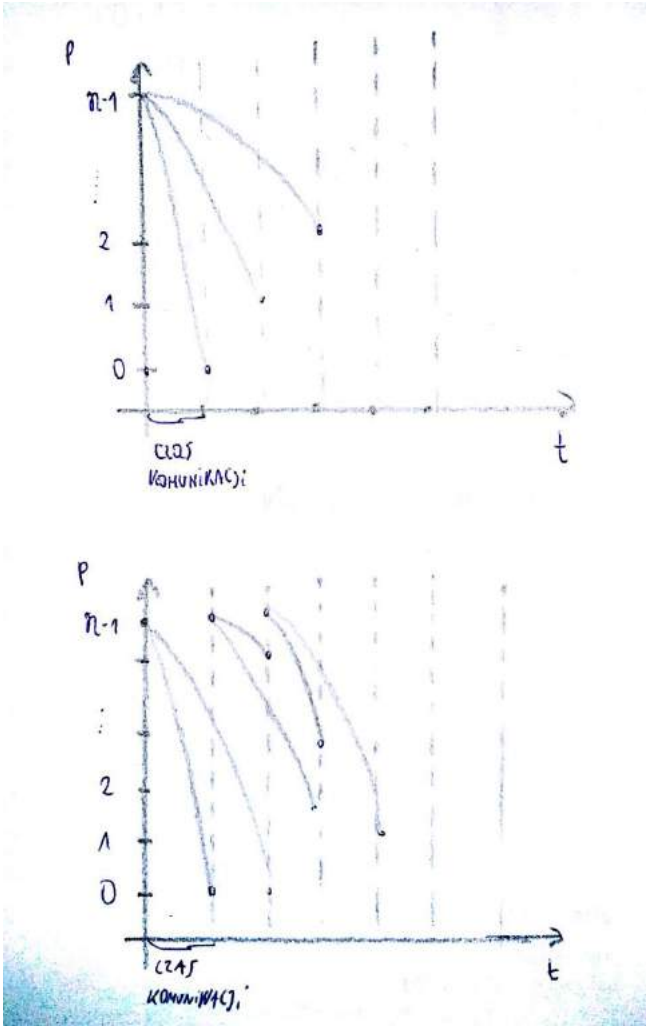
- **gotowości** (*MPI_Rsend, MPI_Irsend)* – system gwarantuje, że procedura odbierania jest gotowa, w związku z czym wysyłanie kończone jest natychmiast.

3. Grupowe rozgłaszania komunikatów.

Operacje komunikacji grupowej albo inaczej operacje globalne, to takie, w których udział biorą więcej niż dwa procesy – zbiór komunikatów przesyłany jest więc między kilkoma

procesami. Wydajność takich operacji jest bardzo różna, zależna od strategii i algorytmów wykorzystywanych w programie.

Poniżej zaprezentowana jest na wykresach czasowych optymalna oraz nieoptymalna realizacja rozgłaszania dla *p* różnych procesów. Stukrotna różnica w wydajności :



Pierwszy wykres prezentuje złą komunikacje grupową – jeden proces potrzebuje aż p 'cykli' komunikacyjnych na przesłanie wiadomości do każdego z procesów. Na drugim obrazku widzimy, że każdy proces, który otrzymał już wiadomość przesyła ją dalej, dzięki temu z każdym cyklem maleje ilość potrzebnych cykli do obsłużenia wszystkich procesów.

Wyróżniamy następujące **typy komunikacji grupowej** :

- **Rozgłaszanie** (broadcast), jeden do wszystkich : Pojedyncza zmienna 'A' w jednym z procesów jest przesyłana do podanego miejsca w pamięci wszystkich pozostałych procesów.

- **Rozpraszanie** (scatter), jeden do wszystkich : Wiele zmiennych ('A', 'B', 'C'…) w jednym z procesów jest dzielonych i przesyłanych do podanego miejsca w pamięci pozostałych procesów, w taki sposób, że proces pierwszy otrzymuje zmienną 'A', proces drugi zmienną 'B' i tak dalej.

- **Zbieranie** (gather), wszyscy do jednego : Operacja odwrotna do rozpraszania, wiele procesów, posiadających wiele zmiennych – każdy proces jedną ('A', 'B', 'C'…) przesyła swoją zmienną do jednego z procesów, który układa je wszystkie po kolei w pamięci.

- **Redukcja** (reduce), wszyscy do jednego : Operacja odwrotna do rozgłaszania, wiele procesów posiadających wiele zmiennych przesyła je do jednego z procesów, a ten wykonuje na nich podaną w argumencie operacje (suma, iloczyn itp.) a wynik umieszcza w jednym miejscu w pamięci.

- **Rozgłaszanie wszyscy do wszystkich** (All-gather) : Wszystkie procesy wysyłają do wszystkich pozostałych jedną własną zmienną. W efekcie każdy proces posiada wszystkie zmienne, od wszystkich procesów plus swoją własną zmienną. Wszystkie procesy mają więc taki sam zestaw danych.

- **Rozpraszanie wszyscy do wszystkich** (All-to-All) : Wszystkie procesy wysyłają do wszystkich pozostałych kilka (wszystkie) własne zmienne. W efekcie każdy proces posiada zbiór zmiennych jednego typu. Przykładowo : Proces pierwszy posiada zmienne 'A,B,C', po wysłaniu tych zmiennych do wszystkich procesów i odebraniu zmiennych od innych procesów jego struktura buforu odbioru jest następująca : 'A1,A2,A3', natomiast struktura drugiego procesu wygląda tak : 'B1,B2,B3', gdzie cyfry oznaczają, z którego wątku został pobrany dany argument (zmienna).

- **Redukcja wszyscy do wszystkich** (All-reduce) : Następuje rozgłaszanie, ale zamiast przekazywać każdemu procesowi zestaw zmiennych zostaje wykonana na nich operacja i przekazywany jest jej wynik.

Argumentami procedur komunikacji grupowej są bufory z danymi do wysłania i (jeżeli to konieczne) bufory na dane odbierane. Wszystkie powyższe procedury są blokujące, ale zakończenie operacji przez jeden proces, nie oznacza, że inne procesy w ramach komunikatora również zakończyły wykonywanie swoich operacji – zaleca się więc korzystanie z barier.

