# Programowanie maszyn z pamięcią wspólną w standardzie OpenMP – ciąg dalszy.

# Dyrektywa atomic

- Często sekcja krytyczna polega na uaktualnieniu jednej lokacji w pamięci. OpenMP dostarcza dyrektywę `atomic` dla tego typu sekcji krytycznych.
- Po dyrektywie `atomic` może wystąpić pojedyncza instrukcja uaktualniająca zmienną postaci: `x op=expr`, gdzie `expr` jest wyrażeniem, a `op` jest jednym z binarnych operatorów (dopuszczalne są również wyrażenia typu `x++` albo `--x`).
- Dyrektywa `atomic` serializuje odczyt i zapis zmiennej x w jednej sekcji krytycznej. Obliczanie wyrażenia `expr` nie jest serializowane !!!
- Każda dyrektywa `atomic` może być zastąpiona dyrektywą `critical`. Jednakże większość maszyn posiada wsparcie sprzętowe (specjalne instrukcje), sprawiające że dyrektywa `atomic` może być o wiele wydajniej zaimplementowana niż `critical`.

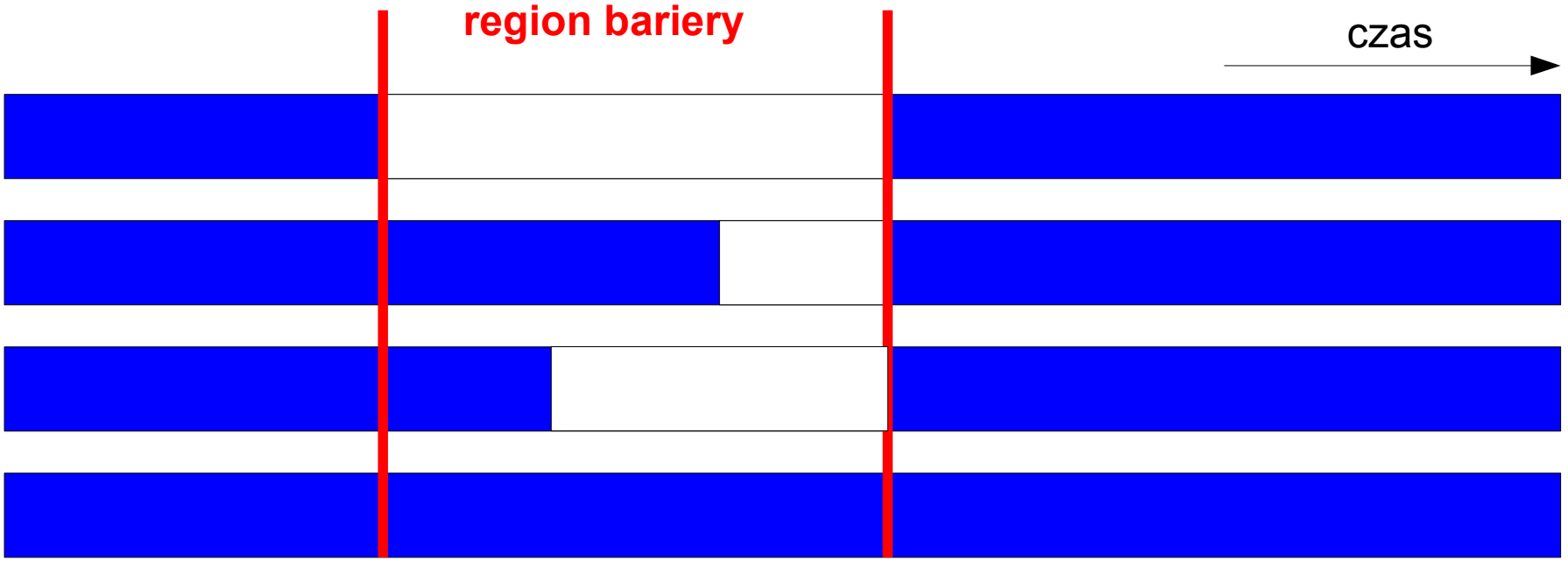
# Dyrektywa barrier

- Jest najprostszą dyrektywą synchronizacyjną. Ma postać:

```
#pragma omp barrier
```

