Bach Khoa University Of Technology
Faculty of Computer Science & Engineering

Course: Parallel Processing
Lab #6 - OpenMP case studies & Speed Up Optimization

Minh Thanh Chung

September 24, 2017

Goal: This lab helps student to practice with case studies in programming with OpenMP: insufficient parallelism cases and thread-affinity aware.

Content: Section 1 shows insufficient parallelism cases, and then we can improve the performance from the baseline case. Section 2 introduces the problem with thread-affinity aware.

Result: After this Lab, student can understand the case studies that need to consider for optimizing the performance of parallel programs.

Contents

1 Threading insufficient parallelism

1.1 Read and understand the given source code

This lab demonstrates dealing with insufficient parallelism through strip-mining and loop collapse. Study the code in *worker.cc* and also take a look at the values of m and n in *main.cc* What do you think will happen in the parallel loop in *worker.cc*?
The source code includes three files:

- *main.cc*: the main program shows some configurations, then call the function named **SumColumns()**. After that, show the performance results from measuring the execution time of this function.
- *worker.cc*: implement the function - **SumColumns()**. We will modified this function to practice with insufficient parallelism case-studies.
- *Makefile*: to compile the program.

```
/* main.cc */
#include <malloc.h>
#include <cmath>
#include <omp.h>
#include <cstdio>

void SumColumns(const int m, const int n, long *M, long *s);

int main(){
     const int n = 100000000; /* the number of columns (inner dimension) */
     const int m = 4;     /* the number of rows (outer dimension) */

     long *matrix = (long *)_mm_malloc(sizeof(long) * m * n, 64);
     long *sums = (long *)_mm_malloc(sizeof(long) * m, 64); /* sum of matrix rows */

     const double HztoPerf = 1e-9 * double(m * n) * sizeof(long);

     const int nTrials = 10;
     double rate = 0;
     double dRate = 0;

     printf("\n\033[1mComputing the sums of elements in each row of a wide, short matrix.\033[0m\n");
     printf("Problem size: %.3f GB, outer dimension: %d, threads: %d (%s)\n\n",
          (double)(sizeof(long)) * (double)(n) * (double)(m) / (double)(1<<30),
          m, omp_get_max_threads(), "CPU");

     /* Initializing data */
     #pragma omp parallel for
          for(int i = 0; i < m; i++)
               for(int j = 0; j < n; j++)
                    matrix[i*n + j] = (long)i;

     printf("\033[1m%5s %10s %10s\033[0m\n", "Trials", "Time, s", "Perf(GB/s)");

     const int skipTrials = 2;

     /* Benchmarking SumColumns(...) */
```

```
    for(int trial = 1; trial <= nTrials; trial++){
        const double tStart = omp_get_wtime();
        SumColumns(m, n, matrix, sums);
        const double tEnd = omp_get_wtime();

        if(trial > skipTrials){
            rate += HztoPerf / (tEnd - tStart);
            dRate += (HztoPerf * HztoPerf) / ((tEnd - tStart) * (tEnd - tStart));
        }

        printf("%5d %10.3e %10.2f %s\n",
            trial, (tEnd - tStart), HztoPerf/(tEnd - tStart), (trial <= skipTrials ? "*" : ""));
        fflush(stdout);

        /* Verifying that the result is correct */
        for(int i = 0; i < m; i++){
            if(sums[i] != i * n)
                printf("Results are incorrect!\n");
        }
    }

    rate /= (double)(nTrials - skipTrials);
    dRate = sqrt(dRate / (double)(nTrials - skipTrials) - rate * rate);

    printf("-----------------------------------------------\n");
    printf("\033[1m%s %4s \033[42m%10.2f +- %.2f GB/s\033[0m\n",
        "Average performance:", "", rate, dRate);
    printf("-----------------------------------------------\n");
    printf("* - warm-up, not include in average\n");

    /* Free memory */
    _mm_free(sums);
    _mm_free(matrix);
}
```

```
/* worker.cc */
void SumColumns(const int m, const int n, long *M, long *s){
    /* Distribute rows across threads */
        for(int i = 0; i < m; i++){
            /* Private variable for reduction */
            long sum = 0;

            /* Vectorize inner loop */
            for(int j = 0; j < n; j++)
                sum += M[i*n + j];

            s[i] = sum;
        }
}
```

```
/* Makefile */
CXX = icpc
CXXFLAGS = -qopenmp -mkl
```

```
CPUFLAGS = $(CXXFLAGS) -xhost
OPTFLAGS = -qopt-report -qopt-report-file=$@.optrpt

CPUOBJECTS = main.o worker.o

TARGET = app-CPU

.SUFFIXES: .o .cc

all: $(TARGET) instructions

%-CPU: $(CPUOBJECTS)
	$(info )
	$(info Linking the CPU executable:)
	$(CXX) $(CPUFLAGS) -o $@ $(CPUOBJECTS)

.cc.o:
	$(info )
	$(info Compiling a CPU object file:)
	$(CXX) -c $(CPUFLAGS) $(OPTFLAGS) -o "$@" "$<"

instructions:
	$(info )
	$(info TO EXECUTE THE APPLICATION: )
	$(info "make run-cpu" to run the application on the host CPU)
	$(info )

run-cpu: app-CPU
	./app-CPU

clean:
	rm -f $(CPUOBJECTS) $(MICOBJECTS) $(TARGET) *.optrpt
```

