PPCES 2009 Lab Exercises

Introduction

We have prepared a set of lab exercises for you. Purpose is to make you feel comfortable with the Sun automatic parallelization feature of the compilers, as well as to write your first OpenMP program. There is also an optional and more advanced lab to further improve the performance of the program, but this is most suitable for the more experienced users.

The lab exercises are organized as follows:

Lab 1 - Automatic shared memory parallelization
Lab 2 - Explicit shared memory parallelization with OpenMP (beginners)
Lab 3 - Tuning the OpenMP lab program (experienced users)

This lab is available in Fortran and C. The exercises are the same for both programming languages.

Please read this introduction part before you start with the actual exercises. The introduction contains information that will be helpful later on

Warning

There will be no handout with the solutions to these labs. Make sure you master the exercises presented next. Please do not hesitate to consult the instructor in case of questions or problems

Installation of the lab exercises

Go to your home directory (% cd) and install the labs through the `setup_lab.sh` script.

When you invoke the script, you need to specify whether you want the UltraSPARC (keyword: sparc) or Intel/AMD (keyword: intel) version of the labs. You also need to specify the Operating System you'll be using (keyword: solaris or linux):

```
% cd
% /var/tmp/setup_lab.sh -s [sparc|intel] -o [solaris|intel]
```

When in doubt what to select, please ask the instructor.

You should have a "sats_labs" directory in your home directory now. Go this directory. Use the Unix “ls” command to see an overview of the directory contents[1]:

```
% ls
lab.c/   lab.fortran/  sats_lib/
```

Now select the directory corresponding to the programming language of choice (C or Fortran) and use the “ls” command to verify the directory contents:

C programmers:

```
% cd sats_labs
% cd lab.c
% ls

Makefile            interface.h       mxv_row.c         plot_par.gnu
check_results.c     labs.h            mxv_row_omp.c     plot_parft.gnu
get_num_threads.c   main.c            plot_omp.gnu
init_data.c         mxv_col.c         plot_ompft.gnu
init_data_omp.c     mxv_col_omp.c     plot_ompth.gnu
```

Fortran programmers:

```
% cd sats_labs
% cd lab.fortran
% ls

Makefile             interface.h      mxv_row.f95       plot_par.gnu
check_results.f95    labs.f95         mxv_row_omp.f95   plot_parft.gnu
get_num_threads.f95  main.f95         plot_omp.gnu
init_data.f95        mxv_col.f95      plot_ompft.gnu
init_data_omp.f95    mxv_col_omp.f95  plot_ompth.gnu
```

If you see the above, congratulations! You have successfully installed the lab exercises. In case you don't see the above, please ask the instructor for help.

Now continue to read until you get to the first assignment. Follow the guidelines you will find there to get started with the first lab.

1 *Directory “sats_lib” is a support directory. It has the libraries needed to build and run the lab programs.*

Executing the lab programs

Although not required, we strongly recommend to use the Unix "tee" command when running the various lab programs. This command is very convenient, as the output of the program is displayed on the screen, as well as stored in the file specified.

For example: `% make run_par | tee par.res`

In this example, the output from the "make run_par" command is displayed on the screen, but also stored in file par.res for future reference.

About the lab exercise

Throughout the lab assignments we work on matrix vector multiplication, computing a = B*c. B is an mxn matrix, c a vector of length n. The result vector a has length m.

This is a popular and simple mathematical operation. A Fortran and C version are available.

We use 2 different implementations. One with poor memory access, as well as one with good memory access.

About the lab program

The main (driver) program uses command line options to specify the matrix sizes "m" and "n". Optionally, one can also specify a threshold value for the OpenMP row and/or column version through the -r and -c options respectively. These options are only relevant for the OpenMP part of this lab and will be discussed later. By default, they are set to 1.

The -h option gives a simple usage overview:

```
% ./mxv_par.exe -h
 Usage:./mxv_par.exe -m <m> -n <n> [-r <r>] [-c <c>] [-f <f>] [-h]

 Options supported:
   <m> number of rows   in the matrix
   <n> number of column in the matrix
   <r> threshold for OpenMP row    version (optional)
   <c> threshold for OpenMP column version (optional)
   <f> fixed repeat count (optional)
   -h print this message and exit
```

To increase the accuracy of the timings, each implementation is executed multiple times. The average execution time is then used to compute the performance in Million floating-point operations per second (Mflop/s).

The driver program estimates this so-called repeat count, such that each of the three implementations executes in approximately the same, prefixed, time.

In case the Mflop/s number is higher for a specific version, it means that it performs more floating-point operations in the same amount of time. Given it solves the same problem, it means the algorithm with a higher Mflop/s number is more efficient.