- Po napotkaniu tej dyrektywy wątek zostaje wstrzymany do momentu, gdy wszystkie inne wątki napotkają tę dyrektywę.
- Dyrektywa `barrier` jest wykonywana automatycznie po zakończeniu dyrektywy `for`, chyba że w dyrektywie `for` umieszczono klauzulę `nowait`.
  - ***Oznacza to że po zakończeniu zrównoleglonej pętli for wątek czeka aż pozostałe wątki zakończą pętlę for.***

# Dyrektywa barrier - wizualizacja




- Kolorem białym oznaczono oczekiwanie wątku.
- Wątek zostaje wypuszczony z punktu bariery dopiero wtedy, gdy wszystkie inne wątki osiągnęły ten punkt.
- Należy pamiętać, że implementacja dyrektywy barrier ma złożoność czasową O(log(p)), gdzie p jest liczbą procesorów.
  - Nadużycie dyrektywy barrier zmniejsza skalowalność.

# Przykład *poprawnego* użycia dyrektywy nowait w klauzuli for

```
pragma omp parallel shared(a,b,n,m,y,z)
{
    #pragma omp for nowait
        for (i=1; i<n; i++)
            b[i] = (a[i] - a[i-1]) / 2.0;
    #pragma omp for nowait
        for (i=0; i<m; i++)
            y[i] = sqrt(z[i]);
}
```

- Obydwie pętle for są niezależne od siebie
- Wątek może więc przystąpić do wykonywania swojej puli iteracji w drugiej pętli for bez czekania na inne wątki.
- Klauzula nowait w drugiej pętli for, to trywialna optymalizacja (zapewne i tak automatycznie wykona ją kompilator) w celu uniknięcia podwójnej operacji bariery.
  - Na zakończenie pętli for i na zakończenie regionu równoległego.
- Uwaga: operacje bariery są bardzo kosztowne i ***poprawne*** użycie dyrektywy nowait może doprowadzić do znacznego zwiększenia wydajności.

# Przykład użycia dyrektywy nowait w klauzuli for z *subtelnym błędem.*

```
pragma omp parallel
{
    #pragma omp for nowait
        for (i=0; i<n; i++)
            a[i]=i
    #pragma omp for
        for (i=0; i<n; i++)
            b[i]=2*a[i];
}
```

- Można argumentować, że, wprawdzie istnieje zależność między pętlami, ale ten sam wątek który stworzył a[i] będzie tym samym który wykorzysta a[i] (Zakres iteracji obydwu pętli for jest identyczny).
- Ale nie pokazaliśmy kompilatorowi jak rozdzielić iteracje pomiędzy wątki (brak klauzli schedule). Nawet z klauzulą schedule można to zrobić na kilka sposobów. W związku z tym iteracje mogą być rozdzielone w dwóch pętlach w różny sposób – standard OpenMP 2.5 nie gwarantuje identycznego podziału.
  - W związku z tym druga pętla for może odczytać wartości z tablicy a, które nie zostały zapisane przez pierwszą pętlę for – sytuacja wyścigu.
- W standardzie OpenMP 3.0 to ma się zmienić.

# Dyrektywy single i master

- Dyrektywa ta ma postać:

```
#pragma omp single [klauzule]
   blok_instrukcji
```

- Po napotkaniu dyrektywy single tylko jeden wątek wykona blok instrukcji. Pozostałe wątki przeskakują na koniec bloku.
- Pozostałe wątki czekają na wątek wykonujący blok instrukcji (operacja bariery), chyba że została podana klauzula `nowait`.
- Dyrektywa `single` nie określa, który wątek wykona blok instrukcji. Będzie to wątek dowolny. Podobną do dyrektywy `single` jest dyrektywa:

```
#pragma omp master [klauzule]
   blok_instrukcji
```

- Sprawia ona że blok instrukcji zostanie wykonany przez wątek nadrzędny (master) - o numerze 0.
- Z dyrektywą master należy uważać, gdyż ***nie ma wbudowanej synchronizacji barierowej***.