Question: You should detect load imbalance if your host has more than 4 threads.

1.2 Load balancing with the parallelized loops

As you could see in subsection 1.1, the problem is that there are not enough iterations in the loop parallelized with "**#pragma omp parallel for**". Let's attempt to improve on this situation by parallelizing the inner loop, which has millions of iterations. Do not forget to resolve race conditions using reduction when you do that.

Question: Benchmark the performance of this implementation and compare to the baseline.

1.3 Parallelizing collapsed loops

Let's try to improve on the solution in subsection 1.2 by parallelizing both the **i-** and the **j-loop**. You can do this using the clause "*collapse(2)*" in "**#pragma omp parallel for**". To resolve race conditions, you may have to introduce a thread-private container that contains partial sums for the resulting array "**s**".

Question: Benchmark the application. What happens?

1.4 Improve the parallelism by strip-mine loops

Let's tweak the implementation from subsection 1.3. The compiler is unable to vectorize the **j-loop** when it is collapsed with the **i-loop**. To help the compiler, strip-mine the **j-loop**. Vectorization will be retained in the innermost loop. The outer two loops will be collapsed. You may have to do some experimentation to find the optimal value of the strip size.

Question: Benchmark the application. What happens?

2 Thread affinity

Thread affinity allows software threads (e.g., OpenMP threads) to execute within the scope of specific processing resources. For example, when running two MPI tasks on a 2-socket node, one may want to bind all threads from task 0 to the first socket and the threads from task 1 to the second socket. It is also possible to bind threads at finer granularities such as cores (instead of sockets) or even hardware threads.

This example demonstrates how to tune thread affinity for a compute-bound application. This problem will be practiced with two 2 source codes: Matrix Multiplication and Discrete Fast Fourier Transform.

