# OMPlab on Sun - Lab Programs

**Introduction**

We have prepared a set of lab programs for you.

We have an exercise to implement OpenMP in a simple algorithm and also various programs to demonstrate certain features of the Sun OpenMP implementation.

The following lab programs are available:

| | | |
|---|---|---|
| mxv.c | - | demonstrates automatic parallelization in C; can also be parallelized with OpenMP |
| mxv.fortran | - | demonstrates automatic parallelization in Fortran; can also be parallelized with OpenMP |
| sun-mp-warn | - | demonstrates the use and run-time behavior of SUNW_MP_WARN |
| autoscoping | - | demonstrates the use of autoscoping in Fortran |
| strassen | - | implements nested parallelism and includes a performance analyzer demo |

Below you will find a description of each lab program.

All directories have a Makefile. If you just type “make” you will get an overview of what commands are supported. There will not be an extensive description of these commands. Most of them are hopefully straightforward. Please ask for help if you need clarification.

**Installation of the OMPlab programs**

Go to your home directory (% cd) and unpack the distribution directory by using the unzip command:

```
% cd
% unzip /tmp/OMPlab_on_Sun.zip
```

You will now have a directory called “OMPlab_on_Sun”. In this directory you will find the various lab programs. Each lab will be in a specific directory. If you go into this directory you will find the Makefile, sources and other files that might be needed to do the lab.

*There will be no hand out with the solutions to these labs. Do not hesitate to consult the Sun people in case of questions or problems.*

 

## Lab: mxv.c and mxv.fortran

This program implements the well-known matrix vector multiplication algorithm. This is a popular and simple mathematical operation. A Fortran and C version are available. The behavior and workings are largely identical and therefore we have combined the description of these two labs.

The program compares the performance on 3 different implementations of this algorithm. One with poor memory access, one with good memory access and a tuned library version from the Sun Performance Library.
We refer to Appendix A for more details on the workings of this lab program.

**Getting started**

Make sure you're search path is set up correctly. Type this in to make sure you will be using the correct versions of the compilers and tools:

```
% make check
```

This command will also check whether you have the “gnuplot' command in your search path

**Assignment 1 - Automatic parallelization**

We will first compile and run the program using the automatic parallelization feature on the Sun compilers. To this end, build and run the program using the make file and standard options supplied:

```
% make apar
% make run_apar
```

After the program has finished, plot the results:

```
% make plot_apar
```

The performances for the row, column and library version are plotted for 1, 2 and 4 threads.

**Assignment 2 - OpenMP parallelization of the row version**

Use OpenMP directives to parallelize the row version (mxv_row.c in C, mxv_row.f95 in Fortran).

Build and run the program:

```
% make omp
% make run_omp
```

After the program has finished, you can plot the results:

```
% make plot_omp
```

| Note that the performance of the column version is the same for all threads.<br>This is because you have not yet parallelized it! |
|---|

 

**Assignment 3 - OpenMP parallelization of the column version**

Verify that the the nested loop in this algorithm can not be parallelized over the outermost loop. This will give rise to a data race condition. Instead, parallelize the initialization loop and the innermost loop in this algorithm.

Build and run the program:

```
% make omp
% make run_omp
```

After the program has finished, you can plot the results:

```
% make plot_omp
```

| Do not worry if the program runs for a long time. The column version has not been parallelized very efficiently! |
|---|

**Assignment 4 - Tuning our OpenMP versions**

The graphs will clearly show for what problem sizes the algorithm should not be executed in parallel. We can use the if-clause in OpenMP to avoid executing in parallel if the matrix is too small.

Include the OpenMP if-clause in both the row and column version. For this you can make use of two program variables called “threshold_row” and “threshold_col” respectively.