4. Rodzaje sieci połączeń w systemach rozproszonych.

Ze względu na łączone elementy możemy podzielić połączenia na połączenia **procesory-pamięć** oraz połączenia **między procesorowe.** Z kolei ze względu na charakterystyki łączenia dzielimy na **sieci statyczne** (zbiór połączeń dwupunktowych) oraz **sieci dynamiczne** (przełączniki o wielu dostępach).

Architektura Intel Knights Landing to **krata przełączników** – system jest skalowalny w przeciwieństwie do magistrali. Jest to przykład sieci dynamicznej, mamy możliwość blokowania połączeń a szerokość pasma transferu jest wysoka.

Do sieci statycznych należy min. gwiazdy, torusy, drzewa, hiperkostki.

VI. Wydajność obliczeń równoległych oraz sortowanie równoległe.

1. Wydajność obliczeń równoległych.

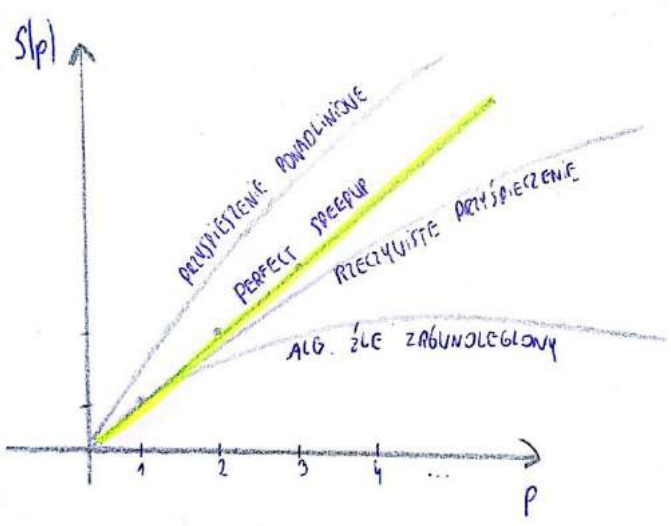
Wydajność obliczeń wyrażać możemy w liczbie operacji na sekundę (MIPS) lub liczbie operacji zmiennoprzecinkowych na sekundę (MFLOPS). Miary te jednak są ciężkie do zastosowania, nie da się bowiem określić jak szybko wykonywana jest pojedyncza instrukcja, jest to zależne od kontekstu. Również ciężko jest określić z ilu i z jakich instrukcji składa się program.

Miarami względnymi (porównawczymi) są więc :

- **Przyśpieszenie obliczeń** (Speed-Up) definiowany jako $S(p) = T_S / T_P(P)$ albo też przyśpieszenie względne definiowane jako $S(p) = T_P(1) / T_P(P)$. T_S jest czasem obliczeń sekwencyjnych przy użyciu najlepszego dostępnego algorytmu sekwencyjnego.

- **Efektywność zrównoleglenia** definiuje się jako $E(p) = S(p) / p$

Poniżej wykres różnych typów przyśpieszeń, zależnie od jakości zrównoleglenia :



Przyspieszenie idealne to przyśpieszenie do jakiego należy dążyć przy zrównoleglaniu programów. Występuje ono w sytuacji perfekcyjnej skalowalności.

Przyspieszenie rzeczywiste to przyspieszenie odbiegające nieznacznie od idealnego, jednak posiadające znamiona dobrej skalowalności. Algorytm cechujący się takim przyspieszeniem może być implementowany.

Źle zrównoleglony algorytm po przekroczeniu pewnej ilości procesów w ogóle nie odnotowuje przyspieszenia – nie jest skalowalny i nie powoduje wzrostu wydajności. Taki algorytm nie powinien być nigdzie wykorzystywany.

Przyspieszenie ponad liniowe może być uzyskane jedynie w specyficznych sytuacjach :

a) kiedy obecność wielu wątków/procesów powoduje **odrzucenie części obliczeń** z uwagi na eliminację konieczności ich przeprowadzania. Przykładem takiego przyspieszenia może być przeszukiwanie drzewa w głąb – liczba potrzebnych kroków do przeszukania drzewa maleje wraz z liczbą wątków (do pewnego rozmiaru puli wątków, równego korzeniom drzewa) z powodu 'odrzucania' części korzeni po analizie przez wątek. Kolejnym przykładem jest zadanie optymalizacyjne.

b) kiedy **przyśpieszenie zależne jest od warunków sprzętowych**, np. mnożenie macierzy 100000x100000 dokonywane jest w pamięci Cache (całkowicie) dzięki podziałowi jej rozmiaru na n części, dzięki czemu brak komunikacji poprzez magistralę FSB powoduje wzrost przyśpieszenia względem algorytmu sekwencyjnego.

Choć równoległość jest dziś powszechnym sposobem na zwiększanie wydajności programów, to w niektórych przypadkach **możliwe jest uzyskanie lepszego lub lepszego czasu** poprawiając jedynie algorytm sekwencyjny.

2. Analiza Amdahla oraz Gustafsona.

Prawo Amdahla mówi, że przy liczbie procesów dążących do nieskończoności czas rozwiązania określonego zadania nie może zmaleć poniżej czasu wykonania sekwencyjnej części programu (podział danych, zebranie danych, wyświetlenie wyników, etc.). Wynika z tego, że efektywność zawsze zmierzać będzie do zera.

W swojej analizie Amdahl nie wziął jednak pod uwagę, że zrównoleglamy nie po to, by rozwiązywać te same zadania coraz szybciej, ale po to, by rozwiązywać w takim samym czasie zadania coraz bardziej złożone.

Poniżej wykresy prezentujące analizę Amdahla oraz Gustafsona :

Problem ten rozwiązuje analiza Gustarfona, która zakłada zwiększanie rozmiaru zadania przy coraz większej liczbie wątków.

