Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Wydział Inżynierii Metali i Informatyki Przemysłowej

Sprawozdanie z Laboratorium: OpenMP

Przedmiot: Przetwarzanie Równoległe i Rozproszone

Kierunek: Inżynieria Obliczeniowa

Autor: Filip Rak

Prowadzący ćwiczenia: dr inż. Mateusz Sitko

Data: 19 Grudnia 2024

Numery lekcji: 9, 10

Grupa laboratoryjna: 4

Cel Zajęć

Celem zajęć było zapoznanie nas z interfejsem programowania aplikacji OpenMP oraz jego wybranymi mechanizmami, które wykorzystaliśmy do zrównoleglania kodu w języku C.

Użycie OpenMP

Weryfikacja obecności OpenMP w kompilatorze

```c
int main()
{
    #ifdef _OPENMP
        printf("OpenMP jest dostępne\n");
    #else
        printf("OpenMP nie jest dostępne.\n");
}
```

Kompilacja z GCC

```
gcc -fopenmp -o example example.c
```

-fopenmp: Włącza obsługę OpenMP.

-example.c: Nazwa pliku wejściowego.

Wybrane Mechanizmy OpenMP

Funkcje biblioteki `<omp.h>`

- `int omp_get_max_threads()` - zwraca maksymalną ilość wątków.
- `int omp_get_thread_num()` - zwraca identyfikator wywołującego wątku.
- `void omp_set_num_threads(4)` – ustawia ilość używanych wątków.
- `void omp_set_nested(val)` – ustawia flagę pozwalająca na zagnieżdżone zrównoleglanie. Dla val będącego wartością `true`/`false` lub 1/0.
- `int omp_get_num_threads()` – zwraca ilość wątków w sekcji równoległej.
- `double omp_get_wtime()`– zwraca czas w sekundach.
- `int omp_in_final()` – zwraca 1 jeżeli warunek w klauzuli *final* został spełniony, 0 w innym wypadku.

Dyrektywy kompilatora

Nie wymagają biblioteki `<omp.h>`

#pragma omp – przedrostek dyrektyw OpenMP.

- **parallel**: sekcja równoległa, gdzie każdy wątek wykonuje ten sam kod.
- **parallel for**: zrównoleglenie pętli. Iteracje pętli są automatycznie rozdzielane między wątki.
- **shared(***args***)**: podane zmienne są wspólne między wątkami. Każdy wątek widzi i modyfikuje tę samą kopię zmiennej.
- **private(***args***)**: podane zmienne są prywatne dla każdego wątku. Każdy wątek ma swoją niezależną kopię zmiennej, ale kopie nie są inicjalizowane.
- **firstprivate(***args***)**: podane zmienne są prywatne dla każdego wątku, ale są zainicjalizowane wartościami z sekcji sekwencyjnej.
- **default(***arg***)**: określa domyślny dostęp wątków do zmiennych.
    - *none*: brak domyślnego dostępu do zmiennych. Wszystkie zmienne muszą być jawnie zadeklarowane jako private, shared, itp.
    - *shared*: domyślny dostęp współdzielony.
    - *private*: domyślny dostęp prywatny. Zmienne są niezainicjalizowane.
- **reduction**(*type*:*var*): wykonuje operacje redukcji na wspólnej zmiennej wynikowej. Kazdy wątek ma kopie lokalną zmiennej *var* wątek wykonuje na koniec na niej operacje *type*.
    - *type*: rodzaj operacji.
        - +: wyniki lokalne są sumowane.
        - *: wyniki lokalne są mnożone.
        - max: zapisywana jest maksymalna wartość.
        - min: zapisywana jest minimalna wartość.
        - Inne: Możliwe są także inne operatory, np. logiczne (&, |, ^).
    - *var*: zmienna.
- **schedule(***type***,** *size***):** definiuje sposób podziału iteracji pętli pomiędzy wątki.
    - *type*: harmonogram.
        - static: stały podział pracy . Domyślnie: liczba iteracji / liczbę wątków.
        - dynamic: dynamiczny podział pracy. Te wątki, które skończyły pracę dostają kolejny zestaw iteracji. Domyślnie: 1
    - *size*: liczbowa wartość określająca wielkość zestawu iteracji.
    - Dla typu static z rozmiarem równym 1, otrzymamy typową dekompozycję cykliczną, dla rozmiaru większego niż 1 – blokową.
- **ordered**: sekcja pętli , która będzie zsynchronizowana pomiędzy wątkami. Pętla musi zawierać klauzulę **ordered**.