2.1 Matrix multiplication with thread affinity

We use as an example the MKL implementation of the DGEMM function to get the performance through tuning thread affinity.

```
/* dgemm.cc  */
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <mkl.h>
#include <omp.h>

int main(int argc, char* argv[]) {

     /* Get arguments from running command: matrix size */
     long n;
     if (argc>1)
     n = atoi(argv[1]);
     else
     n = 8000;

    /* Allocate memory for each matrix */
     double* A=(double*)_mm_malloc(n*n*sizeof(double), 64);
     double* B=(double*)_mm_malloc(n*n*sizeof(double), 64);
     double* C=(double*)_mm_malloc(n*n*sizeof(double), 64);

     /* Variables for benchmarking */
     const double HztoPerf = 1e-9*double(2*n*n*n);
     const int nTrials=10;
     const int skipTrials=2;
     double rate=0, dRate=0;

     printf("\n\033[1mBenchmarking DGEMM.\033[0m\n");
```

```
    printf("Problem size: %dx%d (%.3f GB)\n",
30        n, n, double(3L*n*n*sizeof(double))*1e-9);
    printf("    Platform: %s\n", "CPU");
    printf("     Threads: %d\n", omp_get_max_threads());
    printf("    Affinity: %s\n\n", getenv("KMP_AFFINITY"));

35  /* Initialize data for each matrix */
    #pragma omp parallel for
    for (int i = 0; i < n*n; i++) {
        A[i] =  (double)i;
        B[i] = -(double)i;
40      C[i] = 0.0;
    }

    printf("\033[1m%5s %10s %15s\033[0m\n", "Trial", "Time, s", "Perf, GFLOP/s");

45  for (int trial = 1; trial <= nTrials; trial++) {
        const double tStart = omp_get_wtime();
        cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
            n, n, n, 1.0, A, n, B, n, 0.0, C, n);
        const double tEnd = omp_get_wtime();
50
        if ( trial > skipTrials) { // First two iterations are slow on Xeon Phi; exclude them
            rate  += HztoPerf/(tEnd - tStart);
            dRate += HztoPerf*HztoPerf/((tEnd - tStart)*(tEnd-tStart));
        }
55
        printf("%5d %10.3e %15.2f %s\n",
            trial, (tEnd-tStart), HztoPerf/(tEnd-tStart), (trial<=skipTrials?"*":""));
        fflush(stdout);
    }
60
    rate/=(double)(nTrials-skipTrials);
    dRate=sqrt(dRate/(double)(nTrials-skipTrials)-rate*rate);
    printf("-----------------------------------------------------\n");
    printf("\033[1m%s %4s \033[42m%10.2f +- %.2f GFLOP/s\033[0m\n",
65      "Average performance:", "", rate, dRate);
    printf("-----------------------------------------------------\n");
    printf("* - warm-up, not included in average\n\n");

    _mm_free(A);
70  _mm_free(B);
    _mm_free(C);

    return 0;
}
```

```
/* Makefile */
# MODIFY THESE PARAMETERS TO OPTIMIZE THE CALCULATION:
KMP_AFFINITY=scatter
OMP_NUM_THREADS=1
5 # END OF PARAMETERS TO MODIFY
```

```
CXX = icpc
CXXFLAGS=-qopenmp -mkl
CPUFLAGS = $(CXXFLAGS) -xhost
OPTFLAGS = -qopt-report -qopt-report-file=$@.optrpt

CPUOBJECTS = dgemm.o

TARGET=app-CPU
CXX=icpc

.SUFFIXES: .o .cc

all: $(TARGET) instructions

%-CPU: $(CPUOBJECTS)
	$(info )
	$(info Linking the CPU executable:)
	$(CXX) $(CPUFLAGS) -o $@ $(CPUOBJECTS)

.cc.o:
	$(info )
	$(info Compiling a CPU object file:)
	$(CXX) -c $(CPUFLAGS) $(OPTFLAGS) -o "$@" "$<"

instructions: run-cpu
	$(info )

run-cpu: app-CPU
	$(info )
	$(info TO EXECUTE THIS APPLICATION ON CPU, USE THE FOLLOWING COMMAND: )
	$(info OMP_NUM_THREADS=... KMP_AFFINITY=... ./app-CPU)
	$(info )

clean:
	rm -f $(CPUOBJECTS) $(TARGET) *.optrpt
```

2.2 Discrete Fast Fourier Transform with thread affinity

We implement an application with limited thread scalability, using as an example the MKL implementation of 1D DFFT (discrete fast Fourier transform). Run "make" to compile the code. At the end of the compilation, "make" prints out instructions for running the application on the host and on the coprocessor. Copy and paste the commands listed there.

