Practical Sessions 1&2 : OpenMP

1 Warm up part one: parallel regions

Let us consider the following code [Link] :

```
#include <stdio.h>
int main() {
  #pragma omp parallel
  printf("Hello\n");
  printf("World\n");
  return 0;
}
```

Compile this program without the `-fopenmp` option and run it. What do you observe?

Compile this program with the `-fopenmp` option, and run it. What is the default number of threads?

Change the number of threads dynamically by setting the value of the environment variable `OMP_NUM_THREADS` to the desired number of threads (we didn't study the OpenMP environment variables in class yet, but this one is handy). Which command did you use?

Change the number of threads statically using the `num_threads(n)` clause of the `parallel` directive. Which directive did you write?

Can the word "`World`" be displayed before the word "`Hello`"? If so, in which case? If no, why?

Modify the program so that each thread runs both `printf`.

2 Warm up part one: work sharing

Let us consider the following code [Link] :

```
#include <stdio.h>
#include <omp.h>
#define SIZE  100
#define CHUNK  10

int main() {
  int i, tid;
  double a[SIZE], b[SIZE], c[SIZE], sum = 0.;

  for (i = 0; i < SIZE; i++)
    a[i] = b[i] = i;

  #pragma omp parallel private(tid) reduction(+: sum)
  {
    tid = omp_get_thread_num();
    if (tid == 0)
      printf("Nb threads = %d\n", omp_get_num_threads());
    printf("Thread %d: starting...\n", tid);

    #pragma omp for
    for (i = 0; i < SIZE; i++) {
      c[i] = a[i] + b[i];
      sum += c[i];
      printf("Thread %d: c[%2d] = %g\n", tid, i, c[i]);
    }
  }
  printf("sum = %g\n", sum);
  return 0;
}
```

What does this program do?

What are the instructions executed by a single thread? By all threads?

What is the role of the OpenMP **`for`** directive?

What is the scope of `i` and `tid` variables? What is the scope of `a`, `b` and `c` variables?

What is the role of the `reduction` clause?

3 Program Parallelization

Parallelize the following codes using OpenMP. Code alterations may be necessary in addition to OpenMP directive insertion. If necessary, explain your code alterations, your work sharing and data scoping choices to get correct results. Then discuss the performance issues.

3.1 Matrix-Matrix Product

Study and parallelize the following code [Link] :

```
void matmat_kernel(double C[N][N], double A[N][N], double B[N][N]) {
  size_t i, j, k;

  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      C[i][j] = 0.;
      for (k = 0; k < N; k++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}
```

Make measurements and draw a curve of the execution time as a function of the number of threads for the following cases: (1) original code, (2) parallelized code with static scheduling and (3) parallelized code with dynamic scheduling. Discuss your results.

3.2 Calculating Pi

Study and parallelize the following code [Link] :

```
void pi_kernel(size_t nb_steps, double* pi) {
  double term;
  double sum = 0.;
  double step = 1./(double)nb_steps;

  for (size_t i = 0; i < nb_steps; i++) {
    term = (i + 0.5) * step;
    sum += 4. / (1. + term * term);
  }

  *pi = step * sum;
}
```

3.3 Enumeration Sort

Study and parallelize the following code [Lien] :

```
void enumeration_sort_kernel(double tab[N]) {
  size_t i, j;
  size_t* position = malloc(N * sizeof(size_t));
  double* copy     = malloc(N * sizeof(size_t));

  for (i = 0; i < N; i++) {
    position[i] = 0;
    copy[i] = tab[i];
  }

  for (j = 0; j < N; j++) {
    for (i = 0; i < N; i++) {
      if ((tab[j] < tab[i]) || ((tab[i] == tab[j]) && (i < j))) {
        position[i]++;
      }
    }
  }

  for (i = 0; i < N; i++)
    tab[position[i]] = copy[i];

  free(position);
  free(copy);
}
```

3.4 Bubble Sort

Implement a parallel version of the bubble sort by algorithm change as seen during the exercise session (see the session document). If necessary, use the keyword *odd-even sort* on a search engine.