Your OpenMP version could use these variables as follows:

```
#pragma omp parallel if (m > threshold_row) .....      (mxv_row.c)
#pragma omp parallel if (m > threshold_col) .....      (mxv_col.c)

!$omp parallel if (m > threshold_row) .....            (mxv_row.f95)
!$omp parallel if (m > threshold_col) .....            (mxv_col.f95)
```

The two threshold variables threshold_row and threshold_col are read in from input file INPUT and accessible in the sources of the row and column versions. You can find and set the values on the second row of file INPUT. Currently they have been set to 1. This implies these routines will always run in parallel. Clearly that is not a good idea.

Experiment with higher threshold values to find the optimal cross over point for both algorithms.

**Assignment 5 (advanced) - Tuning the column OpenMP version**

This is a challenge. The column version can be implemented at the outer loop level, but you need to work for it.

Recall that this version takes linear combinations of the columns of the matrix. Well, in that case each thread can calculate partial linear combination and then these partial results can be combined. Sounds like a reduction operation, doesn't it?

 

## Lab: sun-mp-warn

This is a demo program to show you how the SUNW_MP_WARN environment variable can help diagnosing run-time problems (e.g. violations of the OpenMP standard).

**Assignment 1 - Getting started**

Study the Makefile. You will see a make variable called THREADS. This variable controls how many threads will be used. The current setting is such that the program will run the first two cases fine, but will hang on the third (when using 2 threads).

Verify this behavior by executing the “make run” command. This will (re)build the program and run the test cases.

**Assignment 2 - Additional experiments**

Set make variable WARN to TRUE and run the experiments again. Note the run-time messages you now get.

 

## Lab: autoscoping

Autoscoping is a convenient feature to relieve the programmer from explicit scoping as much as possible.

If the compiler is not able to determine the scope of a variable, a warning will be issued and the parallel region will be executed on one thread only. In such a case one can explicitly scope those variables not handled by the compiler, thereby typically greatly reducing the number of variables to be scoped.

The following examples are available:

| | | |
|---|---|---|
| make blas | - | compiles a BLAS routine |
| make cfd | - | compiles a kernel derived from a CFD program |
| make critical | - | compiles and runs an example using critical sections |
| make reduction | - | compiles and runs an example using reductions |
| make as-failure | - | compiles and runs an example where autoscoping fails |

**Assignment 1 - Getting started**

Run each of the cases listed above by issuing the appropriate make command (e.g. “make blas”). Try to understand the messages that come out of the compiler.

**Assignment 2 - Assisting autoscoping**

The last test case (“as-failure”) can be repaired by explicitly scoping the variable the compiler can not scope automatically. Try this and compare the difference in run-time behavior.

 

## Lab: strassen

This is a very elaborate example to demonstrate nested parallelism. It is also a good example to use the Sun Performance Analyzer.

Three different versions of matrix multiplication are compared. We start with the regular "classic" matrix multiply, followed by a faster variant called the "Strassen" algorithm. The latter implements a recursive algorithm and has also been parallelized using nested parallelism in OpenMP. This is the third version we run.

Although this is largely meant as a free running exercise, we will also give some guidance in the form of assignments[1].

**Performance information**

Each of the 3 versions will be timed separately and the results will be shown on the screen. The program will also produce (or update) a file with timing information. This file is called "timings_strassen.csv". Per run, one line will be added to this file. If the file does not exist yet, it will be created. Each line contains the relevant parameters, plus the timing information for that run. The fields are tab separated and can easily be loaded into a (StarOffice) spreadsheet for further processing.

**Assignment 1 - Getting started**

Make sure you can build and run the standard test case. The easiest thing to do is to use the Makefile for this.

Type "make run_quick" and watch what happens.

You will see that the program runs on two threads only.

Note the warning that two Sun specific environment variables have not been set yet. This will be done for you in the next assignment and you can experiment with it yourself in Assignment 4.

**Assignment 2 - Exploiting nested parallelism**

We will now run several cases. We will do this for a fixed number of threads (2 in this case), and different nesting levels.