Although not recommended, the user can have the driver program skip the estimate of the repeat count and specify an explicit value through the -f option. This has to be used when using the Sun Performance Analyzer[2]. Otherwise it will be hard to interpret the data.

The typical program output looks like this (it is assumed one thread is used)::

```
% ./mxv_par.exe -m 1000 -n 1000
-----------------------------------------------------------------------
   M    N   Mem(KB)   Threads   Thresholds    Performance (Mflop/s)
                                 Row   Col       Row      Col
-----------------------------------------------------------------------
1000 1000   7828.12         1      1     1     787.0    788.5
```

The first two columns are the matrix sizes. The third column gives the estimated memory usage in KByte. The fourth column shows the number of threads used. The thresholds are the cut-off values in the OpenMP version. If the matrix size is less than this value, the implementation the threshold applies to (e.g. the row or column version) will not be executed in parallel. This is discussed in detail in the talk and also covered in one of the lab later on. The last two columns show the performances in Mflop/s.

The commands in the make file run the program for a sequence of problem sizes.

The driver program also produces a plot file (called “perf.dat”). This file is used by the GNUplot scripts to plot the performance as a function of the memory footprint.

2 *There is no formal lab for this, but of course you can use the lab programs to experiment with this tool*

Description of the make file

The make file is called "Makefile" and it supports several commands. We strongly encourage you to use this make file to build, run and graphically display the results.

The make file has been set up such that by simply typing "make" at the command line, you will get an overview of all commands supported.

```
% make
Program targets are:
   par     - automatically parallelized by the compiler
   omp     - explicitly parallelized with OpenMP
   ompth   - explicitly parallelized with OpenMP - use thresholds
   parft   - automatically parallelized version optimized for first
             touch
   ompft   - OpenMP version optimized for first touch

Note: we will refer to these targets as <key> below

make <key> - command to build specific a version of the program
  example: % make par - build the automatically parallelized version

make run_<key> - command to run a specific version
  example: % make run_par - run the automatically parallelized version

make plot_<key> - command to plot the results for a specific version
  example: % make plot_par - plot performance of the automatically
             parallelized version

Global commands (apply to all versions):
make build - build all versions
make run   - run   all versions
make plot  - plot  all performance results

Various other commands supported:
make                    - Equivalent to make help
make help               - On-line info
make check              - Check release numbers of compilers and tools
make clean              - Remove all objects and executables
make veryclean          - ARE YOU SURE ? Will remove all objects,
                          executables, listings, plus all .dat files
```

The make file is already prepared for all of the lab exercises. You will start using all of the functionality in a gradual way and it will be explained as we proceed with the exercises.

We recommend you use the built-in make commands to run the various programs.[3]

Warning:

When working on a specific lab assignment, do not use commands that have not been discussed yet. Most likely they will fail.

3 *Of course one can always run a program manually, but then the plot commands in the make file are most likely going to fail.*

The make file has two variables that you can (and should) set by editing the make file as we proceed with the labs:

```
PAR  =                          #-- Auto-parallelization options
OMP  =                          #-- OpenMP parallelization options
```

In the various lab assignments you will get directions what values these variables should be set to. This is rather straightforward, but please follow the guide lines

From the settings in the make file, the compile and link options are derived. All you need to do, is to follow the guidelines in this handout and set the make variables to the values indicated. The make commands will be adapted automatically.

General structure of the lab assignments

Each assignment will usually begin with a short introduction, followed by a specific assignment.

We will also give you the commands to build the code, run the job and to plot the performance results. The typical sequence is as follows:

```
% make <key>         - build the program
% make run_<key>     - run the program
% make plot_<key>    - graphically display the results
```

With <key> we denote a program target supported by the make file e.g. `make par` ("par" is the <key> here, or `make omp`.

Note: The make file automates things for you, but it has been put in verbose mode, so you will see all commands as they are executed

Getting started

Before you proceed, check that your search path is correct. The make file includes a simple command to check for the versions of the Fortran and C compilers, plus the collect and analyzer commands.

For educational purposes the commands to obtain this information (e.g. "f95 -V") are shown in the output too.

```
% make check
f95 -V
f95: Sun Ceres Fortran 95 8.3 Linux_i386 2008/10/22
Usage: f95 [ options ] files.  Use 'f95 -flags' for details
cc -V
cc: Sun Ceres C 5.10 Linux_i386 2008/10/22
usage: cc [ options] files.  Use 'cc -flags' for details
collect -V
collect: Sun Ceres Analyzer 7.7 Linux_i386 2008/10/22
analyzer -V
analyzer: Sun Ceres Analyzer 7.7 Linux_i386 2008/10/22
```

Note that the version numbers might be different on your system

Lab 1 - Automatic Parallelization

We use the compiler to automatically parallelize the row and column implementation of the matrix times vector algorithm.