Czynniki wpływające na wydajność obliczeń równoległych to :

- Udział sekwencyjnej części (nie dającej się zrównoleglić)

- Zrównoważenie obciążenia pomiędzy procesory/wątki

- Czas komunikacji/synchronizacji

- Czas realizacji obliczeń dodatkowych

- Czas wykonywania operacji systemowych (uruchomienie procesów, alokacja pamięci itp.)

3. Algorytmy sortowania równoległego.

Algorytmy sortowania dzielą się na wewnętrzne (korzystające jedynie z pamięci operacyjnej i podręcznej) oraz zewnętrzne (korzystające np. z dysku twardego).

Innym istotnym podziałem, jest podział na algorytmy oparte **wyłącznie na porównywaniu elementów** (złożoność n*log(n)) oraz na algorytmy oparte na **wiedzy o zakresie elementów** (złożoność n).

Wzorzec **recursive-spliting** polega na rekursywnym dzieleniu tablicy na mniejsze części, aż do otrzymania pojedynczych elementów (podobnie jak merge-sort).

W analizie wydajności poszczególnych sortowań musimy zawsze brać pod uwagę scenariusz pesymistyczny.

Sortowanie szybkie składa się z trzech etapów – podziału tablicy (czas *n*), posortowania pierwszej części (czas $T(n_1)$) oraz podziału drugiej części (czas $T(n_2)$). Sumarycznie więc mamy złożoność rzędu T(n) = nlog(n), jednakże dla wariantu równoległego czas będzie zawsze dłuższy niż *n*, co skutkuje słabą skalowalnością. Dzieje się tak, ponieważ podział tablicy jest zawsze częścią sekwencyjną.

$$T(n) = \phi(n) + T(n1) + T(n2) > \phi(n)$$

Sortowanie bąbelkowe/parzyste-nieparzyste, 'czyste' sortowanie bąbelkowe jest bardzo mało wydajne i fatalne w swojej skalowalności. Dokonując jednak pewnej modyfikacji możemy poprawić nieco jego działanie – jest to algorytm parzyste/nieparzyste. Polega to na iteracyjnym sortowaniu za pomocą sortowania bąbelkowego mniejszych części tablicy (podział tablicy) – każdej z osobna. Wówczas złożoność czasowa jest znacznie niższa.

$$T(n,p) = \phi(n/p * \log(n/p)) + \phi(n/p) + \phi(p)$$

Pierwszy człon odpowiada za podział tablicy na małe części i wstępne sortowanie, drugi człon za scalenie tablicy a trzeci za przesuwanie 'bąbelka' o poziom co iterację.

Pomimo popraw algorytmu, nadal nie zapewnia on pełnej skalowalności. Stosując metody heurystyczne możemy w prawdzie poprawić wyniki algorytmu, poprzez natychmiastowe 'celowanie' w odpowiedni fragment tablicy już pierwszym kroku, zamiast iteracyjnego przesuwania od samego dołu do miejsca właściwego, ale nie są to metody stabilne i poprawiają jedynie **oczekiwaną** a nie **pesymistyczną** złożoność obliczeniową.

Wniosek jest prosty – jeżeli chcemy osiągnąć przyśpieszenie idealne musimy zastosować algorytm posiadający wiedze o dziedzinie problemu – o wartościach. Stosujemy więc **sortowanie kubełkowe**.

Pierwszym etapem sortowania jest rozdzielenie wartości do odpowiednich kubełków a drugim jest posortowanie wartości wewnątrz każdego kubła (każdy wątek odpowiada za inny kubeł). Każdy kubeł musi posiadać wartości większe od poprzedniego i mniejsze od następnego.

$$T(n,p) = \phi(n/p) + \phi(n/p * \log(n/p)) + K$$

Dzięki takiemu podejściu, pierwszy człon odpowiada za przydział do kubłów, drugi za sortowanie końcowe wewnątrz kubełków a ostatni (o wartości nie większej niż n/p) za 'przerzucenie' wyników do jednego kubła (na końcu).

Po skróceniu okazuje się, że złożoność takiego algorytmu wynosi $\phi = n/p * \log(n/p)$ - czyli osiągnęliśmy przyspieszenie idealne. Możemy jeszcze poprawić algorytm, ponieważ jeżeli założenie równomiernego rozkładu nie jest spełnione, wówczas należy zastosować optymalną technikę podziału – np. algorytm sortowania próbkowego (sample sort).

VII. Alternatywne modele programowania równoległego.

1. Model PRAM.

Jest to model teoretyczny, służący do analizy algorytmów. Skrót PRAM rozwijamy jako **Równoległa maszyna o dostępie swobodnym**. Założenia to : wiele procesorów (z własnymi rejestrami), magistrala oraz wspólna pamięć.

Model PRAM realizuje algorytmy w sposób równoległy poprzez odpowiednie numerowanie procesów i powiązanie tej numeracji z realizacją operacji. Synchronizacja działania procesorów jest sprzętowa i automatyczna – pozwala to na pominięcie zagadnień złożoności synchronizacji i komunikacji na linii pamięć – procesor.

W tym modelu możemy wykonywać algorytmy bezpośrednio przez jednostkę ALU, ale nie można stosować zapisu do pamięci tej samej komórki z kilku wątków/procesów.

Podtypy oznacza się poprzez **EREW** (wyłączny zapis, odczyt), **CREW** (jednoczesny odczyt, wyłączny zapis), **ERCW** oraz **CRCW** (analogicznie jak poprzednie dwa).

2. Model równoległości danych oraz dzielonej pamięci wspólnej.

Model SPMD (dla maszyn SIMD – jednowątkowe procesory desktopowe), w tym modelu zrównoleglenie realizowane jest poprzez przydział różnych struktur danych poszczególnym procesom. Wymiana informacji pomiędzy procesami następuje automatycznie, bez udziału programisty. Implementacja powinna obejmować zarówno modele z pamięcią wspólną jak i z przesyłaniem komunikatów.

Przykładem takiego modelu jest nieformalny standard **High Performance Fortran** – przystosowany głównie dla operacji wektorowych i macierzowych. Podobnie do OMP działa na zasadzie dyrektyw dla kompilator Fortrana – dyrektywy deklarują podział tablicy danych na tablicę procesów. Minusem jest konieczność ustalenia podziału danych do wątków niejako 'z góry'.