Check the Makefile. This is where to control the jobs to be run. Make variable THREADS can be used to specify the number of threads you would like to run on. With NESTING you control the nesting level. Both can be a list.

Run the standard experiments as specified through the Makefile.

```
% make run
```

You should now have the file "timings_strassen.csv with the timing information.

---

1 *The Makefile has automated several things for you. By changing the values of some make variables you can still conduct many experiments yourself as well. See also Assignment 4.*

 

**Assignment 3 - Using the Sun Performance Analyzer**

The Makefile also has a command to collect performance data on the runs you have done in Assignment 2.

All you need to do is the following:

```
% make run_collect
% make run_analyzer
```

The second command will invoke the Sun Performance Analyzer. It will show you a list with performance experiments to select from.

The first digit in the name denotes the number of threads used, the second digit is the nesting level. For example “strassen.2x3.er” has the performance data for an experiment run on 2 threads with a nesting level of 3.

Select an experiment and use the analyzer to understand the performance behavior. We particularly recommend to take a look at the Timeline tab.

**Assignment 4 - Free running lab**

Now that you're hopefully somewhat familiar with the workings of this program, it is time to explore it yourself.

The program has a built-in help function:

```
% ./strassen.exe -h
Strassen Matrix Multiply - Usage:
Usage: ./strassen.exe -n <dim> -r <repeat> -d <depth> -b <blocking> [-v] [-h]
        -n dimension of square matrices [1024]
        -r repeat count for timing measurements [1]
        -d recursive depth for Strassen algorithm [32]
        -b blocking size for matrix multiply [16]
        -v verbose option [OFF by default, set to -v to activate]
        -h print this message
```

You can change the settings by specifying one or more of the options explicitly. The options not specified will maintain their default values.

Note that the parallel behavior will still be controlled through the following OpenMP environment variables: OMP_NUM_THREADS and OMP_NESTED. In addition to these, you probably also want to experiment with the two Sun-specific environment variables: SUNW_MP_MAX_POOL and SUNW_MP_MAX_NESTED_LEVELS.

| Therefore, it is probably easiest to use the make variables in the Makefile to specify the kind of additional experiments you would like to run. Please check the Makefile on how to do this. |
|---|

 

**Appendix A - Description of the matrix times vector program**

*About the algorithm*

This lab contains an implementation of the matrix times vector operation a=B*c. The program measures performance in Mflop/s for three different implementations.

This algorithm has been chosen because it is both conceptually and from an implementation point of view, quite simple.

A row-oriented and column-oriented version are supplied in source format. In addition to this, we will also measure the performance of routine "dgemv" from the Sun Performance Library. This routine implements various flavors of the matrix times vector operation. We have selected the parameters such that the same product a=B*c will be calculated as implemented in the source versions.

It is recommended to examine the source of the main program for the details how "dgemv" is used. Every routine from the Sun Performance Library has a man page. You can for example look up the information for the function parameters (%man dgemv).

*Lab program input*

The program reads it's input values from standard input, but is easiest to redirect this from a file. Example input can be found in file INPUT. Please use this file when working on the lab exercises.

The structure of file INPUT is as follows:

```
# thresholds (row col)
1 1
#  m     n krep
   5     5 1500000
  10    10  500000
  20    20  200000
     ... etc ...
```

Explanation of the structure:

1. The first and third input line are skipped to allow for comment in the input file.

2. The second line should contain 2 integer numbers. It is recommended to use the OpenMP if-clause to prevent very short loops from running in parallel.

   The 2 integer values (currently set to 1) can be used in the OpenMP if-clause for the row and column versions of the algorithm (first and second value respectively).
   Through this mechanism you can easily perform experiments using different parameter values for the thresholds. This is actually part of these lab exercises: you are asked to find the optimal parameter values.

3. In the remainder of the input (file), three values are expected. The dimensions M and N, plus a repeat count to ensure the test runs sufficiently long.

 