# Przykład błednego użycia konstruktu master

```
#pragma omp parallel shared (a,b) private(i)
{
#pragma omp master
   a=10
#pragma omp for
   for (i=0; i<n; i++)
      b[i]=a;
}
```

- Chcemy aby zmienna a była inicjalizowana tylko raz.
- Ponieważ konstrukt master nie ma wbudowanej synchronizacji, to wątki wykonujące pętle for mogą rozpocząć wykorzystywanie zmiennej a w pętli zanim zostanie ona zainicjalizowana przez wątek główny.
  - Sytuacja wyścigu.
- Należy wtawić #pragma omp barrier albo zmienić master na single.

# Dyrektywa single - wizualizacja

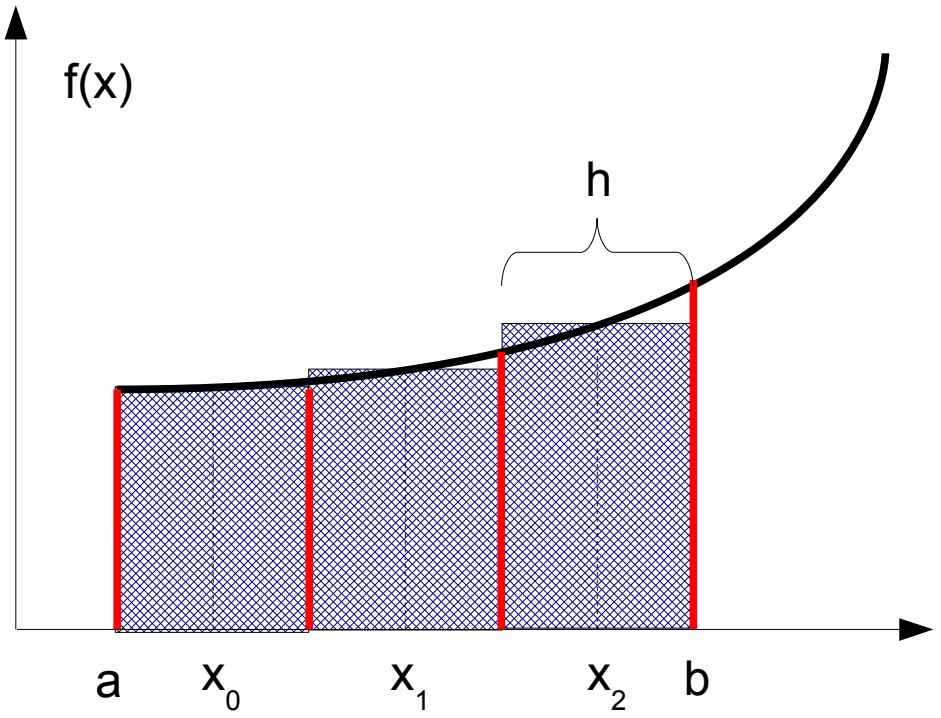

- Kolorem białym oznaczono oczekiwanie wątku.
- Przykłady zastosowania:
  - Obliczanie danych globalnych
  - Wejście-wyjście z bloku równoległego.

# Funkcje operujące na zamkach

```
void omp_init_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
void omp_set_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
int omp_test_lock(omp_lock_t *lock);
```

- Zamek realizuje wzajemne wykluczanie wątków.
- Przed użyciem należy go zainicjalizować poprzez `omp_init_lock`, a gdy staje się zbędny usunąć funkcją `omp_destroy_lock`.
- Funkcja `omp_set_lock` wchodzi, a `omp_unset_lock` wychodzi z sekcji krytycznej. Tylko jeden wątek może przebywać w sekcji krytycznej; pozostałe czekają w funkcji `omp_set_lock` na wejście.
- Funkcja `omp_test_lock` usiłuje wejść do sekcji krytycznej, i jeżeli w danej chwili nie jest to możliwe natychmiast powraca z wynikiem zero. Gdy udało się wejść do sekcji krytycznej natychmiast powraca z wynikiem różnym od zera.
- Wersje funkcji z końcówką `_nest_lock` umożliwiającą wielokrotne (rekurencyjne) wchodzenie do sekcji krytycznej przez ten sam wątek. Liczba wywołań funkcji `set` musi być równa liczbie wywołań funkcji `unset`.