Prior to running the exercise, you must edit the script run-multiple-processes.sh and change the number of processes, threads per process and the expression for the affinity of each process.

```
/*  dfft.cc */
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <mkl.h>
#include <mkl_dfti.h>
#include <omp.h>
```

```
int main(int argc, char* argv[]) {
     /*  */
     long n;
     if (argc>1)
          n = atoi(argv[1]);
     else
          n = 1<<26;

     double* X=(double*)_mm_malloc(n*sizeof(double), 64);

     DFTI_DESCRIPTOR_HANDLE fftHandle;
     MKL_LONG size = n;
     DftiCreateDescriptor (&fftHandle, DFTI_SINGLE, DFTI_REAL, 1, size);
     DftiCommitDescriptor (fftHandle);

     const double HztoPerf = 1e-9*double(2.5*n*log2(double(n)));
     const int nTrials=100;
     const int skipTrials=3;
     double rate=0, dRate=0;

     printf("\n\033[1mBenchmarking DFFT.\033[0m\n");
     printf("Problem size: %d (%.3f GB)\n",
          n, double(n*sizeof(double))*1e-9);
     printf("    Platform: %s\n", "CPU");
     printf("     Threads: %d\n", omp_get_max_threads());
     printf("    Affinity: %s\n\n", getenv("KMP_AFFINITY"));

     // Initializing data
     #pragma omp parallel for
     for (int i = 0; i < n*n; i++){
          X[i] = 0.0;
     }

     printf("\033[1m%5s %10s %15s\033[0m\n", "Trial", "Time, s", "Perf, GFLOP/s");

     for (int trial = 1; trial <= nTrials; trial++) {
          const double tStart = omp_get_wtime();
          DftiComputeForward(fftHandle, X);
          const double tEnd = omp_get_wtime();

          if ( trial > skipTrials) { // First two iterations are slow on Xeon Phi; exclude them
               rate  += HztoPerf/(tEnd - tStart);
               dRate += HztoPerf*HztoPerf/((tEnd - tStart)*(tEnd-tStart));
          }

          if ((trial==1) || (trial%10==0))
               printf("%5d %10.3e %15.2f %s\n",
                    trial, (tEnd-tStart), HztoPerf/(tEnd-tStart), (trial<=skipTrials?"*":""));
               fflush(stdout);
          }

     rate/=(double)(nTrials-skipTrials);
```

```
    dRate=sqrt(dRate/(double)(nTrials-skipTrials)-rate*rate);
    printf("---------------------------------------------------\n");
    printf("\033[1m%s %4s \033[42m%10.2f +- %.2f GFLOP/s\033[0m\n",
     "Average performance:", "", rate, dRate);
    printf("---------------------------------------------------\n");
    printf("* - warm-up, not included in average\n\n");


    DftiFreeDescriptor(&fftHandle);
    _mm_free(X);

}
```

```
/* Makefile */
# MODIFY THESE PARAMETERS TO OPTIMIZE THE CALCULATION:
KMP_AFFINITY=scatter
OMP_NUM_THREADS=1
# END OF PARAMETERS TO MODIFY

CXX = icpc
CXXFLAGS=-qopenmp -mkl
CPUFLAGS = $(CXXFLAGS) -xhost
OPTFLAGS = -qopt-report -qopt-report-file=$@.optrpt

CPUOBJECTS = dfft.o

TARGET=app-CPU
CXX=icpc

.SUFFIXES: .o .cc

all: $(TARGET) instructions

%-CPU: $(CPUOBJECTS)
    $(info )
    $(info Linking the CPU executable:)
    $(CXX) $(CPUFLAGS) -o $@ $(CPUOBJECTS)

.cc.o:
    $(info )
    $(info Compiling a CPU object file:)
    $(CXX) -c $(CPUFLAGS) $(OPTFLAGS) -o "$@" "$<"

instructions: run-cpu
    $(info )

run-cpu: app-CPU
    $(info )
    $(info YOU MUST EDIT run-multiple-processes.sh TO OPTIMIZE THREAD AFFFINITY )
    $(info TO EXECUTE THIS APPLICATION ON CPU, USE THE FOLLOWING COMMAND: )
    $(info ./run-multiple-processes.sh )
    $(info )

```

```
clean:
     rm -f $(CPUOBJECTS) $(MICOBJECTS) $(TARGET) *.optrpt log-process*.txt
```

```
/* run-multiple-processes.sh */
#!/bin/bash

## TO DO THE EXERCISE YOU NEED TO CHANGE THE VALUES AND EXPRESSIONS IN FUNCTION tune_affinity()

function tune_affinity() {

    # CHANGE THIS VALUE to vary how many concurrent processes you will run:
    export NUMBER_OF_PROCESSES=4

    # CHANGE THIS VALUE to very how many threads per process you wish to use
    export OMP_NUM_THREADS=6

    # CHANGE THIS EXPRESSION to bind threads for process number i to respective cores
    # Hint: you may use the running counter OCCUPIED_THREADS in that expression
    export KMP_AFFINITY=none

}

...

# This is the main script
setup_run
run_benchmark
output_results
```

3 Exercises

4 Submission