IF5 – PAR1 : TP OpenMP

Jacquemin Thibault

Marleix Mathieu

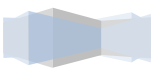


Table des Matières

Introduction

L'objectif de ce TP est de mesurer l'impact de la parallélisation sous OpenMP et d'initier les élèves à la programmation OpenMP.

La programmation sera effectuée sous Visual Studio. Les élèves enverront au cours du TP leur code pour le tester sur la lame 02 du Blade Center d'ESIEE Paris tournant sous Ubuntu. La connexion à la machine s'effectue via SSH sous Putty et les fichiers sont envoyés via WinSCP.

I. Prise en Main

On observe d'abord les caractéristiques de l'architecture à savoir le nombre de processeurs présents et leur fréquence.

Le nombre de cœurs peut être obtenu par la commande « *getconf _NPROCESSORS_ONLN* » qui nous renvoie alors un nombre de 8 cœurs. La fréquence du processeur est obtenu grâce à la commande « *cat /proc/cpuinfo* » comme indiqué dans l'énoncé suivit d'un filtrage par la commande « *grep -i "cpu M"* » : l'affichage de */proc/cpuinfo* nous renvoi de nombreuses informations que nous n'utiliserons pas ici.

On observe donc que le processeur dispose d'une fréquence de 2000.010 Mhz sur chacun de ses 8 cœurs.

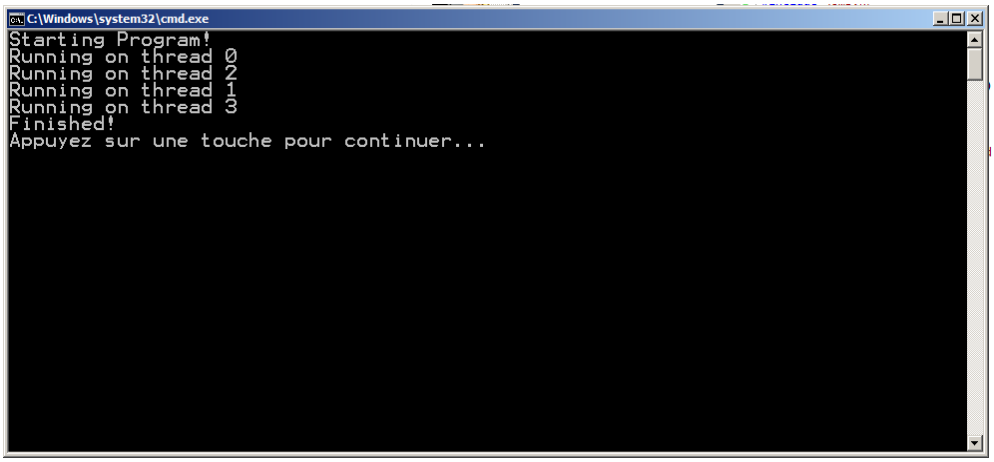
Le programme d'introduction à OpenMP demandé doit se diviser en 4 tâches qui vont chacune afficher leur numéro de rang avant d'afficher la terminaison du programme.

Le code de ce programme est donc le suivant :

```
#include "stdafx.h"
#include <omp.h>


int main(int argc, char* argv[])
{
   printf("Starting Program!\n");

   #pragma omp parallel
   {
      printf("Running on thread %d\n", (int)omp_get_thread_num());
   }

   printf("Finished!\n");

   return 0;
}
```

Pour que 4 tâches s'exécutent, on définit la variable OMP_NUM_THREADS à 4 grâce à la commande
« *seten OMP_NUM_THREADS 4* ».

Le rendu du code présenté peut être observé ci-dessus, lancer via la commande de Windows.