# Przykład: całkowanie metodą prostokątów




$$h = \frac{b-a}{N}$$

$$x_i = a + \frac{h}{2} + i * h$$

$$\int_a^b f(x) \approx h * \sum_{i=0}^{N-1} f(x_i)$$

- Całka oznaczona przybliżana przy pomocy sumy pól powierzchni N (w przykładzie N=3) prostokątów.
- Obliczenia można obsłużyć jedną pętlą for.

# Całkowanie: wersja niewydajna

```
double Sum(int N,double a,double b)
{
   int i;
   double sum=0.0;
   double h=(b-a)/(double)N;

   #pragma omp parallel for default(none) firstprivate(N,h,a) shared(sum)
   for(i=0;i<N;i++) {
        double func=f(a+i*h+h/2);
   #pragma omp critical
                sum+=func;
   }
   return sum*h
}
```

- Ciekawostka: program skompiluje się do wersji jednowątkowej na kompilatorze nie obsługującym OpenMP
- `sum` jest zmienną współdzieloną w której gromadzona jest suma pól prostokątów.
- skoro dostęp do niej ma wiele wątków, które jednocześnie ją zwiększają, to musimy ją chronić sekcją krytyczną.
- Zastosowanie sekcji krytycznej obniża znacznie wydajność, ponieważ wątek będący w sekcji krytycznej i aktualizujący tę zmienną wstrzymuje pozostałe wątki chcące zrobić to samo.

# Całkowanie: jak poprawić wydajność

- Idealnie byłoby gdyby każdy wątek posiadał prywatną kopię zmiennej sum, w której byłaby gromadzona suma pól prostokątów rozpatrywanych w iteracjach przydzielonych temu wątkowi.
  - Te lokalne sumy mogłyby być zsumowane po wyjściu z pętli for.
  - Takie rozwiązanie nie potrzebowałoby synchronizacji w każdej iteracji.
- Tak więc sum musi być zmienną prywatną.
- Pytanie: w jaki sposób zsumować prywatne zmienne ? Odpowiedź: za pomocą kluzuli `reduction(+:sum)`. Określa ona że:
  - każdy wątek w teamie ma prywatną kopię zmiennej sum inicjalizowaną wartością zmiennej sprzed regionu równoległego.
  - przy wyjściu z regionu przez wątek obliczana jest suma (znak +, możliwe są też inne operatory) prywatnych kopii i zapamiętywana w zmiennej sum wątku głównego (master).

# Całkowanie: wersja z klauzulą reduction

```
double Sum(int N,double a,double b)
{
int Trials=N/size,i;
double sum=0.0;
double h=(b-a)/(double)N;

#pragma omp parallel for default(none)firstprivate(N,h,a)reduction(+:sum)
for(i=0;i<N;i++) {
   sum+=f(a+i*h+h/2);
}
return sum*h;
}
```