Model DMS/PGAS to model z rozproszoną pamięcią wspólną, gdzie przestrzeń adresowa również jest dzielona pomiędzy wątki. W efekcie, każdy wątek ma dostęp jedynie do swojej przestrzeni adresowej, ale komunikacja następuje przez wspólną magistralę pamięci.

Języki wspierające ten model to min. CoArray Fotran czy Unified Parrarel C – w przypadku tego języka zastosowano następujące podejście, zmienne współdzielone 'shared' są dystrybuowane pomiędzy wątki (dostęp mają wszystkie wątki ze wszystkich przestrzeni adresowych), natomiast zmienne lokalne 'private' są dostępne tylko dla wątku właściciela. Istnieje również specjalna pętla równoległa – *upc_forall(init;test;affinity)*, którą działa podobnie jak dyrektywa *for* w przypadku OpenMP.

3. GPGPU – obliczenia z wykorzystaniem kart graficznych.

W przeciwieństwie do procesorów desktopowych, tendencją jest duża liczba 'małych' (czyli prostych, mniej funkcjonalnych) rdzeni. Potoki są krótsze, wykonywanie następuje w kolejności a jednostek funkcjonalnych jest mniej.

Jeden, główny układ dekoduje zestaw rozkazów a na wielu potokach wykonuje się rozkazy różnych typów. Istnieje również wiele jednostek ALU, tak więc jeden mikro-procesor podzielony jest na wiele multi-procesorów.

Dzięki takiemu podejściu teoretyczna wydajność oraz przepustowości pamięci są kilkukrotnie wyższe od przepustowości i wydajności w przypadku procesorów desktopowych.

Technologia **cuda-core** odnosi się tylko do potoku przetwarzania liczb stałoprzecinkowych, takich wątków mamy do dyspozycji nawet kilka tysięcy a kod może być implementowany w specjalnym dialekcie C/C++. Nie ma dostępu do pamięci dyskowej, ale pamięć lokalna zarządzana jest z poziomu programu.

Opłacalność wykorzystania tej technologii jest widocznie jedynie w przypadku masowej równoległości. Ponadto konieczna jest synchronizacja przy dostępie do pamięci (GPU dobrze nadają się do obliczeń w stylu równoległości danych – te same obliczenia dla wielu egzemplarzy danych). Istnieją duże możliwości konfiguracyjne.

Dostęp do poszczególnych rejestrów, pamięci jest skomplikowany, ale sam model programowania przypomina OpenMP czy MPI (SPMD/SIMD). Kod programu nazywany jest tradycyjnie kernelem.

Hierarchia pamięci jest dwupoziomowa – zmienne globalne oraz lokalne (dla grup wątków) a także poziom rejestrów – ukryty przed programistą.

Problemem jest wymagania specyficznego typu algorytmów, w niektórych przypadkach, gdy aplikacje są dostosowane do pracy z układami GPU można zyskać wiele, lecz dla większości aplikacji zysk jest nie wielki lub żaden.

Część praktyczna – syntaxy i przykłady wykorzystania

I. Pthreads.

1. Podstawowe operacje na wątkach.

- Identyfikator wątku :

```
pthread_t threadID = null;
```

- Atrybuty wątku :

```
pthread_attr_t attribute = null;
```

- Tworzenie wątku :

```
int pthread_create(pthread_t *IDWątku, pthread_attr_t *AtrybutyWątku,
void * (*funkcja_wątku)(void *), void *ArgumentyWątku);
```

- Oczekiwanie na zakończenie wątku :

```
int pthread_join(pthread_t IDWątku, void **ZwrotZFunkcji)
```

- Przykład stworzenia nowego wątku i otrzymania rezultatu jego pracy :

```
int main()
{
	int arg1 = 58452;
	int result = null;
	int error = null;

	pthread_t threadID = null;
	pthread_attr_t attribute = null;

	error = pthread_create(&threadID, attribute, function, &arg1);
	error = pthread_join(threadID, (void**)&result);

	printf("Wynik to : %d", result);
}
int *function(void *arg_wsk)
{
	int retVal = *((int*)arg_wsk);
	retVal += 55;

	return &retVal;
}
```

Warto zauważyć, że zmienna *arg1* przesyłana przez wskaźnik do funkcji jest zmienną globalną, oznacza to, że każda lokalna modyfikacja jej wartości powoduje zmianę we wszystkich wątkach. Z tego powodu, pracujemy na lokalnej zmiennej *retVal*, która jedynie pobiera wartość ze wcześniej wspomnianej zmienne globalnej. Dzięki temu, po zakończeniu działania wątku możemy przypisać zwróconą wartość do 'czystej' zmiennej lokalnej, zachowując przy tym pierwotny stan zmiennej argumentu.

- Porównywanie identyfikatorów wątków :

```
int pthread_equal(pthread_t id1, pthread_t id2);
```

Zwraca wartość niezerową jeżeli wątki są równe, albo wartość zerową jeżeli nie są równe.

- Zwrócenie identyfikatora aktualnego wątku :

```
pthread_t pthread_self();
```

- Zmiana rodzaju wątku z *joinable* na *detached* :

```
int pthread_detach(pthread_t threadID);
```

Wątek takiego typu (detached) z chwilą zakończenia pracy zwalnia od razu wszystkie zasoby. Z wątku joinable możemy z kolei odczytać wartość po jego zakończeniu a zasoby nie są zwalniane do czasu wywołania funkcji *pthread_join()*.

- Zakończenie działania wątku (natychmiastowe) :

```
int pthread_exit(void *retVal);
```

Jeżeli zamykamy wątek w taki sposób, to zasoby nie są zwalniane, ale wartość, jaką wątek powinien zwrócić jest dostępna dla innego wątku w ramach tego samego procesu, który wywołał metodę *pthread_join()*.