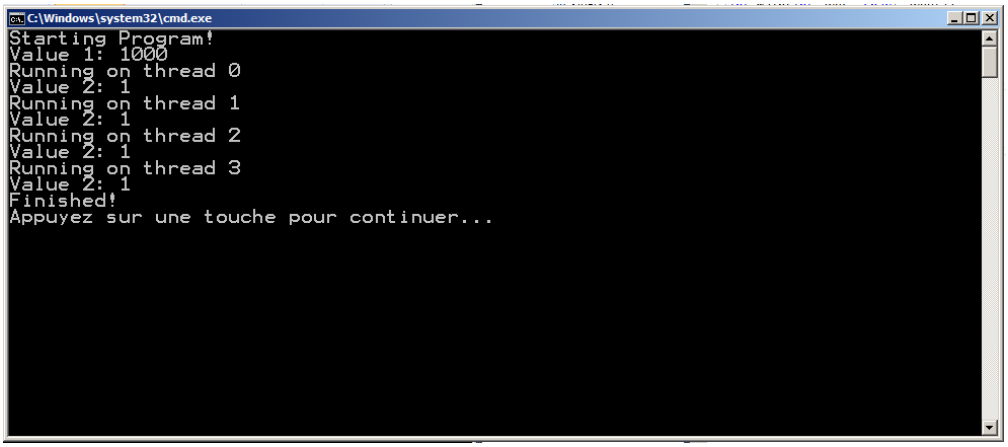


II. Variables privées

Le code programmé pour cet exercice est le suivant :

```
#include "stdafx.h"
#include <omp.h>


int main(int argc, char* argv[])
{
	printf("Starting Program!\n");
	int VALUE1 = 1000, VALUE2 , VALUE2 = 2000;
	printf("Value 1: %d\n", VALUE1);
	#pragma omp parallel private (VALUE2)
	{
		VALUE2 = 0;
		printf("Running on thread %d\n", (int)omp_get_thread_num());
		printf("Value 2: %d\n", ++VALUE2);
	}

	printf("Finished!\n");

	return 0;
}
```

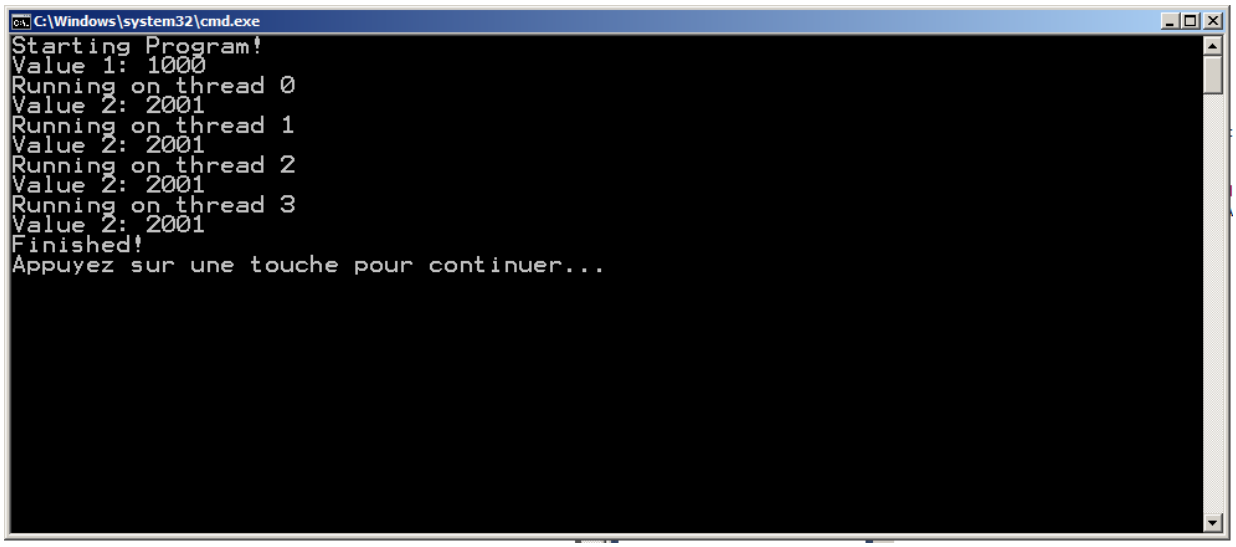
Le rendu de ce code lancé sous l'exécutable commande de Windows donne le résultat suivant.



On observe que VALEUR2 qui devrait être égale à 2001 est ici égale à 1.
En effet, en lisant un extrait de la documentation présente à l'adresse, http://msdn.microsoft.com/en-US/library/c3dabskb%28v=vs.80%29.aspx, on constate qu'une variable déclarée private va être de nouveau instancié par le thread avec le constructeur par défaut. Sa valeur est donc égale à 0 dans chaque thread et suivant à 1 après l'incrémentation.

A contrario, une variable déclarée à l'aide de firstprivate se verra, comme spécifié dans la documentation à l'adresse http://msdn.microsoft.com/en-us/library/zkh091c4%28v=vs.80%29.aspx, initialisée à la valeur précédent la parallélisation dans le code.