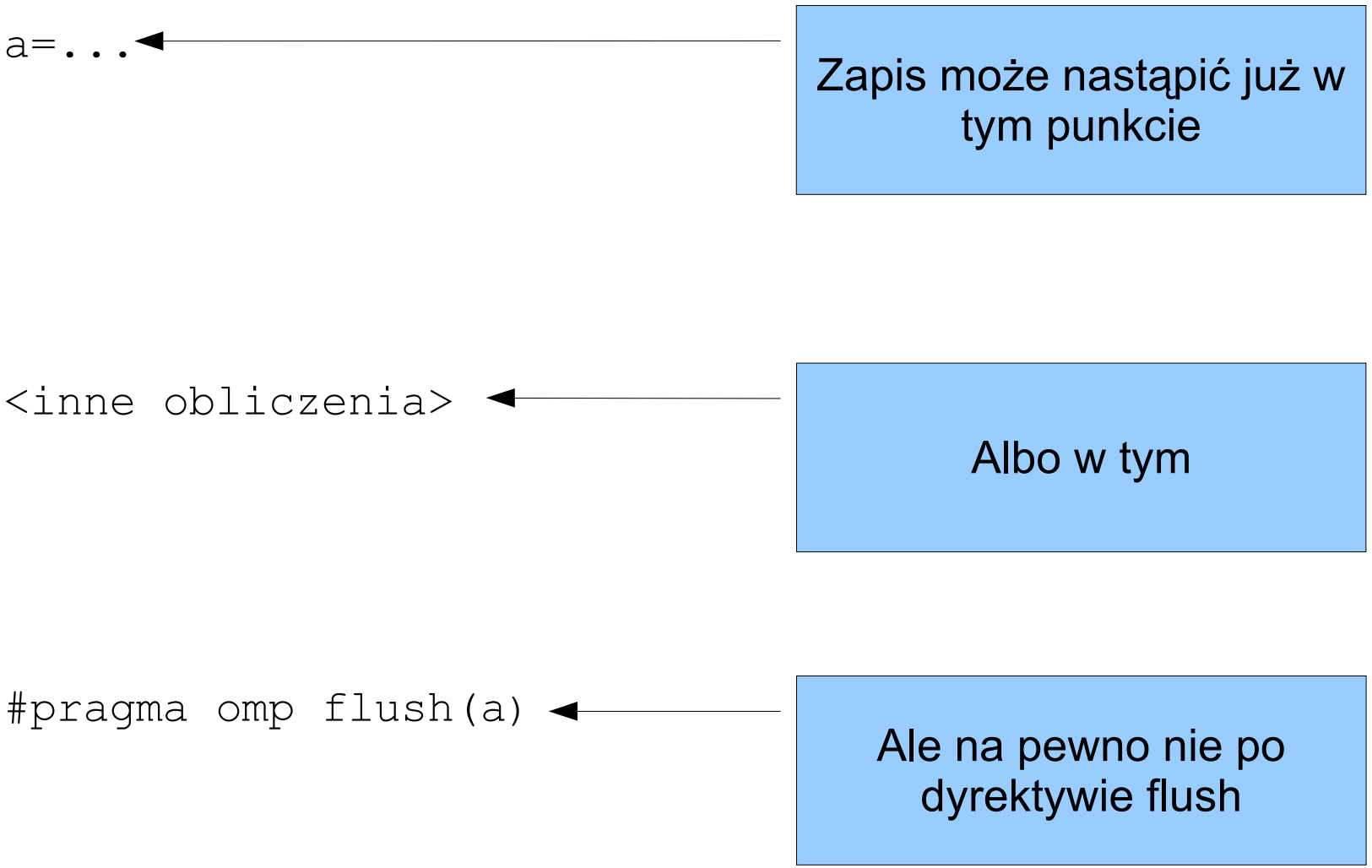
- Ciekawostka: program skompiluje się do wersji jednowątkowej na kompilatorze nie obsługującym OpenMP
- Brak sekcji krytycznej sprawia, że wątki w pętli wykonują się z maksymalną szybkością, a synchronizacja nastąpi dopiero gdy zakończą swoje iteracje (podczas dodawania prywatnych kopii zmiennej `sum`).

# Dyrektywa flush

- Kiedy zmiana zmiennej współdzielonej jest widoczna dla pozostałych wątków ?
- Gdyby zmiana takiej zmiennej była widoczna natychmiast, to kompilator nie mógłby generować kodu przechowującego zmienne w rejestrach procesora.
- W przypadku zmiennych volatile (ulotnych) każda modyfikacja zmiennej wiąże się z zapisem do pamięci a odczyt z odczytem z pamięci.
- Standard OpenMP stwierdza, że wszelkie modyfikacje są zapisane do pamięci, a przez to widoczne innym wątkom w ***punktach synchronizacji***. To samo dotyczy odczytów z pamięci.
- Niejawne punkty synchronizacji to
  - Jawna i niejawna synchronizacja barrierowa
  - Wejście i wyjście z  regionu krytycznego
  - Wejście i wyjście z funkcji operującej na zamkach
- Jawnie punkt synchronizacji możemy wstawić przy pomocy dyrektywy

```
#pragma omp flush(lista_zmiennych)
```