Use the PAR variable in the make file to activate automatic parallelization.

Please set it to:

```
PAR = -xautopar -xloopinfo
```

Build the program using the make file:

```
% make par
```

Verify that the compiler did the right thing by checking the parallelization messages displayed on your screen.

Use the make file to run the program. The program will now be executed on 1 and 2 threads (you may want to study the make file how this is done):

```
% make run_par | tee par.res
```

Use the make file to display the performance graphically:

```
% make plot_par
```


Lab 2 - Explicit parallelization with OpenMP

In this lab we use OpenMP directives to explicitly parallelize the row and column implementations of the algorithm. As we have seen in the previous lab, the compiler will do this for us, but it is useful to try it yourself on a relatively simple case.

The make file has a variable called OMP that has to be used to activate the OpenMP directives you are going to put in the source.

Please modify the make file and set variable OMP to:

```
OMP = -xopenmp -xvpara -xloopinfo      (for C)

OMP = -xopenmp -vpara -xloopinfo       (for Fortran)
```

With the -xopenmp option, the compiler only parallelizes the explicitly parallelized sources that have the OpenMP directives inserted.

The -xloopinfo option causes a short summary message on the parallelization to be printed on the screen.

With the -xvpara/-vpara option the OpenMP compiler tries to detect possible logic errors in the parallelization.

After you have parallelized the program by using OpenMP directives (see below for help how to do this), you will use the commands as indicated below to compile and run the program.

Assignment 2.1 - Explicit parallelization of the row version

We start with the row version. It is straightforward to parallelize this operation. All dotproducts can be accumulated in parallel. Verify that this translates to parallelizing the outer loop over 'i'.

Warning: You need to edit the source files in order to implement the parallelization with OpenMP.

C programmers:

The relevant source fragment of `mxv_row_omp.c` might look like this:

```
#ifdef _OPENMP
#include <omp.h>
#endif
    .......
#pragma omp parallel for default(none) \
        private(...) shared(...)
   for (i=0; i<m; i++)
   {
     a[i] = 0.0;
     for (j=0; j<n; j++)
       a[i] += b[i*n+j]*c[j];
   }
```

You need to determine which of the variables are shared and which ones are private.

Fortran programmers:

The relevant source fragment of `mxv_row_omp.f95` might look like this:

```
!$    use omp_lib
    .......
!$omp parallel do default(none) &
!$omp private(...) shared(...)
      Do i = 1, m
         a(i) = 0.0
         Do j = 1, n
            a(i) = a(i) + b(i,j)*c(j)
         End Do
      End Do
!$omp end parallel do
```

You need to determine which of the variables are shared and which ones are private.

Compile and link the program:

```
% make omp
```

Check the messages to verify that the compiler parallelized the row version. If you don't see parallelization messages, you may have made a mistake in the OpenMP directive, resulting in the compiler ignoring it.

The column version has not been parallelized yet, so you should not see any message for this version.

Once the `make omp` command succeeds, you can run the parallelized program:

```
% make run_omp
```

If you get incorrect results, it is most likely that you made a mistake with the data sharing attributes (private/shared) of variables

In case the program runs to completion, you can graphically display the performance results:

```
% make plot_omp
```

Note that the column version has not been parallelized yet. This is what we will do in the next assignment.

Assignment 2.2 - Explicit parallelization of the column version

Verify that the nested loop in the column version can not be parallelized at the outer loop level, because it would introduce a race condition on array a. Multiple threads would be updating this array simultaneously if the j-loop was to be parallelized.[4]

We will now work on a parallel version of the column oriented implementation. Below are program fragments of the implementation.

C programmers:

The relevant source fragment of `mxv_col_omp.c` might look like this:

```
#ifdef _OPENMP
#include <omp.h>
#endif

#pragma omp parallel default(none) \
        private(...) shared(...)
{
#pragma omp for
   for (i=0; i<m; i++)
       a[i] = b[i*n]*c[0];

   for (j=1; j<n; j++)
   {
#pragma omp for
      for (i=0; i<m; i++)
        a[i] += b[i*n+j]*c[j];
   }

} /*-- End of parallel region --*/
```

4 *It is not optimal to parallelize at the inner loop level and it is possible to parallelize this operation at the outer loop level. This is beyond the scope of this lab however.*

Fortran programmers:

We can use the OpenMP `workshare` directive to parallelize the vector operations. The relevant source fragment of `mxv_col_omp.f95` might look like this:

```
!$    use omp_lib
         .....
!$omp parallel default(none) &
!$omp private(...) shared(...)

!$omp workshare
      a(1:m) = b(1:m,1)*c(1)
!$omp end workshare

      Do j = 2, n
!$omp workshare
         a(1:m) = a(1:m) + b(1:m,j)*c(j)
!$omp end workshare
      End Do

!$omp end parallel
```

Compile the program after you have put in the OpenMP directives:

```
% make omp
```

Check the messages to verify that the compiler parallelized the column version. If you don't see parallelization messages, you may have made a mistake in the OpenMP directive, resulting in the compiler ignoring it.