- Specyfikowanie atrybutów :

```
pthread_attr_t atrybuty;

pthread_attr_init(&atrybuty);

/* tutaj wykonujemy funkcje ustawiające konkretne atrybuty */

pthread_attr_destroy(&atrybuty);
```

- Funkcję modyfikujące atrybuty :

Z racji mnogości możliwości zmian atrybutów podam jedynie kilka przykładowych funkcji.

```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);

int pthread_attr_getdetachstate(const pthread_attr_t *attr, int
*detachstate);

int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);

int pthread_attr_getstacksize(const pthread_attr_t *attr, size_t
*stacksize);
```

Powyższe funkcje kolejno : ustawiają stan *joinable* lub *deatched*, pobierają aktualny stan oraz ustanawiają rozmiar stosu i pobierają rozmiar stosu. Argument jaki należy przykładowo podać dla pierwszej funkcji to `'PTHREAD_CREATE_DETACHED'`.

2. Synchronizacja – zamki.

- Zamek czyli mutex :

```
pthread_mutex_t mutexObject = null;
```

- Inicjacja mutexa :

```
int pthread_mutex_init(pthread_mutex_t *mutexObject,

       const pthread_mutex attr_t *mutexAtrybuty)
```

Atrybuty nie muszą być podawane, można przekazać w tym miejscu wartość null.

- Zwolnienie zasobów mutexu (zniszczenie ostateczne) :

```
int pthread_mutex_destroy(pthread_mutex_t *mutexObject)
```

- Zamknięcie zamka na mutexie (początek sekcji krytycznej) :

```
int pthread_mutex_lock(pthread_mutex_t *mutexObject)
```

- Próba zamknięcia zamka z natychmiastowym zwróceniem sterowania :

```
int pthread_mutex_trylock(pthread_mutex_t *mutexObject)
```

W przypadku wolnego zamka działa tak samo jak zwykłe zamknięcie, ale gdy zamek jest już zamknięty lub nie można go z jakiegoś powodu zamknąć zwraca wartość niezerową. Zero zwraca jedynie gdy muteks typu *reentrant* został zamknięty przez ten sam wątek.

- Otwarcie zamka na mutexie (koniec sekcji krytycznej) :

```
int pthread_mutex_unlock(pthread_mutex_t *mutexObject)
```

- Mutex zwyczajny, jednorazowy (nie wymaga specjalnego atrybutu) :

```
"PTHREAD_MUTEX_NORMAL" lub "PTHREAD_MUTEX_DEFAULT"
```

- Mutex wielokrotnego wejścia (reentrant), posiada zmienną zliczającą zamknięcia :

```
"PTHREAD_MUTEX_RECURSIVE"
```

- Deklarowanie typu mutexu poprzez argument :

```
int pthread_mutexattr_settype(pthread_mutexattr_t *at, tutaj_podajemy_typ);
```

W celu stworzenia obiektu atrybutu postępujemy analogicznie jak w przypadku atrybutów wątków (zamieniając jedynie *pthread_attr* na *pthread_mutexattr*).

- Prosty przykład dekompozycji pętli i sumowania z wykorzystaniem wielu wątków przy użyciu zamków (mutexów) :

```
pthread_mutex_t zamek = null;
int sumaGlobal = 0;
pthread_t watki[4] = null;

int main()
{
	int i = 0;
	int tID = 0;
	pthread_mutex_init(&zamek, NULL);
	for (i = 0; i < 4; i++)
	{
		tID = i;
		pthread_create(&watki[i], NULL, function, (void *)&(i));
	}
	for (int i = 0; i < 4; i++)
	{
		pthread_join(watki[i], NULL);
	}

	printf("Wynik to : %d", sumaGlobal);
	pthread_mutex_destroy(&zamek);
}

int *function(void *arg_wsk)
{
	int mojeID = *((int*)arg_wsk);
	int sumaLokalna = 0;
	int ratio = 50;
	int i = 0;

	for (i = mojeID * ratio; i < (mojeID * ratio * 2); i++)
		sumaLokalna += 1560 * i;
	pthread_mutex_lock(&zamek);
	sumaGlobal += sumaLokalna;
```

```
	pthread_mutex_unlock(&zamek);
	pthread_exit((void*)0);
}
```

Powyższy przykład pokazuje jak w prosty sposób dokonać dekompozycji pętli – każdy wątek dostaje swój identyfikator, dzięki któremu wie jaką część pętli ma obsłużyć.

3. Zmienne warunku.

- Zmienna warunku :

```
pthread_cond_t conditionalLock = null;
```

- Tworzenie zmiennej warunku :

```
int pthread_cond_init(pthread_cond_t *cond,pthread_condattr_t *cond_attr)
```

- Uśpienie wątku na zmiennej :

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *zamek)
```

- Wybudzenie pierwszego w kolejności wątku :

```
int pthread_cond_signal(pthread_cond_t *cond)
```

- Wybudzenie wszystkich oczekujących wątków :

```
int pthread_cond_broadcast(pthread_cond_t *cond)
```

- Przykład wykorzystania zmiennych warunku :

```
pthread_mutex_t zamek = null;
pthread_cond_t condVar = null;
int sumaGlobal = 0;
pthread_t watki[4] = null;

int main()
{
	int i = 0;
	int tID = 0;
	pthread_mutex_init(&zamek, NULL);
	pthread_cond_init(&condVar, NULL);

	for (i = 0; i < 6; i++)
	{
		tID = i + 1;
		pthread_create(&watki[i], NULL, function, (void*)&tID);
	}
	/* Join, czyszczenie etc. */
}
```

```
int *function(void *arg_wsk)
{
	int myID = *((int*)arg_wsk);
	pthread_mutex_lock(&zamek);
	if (myID > 3){
		pthread_cond_wait(&condVar);
		pthread_mutex_unlock(&zamek);}

	/* Wykonuje działania */
	pthread_mutex_unlock(&zamek);

	if (myID == 6)
		pthread_cond_signal(&condVar);
}
```

W tym przykładzie wątki o numerach większych od 3 mają pierwszeństwo w dostępie do sekcji krytycznej, a dopiero gdy wszystkie wątki o wysokich numerach ukończą zadanie, wówczas przyznawany jest dostęp zadaniom o numerach niskich, oczekujących na zmiennej warunku.

II. Java.

1. Tworzenie wątków i podstawowe operacje na nich.

- Tworzenie wątku z użyciem klasy Thread :