# Zapis zmiennej do pamięci



# Dyrektywa flush - przykład

```
int data,flag
#pragma omp parallel sections shared(data,flag) num_threads(2)
    {
       #pragma omp section
       {
          read(&data);
          #pragma omp flush(data)
          flag = 1;
          #pragma omp flush(flag)
          // Do more work.
       }

       #pragma omp section
       {
          while (!flag) {
             #pragma omp flush(flag)
          }
          #pragma omp flush(data)
          process(&data);
       }
    }
}
```

# Wydajność - minimalizuj sekcje krytyczne

```
#pragma omp parallel shared(a,b) private(c,d)
{
// tu jakiś kod
#pragma omp critical
    {
       a+=2*c;
       b=d*d;
    }
// tu dalszy kod
}
```

- Operacja zapisu do zmiennej współdzielonej zmiennej a musi być wewnątrz sekcji krytycznej (niebezpieczeństwo wyścigu). Jednakże przypisanie wartości zmiennej prywatnej d możemy bezpiecznie przenieść poza sekcję krytyczną.
- Wtedy możemy zastąpić sekcję krytyczną dyrektywą `atomic`.

```
#pragma omp parallel shared(a,b) private(c,d)
{
// tu jakiś kod
#pragma omp critical
       a+=2*c;
c=d*d;
// tu dalszy kod
}
```

# Wydajność - maksymalizuj regiony równoległe

```
#pragma omp parallel shared(a,b) private(c,d)
for(i=0;i<n;i++)
   for(j=0;j<n;j++)
#pragma omp parallel for
      for(k=0;k<n;k++)
         { ........}
```

- Koszt wejścia do i wyjścia z regionu równoległego ponoszony $n^2$ razy.

```
#pragma omp parallel private(i,j)
for(i=0;i<n;i++)
   for(j=0;j<n;j++)
#pragma omp for
      for(k=0;k<n;k++)
         {........}
```

- Koszt wejścia do i wyjścia z regionu równoległego ponoszony 1 raz.
- Zresztą należy rozważyć czy nie możemy zrównoleglić którejś z zewnętrznych pętli for.

# Wydajność – unikaj współdzielenia (w tym fałszywego) połączonego z zapisem.

- Przypomnienie z drugiego wykładu: gdy wiele wątków wykonuje zapisy do współdzielonego wiersza pamięci cache następuje znaczne spowolnienie systemu (konieczność uruchomienia protokołu spójności).
- Niech nt jest liczbą wątków (przykład celowo źle skonstruowany)

```
#pragma omp parallel for shared(nt,a) schedule(static,1)
for(int i=0;i<nt;i++)
   a[i]+=i;
```

- Statyczny rozdział puli iteracji algorytmem round-robin po jednej między wątki: wątek zerowy aktualizuje a[0], pierwszy a[1], drugi a[2], etc...
- Z punktu widzenia języka programowania wątki nie współdzielą elementów a.
- Ale a[0],a[1],a[2] mają duże szanse na rezydowanie w jednym wierszu pamięci. Z punktu widzenia hardware'u może być to współdzielenie.

# Wydajność - uwaga na funkcje biblioteczne !!!

- Pewien student (obecny na wszystkich wykładach z obliczeń równoległych !!!) zaobserował spowolnienie zamiast przyspieszenia w następującym kodzie:

```
#pragma omp parallel
{
  ..............
  x=rand();
  ................
};
```