Après vérification, on obtient bien le résultat souhaité, à savoir :

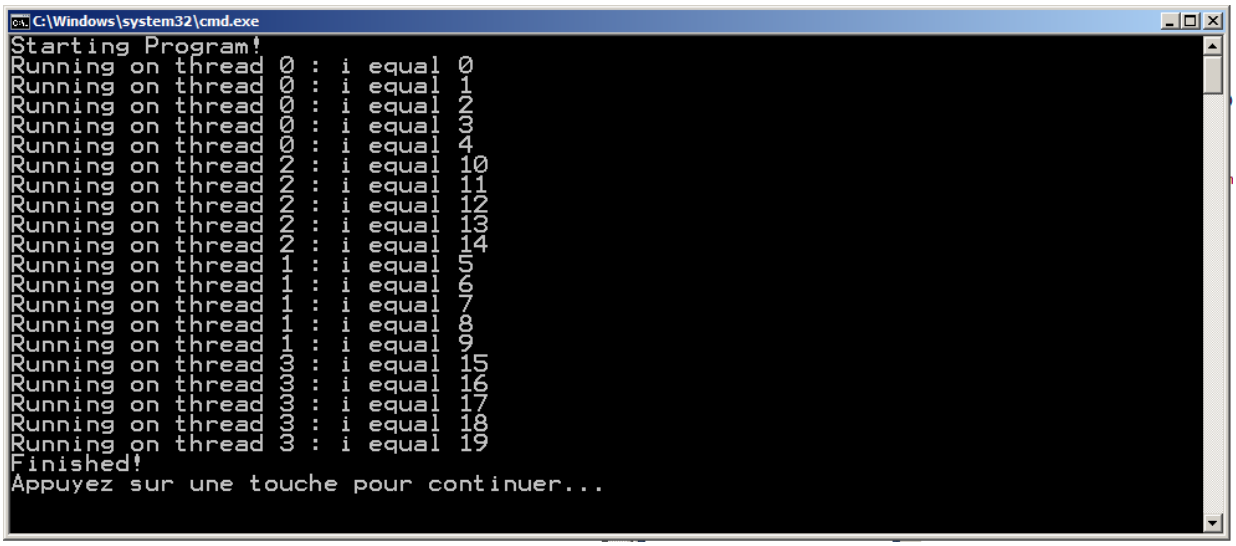


III. Boucles parallèles

Le code demandé au sein de cet exercice a été implémenté de la façon suivante :

```
#include "stdafx.h"
#include <omp.h>


int main(int argc, char* argv[])
{
	printf("Starting Program!\n");
	int i = 0;
	#pragma omp parallel for
		for(i=0;i<20;i++){
			printf("Running on thread %d : i equal %d\n", (int)omp_get_thread_num(),i);
		}

	printf("Finished!\n");

	return 0;
}
```

Les résultats obtenus sont alors les suivants :



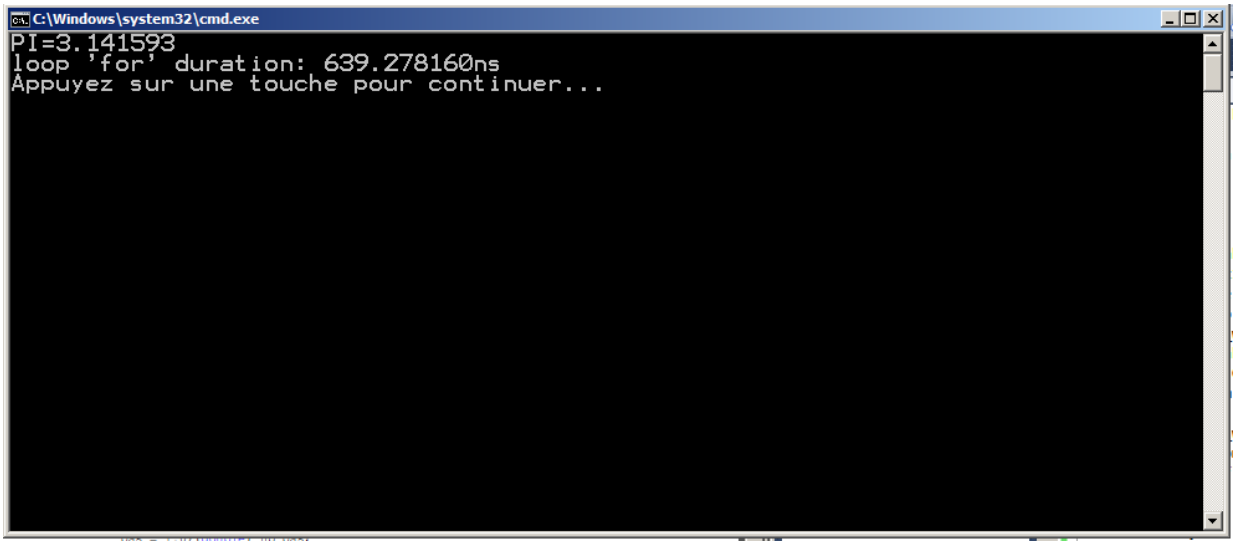
On observe que l'ordre obtenu n'est pas forcément celui espéré puisque l'ordonnancement des tâches dépend du processeur. Dans le cas de besoin précis il est donc nécessaire d'ajouter des verrous et des mutex pour s'assurer de l'exécution correcte de son code.