```
class Wątek extends Thread
{
    public void run(){ System.out.println("Thread is running");}
}
Wątek t1 = new Wątek();
```

- Tworzenie wątku z użyciem interfejsu Runnable :

```
class Wątek implements Runnable
{
    public void run(){ System.out.println("Thread is running");}
}
Thread t1 = new Thread(new Wątek());
```

- Uruchamianie wątku :

```
t1.start();
```

- Uśpienie bieżącego wątku na określony czas :

```
t1.sleep(czas_w_ms);
```

- Oczekiwanie na zakończenie wątku :

```
t1.join(czas_w_ms);  // czas jest opcjonalny
```

- Wysłanie sygnału przerwania wątku :

```
t1.interrupt();
```

- Sprawdzenie stanu sygnału przerwania wątku :

```
        if (t1.isInterrupted())
            t1.flaga = false;
```

Gdzie atrybut typu *boolean* o nazwie 'flaga' jest sprawdzany w krytycznych momentach metody *run()* wątku, w celu umożliwienia 'eleganckiego' przerwania wątku.

2. Synchronizacja wątków w Javie.

- Metoda synchronizowana :

```
public synchronized void funkcjaSynchronizowana(){};
```

- Blok *synchronized* :

```
        Object singleMutex = new Object();
        synchronized(singleMutex)
        {
            zmiennaGlobalna++;
        }
```

W przypadku opuszczenia pierwszej linijki oraz zasobu do bloku otrzymamy 'imperatywny' blok, którego zamka/mutexu nie będziemy mogli przesłać do żadnej innej funkcji czy klasy.

- Obiekty zamków typu *Lock* :

```
        Lock basicLock = new ReentrantLock();
        basicLock.lock();
        try{/* Operacje przeprowadzane synchronicznie */}
        finally{basicLock.unlock();}
```

- Obiekty atomowe :

```
        AtomicInteger bezpiecznyLicznik = new AtomicInteger(50);
        AtomicIntegerArray bezpiecznaTablica = new AtomicIntegerArray(10);
```

- Przykład implementacji puli wątków :

```
ExecutorService wykonawcaPuli = Executors.newFixedThreadPool(8);
for (int i=0; i<800; i++)
{
    Runnable tmpWątek = new Wątek();
    wykonawcaPuli.execute(tmpWątek);
}
wykonawcaPuli.shutdown();
while(!wykonawcaPuli.isTerminated()){}
```

Na początku tworzony jest zarządca puli wątków (o stałym rozmiarze puli wynoszącym 8), a następnie w pętli przesyłane są do zarządcy nowe wątki w liczbie przekraczającej znacznie pulę wątków. Wykonawca więc oczekuje z wykonaniem wątków, które nie mieszczą się w puli i umieszcza je w kolejce. Po wywołaniu metody *shutdown()* wykonawca nie przyjmuje już nowych wątków i oczekuje jedynie na zakończenie aktualnie działających. Aby nie zakończyć programu przed ukończeniem pracy przez wszystkie wątki czekamy w pętli aż wykonawca puli zostanie zamknięty.

- Interfejs *Callable* i zwrot wyniku :

```
ExecutorService wykonawcaPuli = Executors.newFixedThreadPool(8);
List<Future<String>> listaZwrotu = new ArrayList<>();

for (int i=0; i<800; i++)
{
    Callable<String> tmpWątek = new Wątek();
    Future<String> tmpZwrot = wykonawcaPuli.submit(tmpWątek);
    listaZwrotu.add(tmpZwrot);
}

wykonawcaPuli.shutdown();
while(!wykonawcaPuli.isTerminated()){}
```

W tym przypadku w każdej iteracji tworzymy obiekt typu *Future<String>* a do wykonawcy zadanie przesyłamy za pomocą metody *submit().* Efekt jest podobny, ale z każdą iteracją otrzymujemy zwrócony przez zadanie wynik w postaci łańcucha znaków.

3. Monitory i zmienne warunku.

- Uśpienie wątku :

```
this.wait(czas_w_ms);  // czas jest opcjonalny
```

- Obudzenie wątku :

```
this.notify();
```

- Obudzenie wszystkich wątków :

```
this.notifyAll();
```

- Przykładowa klasa monitora :

```
public class MonitorObject
{
    public synchronized void wejdźDoMonitora(boolean czyPriotytet) throws
InterruptedException
    {
        if (czyPriotytet){/* Realizuje zadanie*/}
        else
            this.wait();
    }
    public synchronized void wyjdźZMonitora()
    {
        notifyAll();
    }
}
```

III. OpenMP.

1. Podstawowe dyrektywy i klauzule.

- Format każdej dyrektywy :

```
#pragma omp nazwa_dyrektywy lista_klauzul
```

- Dyrektywa zakresu obszaru równoległego :

```
#pragma omp paralel lista_klauzul
```

- Ustanawianie liczby wątków :

a) Za pomocą procedury :

```
omp_set_num_threads(liczba_wątków);
```

b) Za pomocą zmiennej środowiskowej :

```
$ set OMP_NUM_THREADS = 10;
```

c) Za pomocą klauzuli :

```
#pragma omp parallel num_threads(10)
```

- Pobieranie zmiennych wykonania :

Aby korzystać z funkcji bibliotecznych należy dołączyć bibliotekę *omp.h*. Funkcje ustawiające mają format void *funkcja*(int); natomiast pobierające int *funkcja* (void).