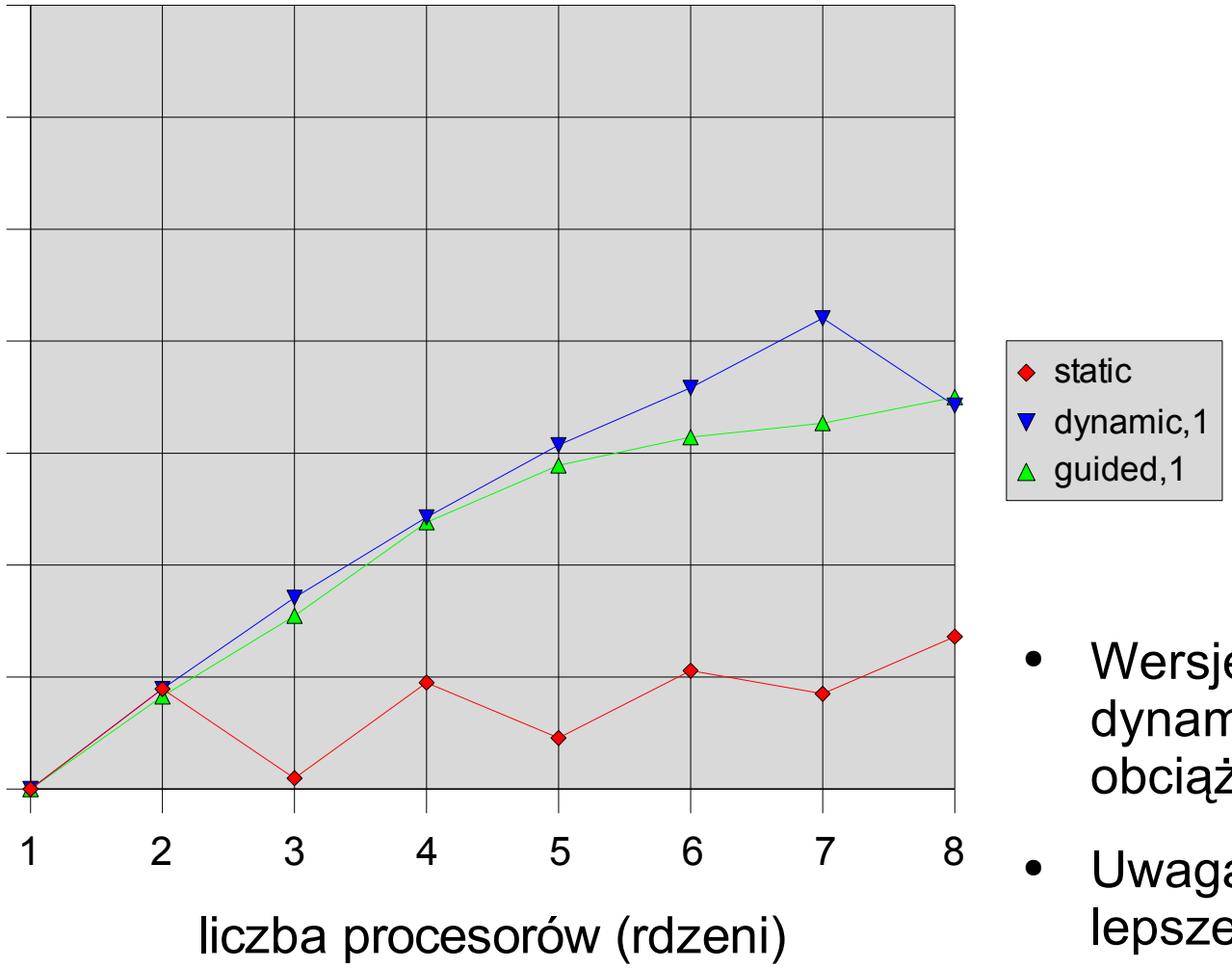
- Co robi funkcja rand. Zwraca nową liczbę losową na podstawie ***zmiennej globalnej*** będącej stanem generatora liczb losowych i ***aktualizuje stan przechowywany w tej zmiennej***.
- Mamy doczynienie ze współdzieleniem spowalniającym program a ponadto z wyścigiem (dlaczego z wyścigiem).
- Wyjście: zastosować funkcję która pozwala na jawne przekazanie wskaźnika na stan generatora; jedna instancja stanu generatora na jeden wątek. Ponadto każdy wątek powinien używać innego ziarna (dlaczego ???).
- Co się dzieje gdy wołamy malloc/free/new/delete z wielu wątków ? ***Na szczęście*** funkcje te wykonywane są w sekcjach krytycznych (są bezpieczne ze względu na wielowątkowść ang. thread safe). ***Na nieszczęście*** sekcje krytyczne serializują kod. Jeżeli wątki dużo czasu spędzają wewnętrz malloc/free to ....