IV. Calcul de PI

Le code permettant le calcul de PI parallèle implémenté est le suivant :

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

int main(int argc, char* argv[])
{
        static long nb_pas = 100000;
   omp_set_num_threads(16);
   if(argc == 3){
     omp_set_num_threads(atoi(argv[1]));
        nb_pas = atoi(argv[2]);
        }
        double pas, t1, t2;
        int i; double x, pi, som = 0.0;
        pas = 1.0/(double) nb_pas;
        t1 = omp_get_wtime();
        #pragma omp parallel for reduction(+ :som) private (x)
        for (i=0;i< nb_pas; i++){
                x = (i + 0.5)*pas;
                som = som + 4.0/(1.0+x*x);
        }
        t2 = omp_get_wtime();
        pi = pas * som;
        printf("PI=%f with a step number of %ld and %f task(s)\n",pi,nb_pas,atoi(argc[1]));
        printf("loop 'for' duration: %fms\n", (double)(t2-t1)*1000);

        return 0;
}
```

On test d'abord un lancement sur la machine de développement. Le résultat obtenu nous donne le résultat de la figure ci-dessous.



Une durée de 639 ms pour une précision de 100000.

Ceci effectué, nous copions nôtre code sur la machine blade02.esiee.fr et exécutons le programme modifié pour prendre en compte une variation du nombre de tâche et de la précision.

Le résultat est tel que :

```
if5par1@blade02:~/jacquemt$ nano openmp_1_4.cpp
if5par1@blade02:~/jacquemt$ icpc -openmp -o pi2 openmp_1_4.cpp
if5par1@blade02:~/jacquemt$ !s
sh bench.sh
PI=3.141593 with a step number of 100000 and 1 task(s)
loop 'for' duration: 0.724077ms
PI=3.141593 with a step number of 100000 and 2 task(s)
loop 'for' duration: 0.587940ms
PI=3.141593 with a step number of 100000 and 3 task(s)
loop 'for' duration: 0.511169ms
PI=3.141593 with a step number of 100000 and 4 task(s)
loop 'for' duration: 0.542879ms
PI=3.141593 with a step number of 100000 and 5 task(s)
loop 'for' duration: 9.744883ms
PI=3.141593 with a step number of 100000 and 6 task(s)
loop 'for' duration: 15.267849ms
PI=3.141593 with a step number of 100000 and 7 task(s)
loop 'for' duration: 21.422148ms
PI=3.141593 with a step number of 100000 and 8 task(s)
loop 'for' duration: 58.769941ms
PI=3.141593 with a step number of 100000 and 9 task(s)
loop 'for' duration: 0.824213ms
PI=3.141593 with a step number of 100000 and 10 task(s)
loop 'for' duration: 0.953913ms
PI=3.141593 with a step number of 100000 and 11 task(s)
loop 'for' duration: 1.008987ms
PI=3.141593 with a step number of 100000 and 12 task(s)
loop 'for' duration: 1.113176ms
PI=3.141593 with a step number of 100000 and 13 task(s)
loop 'for' duration: 1.133204ms
PI=3.141593 with a step number of 100000 and 14 task(s)
loop 'for' duration: 1.148939ms
PI=3.141593 with a step number of 100000 and 15 task(s)
loop 'for' duration: 1.202822ms
PI=3.141593 with a step number of 100000 and 16 task(s)
loop 'for' duration: 1.283169ms
```

On constate de meilleurs résultats lors de l'exécution sur la lame. Cependant la variation du nombre de tâche augmente ici le temps d'exécution : le temps d'intercommunication entre les tâches devient plus grand que celui de calcul.

Pour une plus grande précision, on constate que l'exécution parallèle augmente grandement le temps d'exécution :