- **critical(***nazwa***)**: sekcja krytyczna do której, w danym czasie, będzie dopuszczony maksymalnie jeden wątek.
    - *nazwa*: opcjonalny identyfikator sekcji krytycznej. Wejście wątku do dowolnej sekcji krytycznej o tej nazwie zablokuje wejście do innych sekcji krytycznych identyfikowanych tą samą nazwą i vice versa.
- **final(***warunek***):** ustawia flagę zwracaną przez funkcje `int omp_in_final() –` na wartość logiczną podanego warunku (1 lub 0).
- **num_threads(***number***)**: ustawia ilość wątków w danej sekcji na number.
- **barrier:** synchronizacja wszystkich wątków w bieżącej sekcji. Wątki nie przejdą dalej, dopóki wszystkie nie osiągną tej dyrektywy
- **atomic:** zabezpiecza pojedynczą operację modyfikacji zmiennej przed wyścigami danych.
- **task:** sekcja tworzy zadanie, które może zostać obsłużone przez dowolny wątek.
- **taskwait**: zatrzymuje wątek wywołujący dopóki wszystkie zadania **task** nie zostaną wykonane.
- **parallel sections**: sekcja w której deklarowane są sekcje (**section)** kodu równoległego do wykonania na pojedynczym wątku. Wątek wywołujący czeka na końcu tej sekcji aż wątki potomne skończą pracę.
- **section**: sekcja kodu wykonywanego przez wątek wewnątrz sekcji **sections**. W odróżnieniu od task nie ma dynamicznego przydzielania do wolnego wątku.

Zmienna Środowiskowa

Modyfikacja ilości wątków w OpenMP nie musi wymagać zmian w kodzie źródłowym ani rekompilacji. Zmienna środowiskowa *OMP_NUM_THREADS* ustawia liczbę wątków jaka będzie używana przez OpenMP.

Przykład modyfikacji zmiennej w terminalu systemu Linux:

```
export OMP_NUM_THREADS=4
```

W OpenMP mamy trzy sposoby zmiany liczbę wątków, gdzie każdy kolejny jest ma priorytet ponad poprzedni.

- Zmienna środowiskowa *OMP_NUM_THREADS*: Ustawia liczbę wątków dla każdego programu OpenMP.
- Funkcja `void omp_set_num_threads(int)` – ustawia liczbę wątków w danym programie od momentu wywołania. Ma priorytet ponad zmienną środowiskową.
- Klauzula **num_threads(***number***)**: ustawia liczbę wątków dla konkretnej sekcji równoległej. Priorytet nad zmienną środowiskową oraz funkcją.

Przykładowe fragmenty kodu w języku C

Proste zrównoleglenie pętli.

Kod źródłowy:

```cpp
#include <iostream>
#include <omp.h>

int main()
{
	// Array of strings to reverse
	std::string arr[] = { "aA", "bB", "cC", "dD", "eE" };
	int arr_size = sizeof(arr) / sizeof(std::string);

	// Sum of all the character
	int character_sum = 0;

	std::cout << "-------- Parallel Section --------\n";

	#pragma omp parallel for default(none) reduction(+:character_sum) shared(arr)
	for (int i = 0; i < arr_size; i++)
	{
		// Swap the characters within a string
		std::swap(arr[i][0], arr[i][1]);
		character_sum += arr[i].size();

		// Print out the reversed string
		int thread_id = omp_get_thread_num();

		#pragma omp critical
		std::cout << "Thread: " << thread_id << "\tReversed string: "
			<< arr[i] << "\tLocal sum: " << character_sum << "\n";
	}

	// Print out the total number of characters
	int thread_id = omp_get_thread_num();
	std::cout << "\n-------- Sequential Section -------\n";
	std::cout << "Thread: " << thread_id << "\tTotal characters: "
		<< character_sum << "\n";
}
```