```
omp_get_num_threads();  // Pobieranie liczby wątków

omp_set_num_threads(liczba_wątków);// Ustanawianie liczby wątków

omp_get_thread_num();   // Pobieranie ID aktualnego wątku

omp_in_parallel();      // Sprawdzenie wykonania równoległego

omp_get_max_threads();  // Pobranie maksymalnej liczby wątków

omp_set/get_dynamic();  // Pobranie lub ustanowienie dynamicznej liczby
wątków

omp_set/get_nested();   // Pobranie lub ustanowienie możliwości
zagnieżdżania obszarów równoległych.
```

2. Dyrektywy obsługi zamków.

- Funkcje obsługi zamków :

Choć zamki nie są powszechnie stosowanym mechanizmem, z uwagi na lepsze możliwości zrównoleglania zaszyte w OMP, to można zastosować także to podejście.

a) Postać zamka i jego inicjalizacja:

```
omp_lock_t zamek = null;

omp_init_lock(&zamek);
```

b) Zamykanie, otwieranie oraz niszczenie :

```
omp_set_lock(&zamek);        // Zamknięcie zamka

omp_test_lock(&zamek);       // Próba zamknięcia bez blokady

omp_unset_lock(&zamek);      // Otwarcie zamka

omp_destroy_lock(&zamek);          // Zwolnienie zasobów zamka
```

- Funkcje pomiaru czasu :

```
int time = omp_get_wtime();          // Pobranie czasu zegarowego

int timeResolution = omp_get_wtick();        // Pobranie dokładności zegara
```

Istnieje możliwość określenia również atrybutów *dynamic, nested* oraz *schedule* za pomocą zmiennych środowiskowych o nazwach odpowiadających atrybutom z przedrostkiem 'OMP_(atrybut) TRUE/FALSE'.

3. Główne klauzule obszaru równoległego.

- **Klauzule dla dyrektywy parallel for:**

a) Składania dyrektywy :

```
#pragma omp parallel for [klauzula]
```

b) Możliwe klauzule :

* private(*zmienne*) : każdy wątek otrzymuje swój, prywatny egzemplarz tej zmiennej.

* shared(*zmienne*) : zmienna traktowana jest jak globalna dla wszystkich wątków.

* firstprivate(*zmienne*) : każdy wątek otrzymuje swój egzemplarz tej zmiennej, inicjowany wartością początkową.

* lastprivate(*zmienne*) : każdy wątek otrzymuje swój egzemplarz a wartość po wyjściu z obszaru jest taka, jaka była w ostatnim wątku.

* nowait : usuwa domyślną barierę, zaszytą np. w dyrektywie *for*.

* ordered : zapewnia, że kod umieszczony pod tą klauzulą będzie wykonany sekwencyjnie.

* reduction(*operacja* : *zmienna*) : umożliwia wykorzystanie zmiennej wspólnej do danej operacji wykonywanej przez wszystkie wątki bez obawy o wyścig.

* schedule(*typ* , *chunk_size*) : specyfikuje rozmiar iteracji dla każdego wątku oraz typ podziału. Możliwe typy to : *dynamic* (kolejność iteracji ustalana dynamicznie, nie jest dokładnie znana), *static* (każdy wątek decyduje o tym, które iteracje 'weźmie'), *guided* (niejako połączenie dwóch poprzednich), *runtime* (biblioteka decyduje o kolejności).

- Dodatkowe klauzule dla samej dyrektywy parallel :

* copyin(*zmienna*) : Pozwala przyjąć wartość wątku 'master' jako wartość 'threadprivate'.

* default(shared/none) : Definiuje czy należy traktować nie zdeklarowane zmienne jako wspólne, czy pomijać je.

* if (*wyrażenie_warunkowe*) : Jeśli prawda to obszar jest wykonywany równolegle, jeśli fałsz to obszar jest wykonywany sekwencyjnie.

* num_threads(*ilość_wątków*) : Specyfikuje ilość wątków wykonujących obszar.

* single [*inne_klauzule*] : Tylko jeden wątek może wykonać ten obszar.

* threadprivate (*zmienne*) : Zmienna jest własnością wątku i przenoszona jest między obszarami równoległymi.

* sections [*inne_klauzule*] : Definiuje, że w obszarze znajdą się sekcje, które dzielone są pośród wszystkie wątki, tak, że część wykonuje jedną sekcję, część drugą etc. Same sekcje wewnątrz obszaru specyfikujemy :

```
#pragma omp section { /* kod sekcji */ }
```

- Klauzule dla dyrektywy task :

a) Składnia dyrektywy :

```
#pragma omp task [klauzule]
```

Powoduje stworzenie zadania w formie bloku kodu, którego wykonanie może być opóźnione (asynchroniczne).

b) Możliwe klauzule :

* klauzule zmiennych, jak w przypadku *parallel*

* if (*wyrażenie_warunkowe*) : Jeśli **prawda** to zadanie jest wykonywane asynchroniczne, tak jak powinno być wykonane, natomiast jeśli **fałsz** to zadanie wykonywane jest od razu (synchronicznie)

* united : W przypadku wstrzymania wykonania zadanie może przejąć inny wątek.

* final (*wyrażenie_warunkowe*) : Odwrotność *if,* jeżeli **prawda** to kolejne zadania (po tym oznaczonym tą klauzulą) są wykonywane natychmiastowo.

* mergable : klauzula dla zadań zagnieżdżonych (jedno wewnątrz drugiego).

Wewnątrz zadania można użyć następujących dyrektyw :

#pragma omp taskyield : W tym miejscu wstrzymywane jest wykonanie zadania w celu uruchomienia innego zadania.

#pragma omp taskwait : Bariera dla zadań, zadanie oczekuje do momentu aż wszystkie zadania zostaną ukończone.

- Pozostałe dyrektywy :

a) Dyrektywa *master* sprawia, że tylko główny wątek wykonuje podany blok :

```
#pragma omp master { /* kod */}
```

b) Dyrektywa *flush* wymusza unifikowanie widoku pamięci dla obiektów '*shared*' danego wątku :

```
#pragma omp flush (zmienna_opcjonalna)
```

c) Dyrektywa *critical* tworzy sekcję krytyczną wewnątrz bloku :

```
#pragma omp critical(nazwa_sekcji) { /* sekcja krytyczna */}
```

d) Dyrektywa *barrier* zakłada barierę w danym miejscu :

```
#pragma omp barrier
```

e) Dyrektywa *atomic* zapewnia realizację poniższej operacji jako atomowa :

```
#pragma omp atomic

/* wyrażenie /*
```

IV. MPI.

1. Podstawowe funkcje do pracy w środowisku MPI.

- Inicjalizacja MPI :