```
if5par1@blade02:~/jacquemt$ sh bench.sh
PI=3.141593 with a step number of 10000000 and 1 task(s)
loop 'for' duration: 50.903082ms
PI=3.141593 with a step number of 10000000 and 2 task(s)
loop 'for' duration: 25.703192ms
PI=3.141593 with a step number of 10000000 and 3 task(s)
loop 'for' duration: 17.239094ms
PI=3.141593 with a step number of 10000000 and 4 task(s)
loop 'for' duration: 13.096094ms
PI=3.141593 with a step number of 10000000 and 5 task(s)
loop 'for' duration: 20.169973ms
PI=3.141593 with a step number of 10000000 and 6 task(s)
loop 'for' duration: 13.453007ms
PI=3.141593 with a step number of 10000000 and 7 task(s)
loop 'for' duration: 33.028126ms
PI=3.141593 with a step number of 10000000 and 8 task(s)
loop 'for' duration: 16.777992ms
PI=3.141593 with a step number of 10000000 and 9 task(s)
loop 'for' duration: 11.961937ms
PI=3.141593 with a step number of 10000000 and 10 task(s)
loop 'for' duration: 10.961056ms
PI=3.141593 with a step number of 10000000 and 11 task(s)
loop 'for' duration: 14.673948ms
PI=3.141593 with a step number of 10000000 and 12 task(s)
loop 'for' duration: 9.809971ms
PI=3.141593 with a step number of 10000000 and 13 task(s)
loop 'for' duration: 12.598038ms
PI=3.141593 with a step number of 10000000 and 14 task(s)
loop 'for' duration: 11.860132ms
PI=3.141593 with a step number of 10000000 and 15 task(s)
loop 'for' duration: 11.178017ms
PI=3.141593 with a step number of 10000000 and 16 task(s)
loop 'for' duration: 10.676861ms
```

V. Algorithme de l'élimination de Gauss

Partis du code du Gauss en séquentiel du tp sur PVM, il nous fallait identifier la partie parallèle du calcul qui se situe au niveau de la 1ère boucle imbriquée pour ajouter la pragma '*omp parallel for*'.

Le code implémenté est le suivant :

```
#include <omp.h>
#include <stdlib.h>
#include <stdio.h>

#include <errno.h>

void matrix_load ( char nom[], double *tab, int N ) {
  FILE *f;
  int i,j;

  if ((f = fopen (nom, "r")) == NULL) { perror ("matrix_load : fopen "); }
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      fscanf (f, "%lf", (tab+i*N+j) );
    }
  }
  fclose (f);
}

void matrix_save ( char nom[], double *tab, int N ) {
  FILE *f;
  int i,j;

  if ((f = fopen (nom, "w")) == NULL) { perror ("matrix_save : fopen "); }
  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      fprintf (f, "%8.2f ", *(tab+i*N+j) );
    }
    fprintf (f, "\n");
  }
  fclose (f);
}

void matrix_display ( double *tab,int  N ) {
  int i,j;

  for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
      printf ("%8.2f ", *(tab+i*N+j) );
    }
    printf ("\n");
  }

}

void gauss ( double * tab, int N ) {
  int i,j,k;
  double pivot, t0, t1;

  t0 = omp_get_wtime();
  for ( k=0; k<N-1; k++ ){ /* mise a 0 de la col. k */
    /* printf (". "); */
```

```
    if ( fabs(*(tab+k+k*N)) <= 1.0e-11 ) {
      printf ("ATTENTION: pivot %d presque nul: %g\n", k, *(tab+k+k*N) );
      exit (-1);
    }
    #pragma omp parallel for
    for ( i=k+1; i<N; i++ ){ /* update lines (k+1) to (n-1) */
      pivot = - *(tab+k+i*N) / *(tab+k+k*N);
      for ( j=k; j<N; j++ ){ /* update elts (k) - (N-1) of line i */
        *(tab+j+i*N) = *(tab+j+i*N) + pivot * *(tab+j+k*N);
      }
      /* *(tab+k+i*N) = 0.0; */
    }
  }
  t1 = omp_get_wtime();

  printf ("computation time: %10.8f sec.\n", ((double) t1-t0)/1000000.0 );
}

main (int argc, char **argv) {
  int N, i, j, k;
  double *tab, pivot;
  char nom[255], nom_fichier[63];
  FILE *f;

  if (argc != 3){
    printf ("Usage: %s <matrix size> <matrix name>\n", argv[0]);
    exit (-1);
  }
  N = atoi ( argv[1] );
  strcpy(nom, argv[2]);
  if ( (tab=malloc(N*N*sizeof(double))) == NULL ) {
    printf ("Cant malloc %d bytes\n", N*N*sizeof(double));
    exit (-1);
  }

  matrix_load ( nom , tab, N );
  gauss ( tab, N );

  //sprintf ( nom+strlen(nom), ".result" );
  //matrix_save ( nom, tab, N );
}
```

Conclusion

Nous avons donc pu vérifier l'impact de la parallélisation d'algorithme sous OpenMP. Nous avons pu découvrir la programmation parallèle OpenMP sous Visual Studio.