Wydruk:

```
-------- Parallel Section --------
Thread: 0       Reversed string: Aa     Local sum: 2
Thread: 1       Reversed string: Bb     Local sum: 2
Thread: 2       Reversed string: Cc     Local sum: 2
Thread: 3       Reversed string: Dd     Local sum: 2
Thread: 4       Reversed string: Ee     Local sum: 2

-------- Sequential Section -------
Thread: 0       Total characters: 10
```

Schemat ogólny wykorzystania zadań (task) z przykładem

```cpp
#include <iostream>
#include <cmath>
#include <omp.h>

int ascii_sum(std::string str)
{
	int sum = 0;
	for (int i = 0; i < str.size(); i++)
		sum += int(str[i]);

	return sum;
}

/* Main Function */
int main()
{
	// Exercise: Find the sum of all ascii character values within the string
	std::string text = "abcdefghijklmnoprtuvwxyz";
	int total_sum = 0;

	/* Parallel Section */
	#pragma omp parallel
	{
		// Single thread distributes work
		#pragma omp single
		{
			int thread_num = omp_get_num_threads();
			int per_thread = std::ceil((double)text.size() / thread_num);

			for (int i = 0; i < thread_num; i++)
			{
				// Every thread gets a block of characters to process
				int local_start = i * per_thread;
				int local_end = ((i + 1) * per_thread > text.size()) ?
					text.size() : (i + 1) * per_thread;

				// Create a task for a thread
				#pragma omp task default(shared) firstprivate(local_start, local_end)
				{
					// Sum the ascii characters within a given block
					int local_sum = 0;
					for (int j = local_start; j < local_end; j += 1)
						local_sum += int(text[j]);

					// Add the local sum to total sum showcasing atomic directive
					#pragma omp atomic
					total_sum += local_sum;
				}
			}

			// Wait for all tasks to finish before proceeding
			#pragma omp taskwait
		}
	}
	/* Sequential Section */
	// Print the results
	int seq_res = ascii_sum(text);
	int pal_res = total_sum;
	bool same_result = (seq_res == pal_res);

	std::cout << "Sequential sum:\t" << seq_res << "\n";
	std::cout << "Parallel sum:\t" << pal_res << "\n";
	std::cout << "Results are ok:\t" << same_result << "\n";
}
```

Wydruk:

```
Sequential sum: 2619
Parallel sum:   2619
Results are ok: 1
```

Schemat użycia sekcji (sections i section)

Kod źródłowy:

```
#include <iostream>
#include <omp.h>

int main()
{
    #pragma omp parallel
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                #pragma omp critical
                {
                    std::cout << "Section 0 is executed by thread "
                        << omp_get_thread_num() << "\n";
                    std::cout << "Section 0: Threads within the pool: "
                        << omp_get_num_threads() << "\n";
                }
            }

            #pragma omp section
            {
                #pragma omp critical
                std::cout << "Section 1 is executed by thread "
                    << omp_get_thread_num() << "\n";
            }

            #pragma omp section
            {
                #pragma omp critical
                std::cout << "Section 2 is executed by thread "
                    << omp_get_thread_num() << "\n";
            }
        }
    }

    return 0;
}
```

Wydruk:

```
Section 1 is executed by thread 4
Section 0 is executed by thread 3
Section 0: Threads within the pool: 12
Section 2 is executed by thread 0
```

Sekcja równoległa wykonywana osobno przez każdy wątek

Kod źródłowy:

```
#include <iostream>
#include <omp.h>

int main()
{
	#pragma omp parallel num_threads(3)
	{
		std::cout << "# ";

		#pragma omp barrier
		#pragma omp critical
		{
			int thread_id = omp_get_thread_num();
			std::cout << "\nThread: " << thread_id << " has finished";
		}
	}

	std::cout << "\n";
}
```

Wydruk:

```
# # #
Thread: 2 has finished
Thread: 1 has finished
Thread: 0 has finished
```