```
int MPI_Init(int *pargc, char ***pargv);
```

- Pobranie rozmiaru komunikatora oraz rangi procesu :

```
int MPI_Comm_size(MPI_COMM_WORLD, int *pCount);

int MPI_Comm_rank(MPI_COMM_WORLD, int *pID);
```

- Finalizowanie pracy z MPI :

```
int MPI_Finalize();
```

2. Przesyłanie dwupunktowe.

- Przesyłanie blokujące, strona wysyłająca (syntax + użycie) :

```
int MPI_Send(void *zmienna, int ilośćZmiennych, MPI_Datatype typDanej, int
rangaCelu, int tagID, MPI_Comm komuniktator);
```

```
errorCode = MPI_Send(&zmienna, 1, MPI_INT, 2, 0, MPI_COMM_WORLD);
```

- Odbieranie blokujące, strona odbierająca (syntax + użycie) :

```
int MPI_Recv(void *zmiennaOdbioru, int ilośćZmiennych, MPI_Datatype tymDanej, int rangaŹródła, int tagID, MPI_Comm komunikator, MPI_Status *info);

errorCode = MPI_Recv(&reciveValue, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, NULL);
```

- Inne typy blokującego przesyłania :

a) Buforowane – MPI_Bsend(*argumenty takie same jak dla zwykłego przesyłania)*

b) Synchroniczne – MPI_Ssend(*argumenty takie same jak dla zwykłego przesyłania*)

c) Gotowości – MPI_Rsend(*argumenty takie same jak dla zwykłego przesyłania*)

- Przesyłanie nie blokujące :

```
int MPI_Isend(void *zmienna, int count, MPI_Datatype typDanej, int cel, int tagID, MPI_Comm komunikator, MPI_Request *zwrot);

errorCode = MPI_Isend(&zmienna, 1, MPI_Char, 2, 0, MPI_WORLD, &retBuf);

MPI_Wait(&retBuf, &status);    // Oddaje sterowanie po wysłaniu
```

- Odbieranie nie blokujące :

```
int MPI_Irecv(void *zmiennaZwrotu, int ilość, MPI_Datatype typDanej, int źródło, int tagID, MPI_Comm komunikator, MPI_Request *kontrolerTransmisji);
```

- Procedury przesyłania nie blokującego :

```
int MPI_Wait(MPI_Request *preq, MPI_Status *pstat); // Czeka na zakończenie

int MPI_Test(MPI_Request *preq, int *flaga, MPI_Status *status); // Sprawdza czy zakończono transmisję, we fladze umieszcza true jeśli tak)
```

Istnieją również warianty *Waitany, Waitall* oraz *Testany, Testall*.

- Procedury testowania przybycia komunikatów (bez odbioru) :

```
int MPI_Probe(int scr, int tag, MPI_Comm komunitator, MPI_Status *status);

int MPI_Iprobe(int scr, int tag, MPI_Comm komunikator, int *flag, MPI_Status *status);
```

Możemy w każdym przypadku zignorować argument *status* podając :

MPI_STATUS_IGNORE

3. Przesyłanie grupowe.

- Bariera :

int **MPI_Barrier**(MPI_Comm komunikator);

- Rozgłaszanie :

int **MPI_Bcast**(void *zmienna, int count, MPI_Datatype typDanej, int rangaWysyłającego, MPI_Comm komunikator);

- Zbieranie :

int **MPI_Gather**(void *zmienna, int ilośćZmiennych, MPI_Datatype typZmiennej, void *zapis, int ilośćZapisywanych, MPI_Datatype typZapisywanej, int wysyłający, MPI_Comm komunikator);

- Rozpraszanie :

int **MPI_Scatter**(void *zmienna, int ilośćZmiennych, MPI_Datatype typZmiennej, void *zapis, int iloścZapisywanych, MPI_Datatype typZapisywanej, int źródło, MPI_Comm komunikator);

- Redukcja :

int **MPI_Reduce**(void *buforZmiennej, int ilośćZmiennych, MPI_Datatype typZMiennej, MPI_Op operacja, int źródło, MPI_Comm komunikator);

- Zbieranie wszyscy do wszystkich :

int **MPI_Allgather**(void *buforZmiennych, int ilośćZmiennych, MPI_Datatype typZmiennych, void* buforZapisu, int ilośćZapisanych, MPI_Datatype typZapisanych, MPI_Comm komunikator);

- Rozpraszanie wszyscy do wszystkich :

int **MPI_Alltoall**(void *buforZmiennych, int ilośćZmiennych, MPI_Datatype typZmiennych, void *buforZapisu, int ilośćZapisywanych, MPI_Datatype typZapisywanych, MPI_Comm komunikator);

- Redukcja wszyscy do wszystkich (połączona z rozgłaszaniem) **:**

int **MPI_Allreduce**(void *buforZmiennych, void *buforZapisu, int ilośćPrzesyłanych, MPI_Datatype typDanych ,MPI_Op operacja, MPI_Comm komunikator);

- Możliwe operacje :

* MPI_MAX : maksimum

* MPI_MIN : minimum

* MPI_SUM : suma

* MPI_PROD : iloczyn

- Definiowanie operacji :

int MPI_Op_create(MPI_User_function *funkcja, int commute, MPI_Op *nowaOperacja);

4. Przykłady wykorzystania MPI.

- Równoległe obliczanie całki :

```
	int myRank = 0;
	int processCount = 0;

	MPI_Init(&argvc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
	MPI_Comm_size(MPI_COMM_WORLD, &processCount);

	if (myRank == 0)
	{
		getIntegralData();
	}
	int root = 0;
	MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
	MPI_Bcast(&a, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
	MPI_Bcast(&b, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

	double sumaLokalna = liczCalke(N / size, myRank * ratio, myRank *
ratio * 2);
	double sumaGlobalna = 0;
	MPI_Reduce(&sumaLokalna, &sumaGlobalna, 1, MPI_Double, MPI_SUM, root,
MPI_COMM_WORLD);

	if (myRank == root) { printf("Wynik to %lf", &sumaGlobalna); }
```