# Wydajność - dbaj o zrównoważenie obciążenia

```
void CalcImage(int Width,int Height)
{
#pragma omp parallel for schedule(runtime)
        for(int i=0;i<Height;i++)
                CalcImgRow(Image[i],i,Width,Height);
}
```

- Omawiany już, przy okazji biblioteki MPI, przykład ze zbiorem Mandlebrota. Czas wykonania funkcji `CalcImgRow` silnie zależy od numeru wiersza i.
- Klauzula `schedule(runtime)` nakazuje podział iteracji pętli for w zależności od wartości zmiennej środowiskowej OMP_SCHEDULE.
- Zmienna ta przyjęła 3 wartości:
  - static
  - dynamic,1
  - guided,1
- Obliczenia na klastrze Mordor2: dwa procesory czterordzeniowe; razem 8.

# Przyspieszenie




- Wersje (dynamic,guided) z dynamicznym równownoważeniem obciążenia spisują się o wiele lepiej.
- Uwaga: guided, dynamic nie zawsze lepsze niż static (dlaczego ?)
- Widoczny wpływ prawa Amdahla - część sekwencyjna m.in zapis wynikowego obrazu do pliku

# Poprawność - eliminuj wyścigi (ang. race)

```
double a[n],b[n];
.....................
#pragma omp parallel for
for(int i=0;i<n-1;i++)
    a[i]=a[i]+b[i]
```

- Ta pętla jest poprawnie zrównoleglona. Kolejna pętla zawiera zależność pomiędzy iteracjami (ang. loop carried dependence)

```
#pragma omp parallel for
for(int i=0;i<n-1;i++)
    a[i]=a[i+1]+b[i]
```

- Ta pętla jest zrównoleglona błędnie. Otóż różne wątki usiłują wykonać współbieżnie różne iteracje pętli jeżeli iteracja i+1 zostanie wykonana przed iteracją i, a[i+1] zostanie wyliczone wcześniej. A zatem do obliczenia a[i] użyjemy „nowej" (a nie „starej" jak w wersji sekwencyjnej) wartości a[i+1].
- W przypadku wyścigu błędny wynik może, ale nie musi wystąpić; jest to niedeterministyczne. Fakt wystąpienia błędu może być uwarunkowany: (a) liczbą wątków (b) obciążeniem systemu (c) danymi wejściowymi

# Eliminacja zależności między pętlami

```
double c[n];
#pragma omp parallel for
for(int i=0;i<n;i++)
    c[i]=a[i]

#pragma omp parallel for
for(int i=0;i<n-1;i++)
    a[i]=c[i+1]+b[i]
```

- Kod po przetransformowaniu do powyższej wersji został poprawnie zrównoleglony.

# Inne pułapki

- Uwaga na domyślne typy zmiennych (zmienna staje się współdzielona, a myślimy że jest prywatna) - użycie default(none).
- Zmienne prywatne są niezainicjalizowane, a ich wynik po wyjściu z pętli jest nieokreślony - użycie lastprivate/firstprivate.
- Funkcje biblioteczne muszą być bezpieczne ze względu na wielowątkowość (ang. thread safe) - przykład z funkcją rand.
- Większość dyrektyw np. #pragma omp for, #pragma omp barrier musi być wykonana przez wszystkie wątki. Poniższy kod jest błędny:

```
#pragma omp parallel
if (omp_get_thread_num()>0) {
   #pragma omp parallel for
      for(int i=0;i<n;i++)
              ............
}
```

# Podsumowanie

- OpenMP jest standardem programowania systemów z pamięcią wspólną.
- Określa standard programowania wielowątkowego i jest API wyższego poziomu niż POSIX threads.
- Opiera się na dyrektywach #pragma i wymaga wsparcia kompilatora.
- Jest rozpowszechniony w przemyśle software'owym (standard przemysłowy) i dlatego warto go poznać.
- Wraz z rozpowszechnianiem się procesorów wielordzeniowych jego znaczenie będzie rosnąć.