Once the `make omp` command succeeds, you can run the parallelized program:

```
% make run_omp
```

If you get incorrect results, it is most likely that you made a mistake with the data sharing attributes (private/shared) of variables

In case the program runs to completion, you can graphically display the performance results:

```
% make plot_omp
```

You should now see that both versions execute in parallel, be it that the column version does not perform very well.

Lab 3 - OpenMP performance tuning (experienced users)

Assignment 3.1 - Use the OpenMP if-clause to fine tune performance

As explained in the seminar, the OpenMP `if-clause` can be used to control when the parallel region should be executed by multiple threads, or by a single thread only. This mechanism is used in this next assignment to fine tune the performance.

In the main program there are two variables `threshold_row` and `threshold_col` that can be used for this purpose. The main program assigns a value to them through the optional `-r` and `-c` options on the command line. By default they are set to 1.

The values of these variables are passed in to the row and column versions.

> *What you need to do is to include the `if-clause` in the OpenMP source versions of these algorithms. In other words, modify the source of `mxv_row_omp.c` and `mxv_col_omp.c` and/or `mxv_row_omp.f95` and `mxv_col_omp.f95`.*

C programmers:

Your source will have to look something like this for the row version:

```
#pragma omp parallel for if (m > threshold_row) default(none) \
        private(...) shared(...)
```

For the column version, you could use the following:

```
#pragma omp parallel if (m > threshold_col) default(none) \
        private(...) shared(...)
{ ... }
```

Fortran programmers:

Your source will have to look something like this for the row version:

```
!$omp parallel do if (m > threshold_row) default(none) &
!$omp private(...) shared(...)
```

For the column version, you could use the following:

```
!$omp parallel if (m > threshold_col) default(none) &
!$omp private(...) shared(...)
      ....
!$omp end parallel
```

Make sure these new versions compile without problems:

```
% make omp
```

Once you have rebuilt the OpenMP version of these two versions you can conduct additional performance experiments.

The make file has already been prepared to assist with this:

> *Check the make file for variables THRESHOLD_ROW and THRESHOLD_COL. Both are currently set to 1. Change these numbers to a more appropriate value.*

Tip: The initial curves from Lab 2 use a threshold of 1. Look at the performance charts to give you an indication what the threshold values should be. You can literally see at which memory footprint the performance on 2 threads exceeds the performance on 1 thread. This footprint corresponds to a specific size of the matrix. That size is your threshold value.

After you have changed these values in the make file, execute the following command:

```
% make run_ompth
```

> *Note that the threshold values specified in the make file should appear in the output shown on the screen.*

Display the results through this command:

```
% make plot_ompth
```

You should now no longer see a performance degradation when using two threads for (too) small matrix sizes.

Assignment 3.2 - Optimize OpenMP performance for First Touch placement

Warning: This assignment is only meaningful if you use a cc-NUMA system for the labs. When in doubt, please consult the instructor.

The current Autopar and OpenMP versions do not exploit First Touch (FT) data placement. By default, only one node initializes all the data. On a cc-NUMA architecture this can inhibit parallel scalability. In this lab we show how to optimize the program for First Touch.

To this end, parallelize the data placement by putting in the right OpenMP directives in function `init_data_omp.c` (in C) or `init_data_omp.f95` (in Fortran).

You need to make sure to parallelize the initialization such that the thread accessing a portion of the data in the matrix times vector implementation, has most, if not all, of that data in its local memory. For this it is important to follow the storage order, so focus on the row version in C and the column version in Fortran.

After you have put in the OpenMP directives in the initialization function, compile and make sure the loops are parallelized.

Now run the program and display the performance. These are the commands to use for this:

For the automatically parallelized version:

```
% make parft
% make run_parft
```

Display the results through this command:

```
% make plot_parft
```

For the OpenMP version:

```
% make ompft
% make run_ompft
```

Display the results through this command:

```
% make plot_ompft
```

On a cc-NUMA system you should see a performance difference when using two threads. Note that the matrix times vector routines have not changed! We link in the object files you generated in the preceding labs. All that is different is the way the data is initialized.

-- END OF THE PPCES 2009 LAB --

