Przemysław Szymoniak

Przetwarzanie współbieżne. Programowanie równoległe i rozproszone.

Sprawozdanie z laboratorium 11.
Celem laboratorium było nabycie umiejętności pisania programów w środowisku OpenMP z wykorzystaniem puli wątków.

W ramach zajęć zrealizowałem następujące kroki:

- Pobrałem i rozpakowałem wszystkie wymagane pliki, utworzyłem wymagane katalogi oraz uruchomiłem program.
- Dodanie definicji zadań dla wersji równoległej wyszukiwania liniowego:

```
int i;
double a_max_local = a_max;
#pragma omp task private(i) shared(a_max) firstprivate(A, p_task, k_task, a_max_local) default(none)
{
    for(i=p_task; i<=k_task; i++)
        if(a_max_local < A[i])
            a_max_local = A[i];

    #pragma omp critical (cs_a_max)
        {
          if(a_max < a_max_local) a_max = a_max_local;
        }


} // end task definition
```

- W funkcji „merge_sort_openmp” realizującej sortowanie przez scalanie zdefiniowano rozrost drzewa na 2 poziomy.
  `#pragma omp task final(poziom>max_poziom) default(none) firstprivate(A,p,r,q1,poziom)` – generujemy zadania do wykonania przez wątki w sposób asynchroniczny. Dopóki poziom mniejszy od `max_poziom` kolejne zadania wykonywane sa od razu, `default(none)` to ustawienie, że domyślnie żadna zmienna nie jest prywatna ani wspólna, `firstprivate` zaś to zmienne inicjowane przy użyciu wartości zmiennych o nazwach jak w wątku głównym. `#pragma omp taskwait` wymusza oczekiwanie na zakończenie wykonywania wcześniej utworzonych zadań.
- Funkcja „`merge_sort_openmp_2`” różni się od poprzedniej posiadaniem 3 poziomów rozrostu oraz wykorzystaniem funkcji sortowania szybkiego dla finałowego wykonania zadania (sprawdzanie w warunku `if(omp_in_final())`) w innym wypadku wywołuje samą siebie dzieląc podzbiór na mniejsze części. Po skończeniu podziału algorytm zaczyna scalać tablice w posortowanej kolejności.
- Funkcja „`merge_sort_openmp_4`” wykorzystuje zagnieżdżenia aktywowane przez `omp_set_nested(1);`
  `#pragma omp parallel sections default(none) firstprivate(A,p,r,q1)` – jedyną zmianą w stosunku do poprzedników jest użycie dyrektywy `sections` służy ona do nie iteracyjnego podziału bloku pomiędzy grupę wątków. Każdy blok tworzymy przez użycie dyrektywy `section`. W taki właśnie sposób tworzymy 2 bloki a każdy z nich dzielimy później przy użyciu tych samych dyrektyw na 2 następne bloki.
- Algorytm sortowania binarnego z wykorzystaniem sortowania liniowego po przekroczeniu poziomu:

```
if(p<r)
{
    level++;
    int s=(p+r)/2;
    double max_1, max_2;
    #pragma omp task final( level > max_level) default(none) firstprivate(A,p,r,s,level) shared(max_1)
    {
        if(omp_in_final())
        {
            max_1=search_max(A,p,s);
        }
        else
        {
            max_1=bin_search_max_task(A,s+1,r,level);
        }
    }
#pragma omp task final( level > max_level ) default(none) firstprivate(A,p,r,s,level)
shared(max_2)
{
    if(omp_in_final())
    {
        max_2=search_max(A,p,s);
    }
    else
    {
        max_2=bin_search_max_task(A,s+1,r,level);
    }
}
#pragma omp taskwait
{
    if(max_1<max_2)
    {
        return max_2
    }
    else
    {
        return max_1;
    }
}
```

Wnioski:

- Dyrektywa task służy do generowania zadań do wykonania przez wątki w sposób asynchroniczny.
- Wykorzystanie hybrydowego sortowania (łączącego sortowanie binarne/merge i liniowe/szybkie) w znaczący sposób przyśpiesza prace algorytmu.
- Funkcja omp_in_final() zwraca true Jeśli znajdujemy się w zadaniu które jako ostatnie spełnia warunek podany w dyrektywie final.