Parallelware Tools ATMUX Quickstart

Parallelware tools provide an innovative solution for the development of C/C++/Fortran parallel code targeting multicore CPUs and GPUs. Its new static code analysis specializing in parallelism helps to accelerate the software run-time by reducing development effort through detection and generation of bug-free parallel code.

What is Parallelware Analyzer?

Parallelware Analyzer is a new static code analyzer specializing in parallelism. It consists of a suite of command-line tools to detect and prevent parallel bugs originated by data races and data movement issues, and get best-practice recommendations to develop faster software. Parallelware Analyzer provides the following command-line tools for the key stages of the parallel development workflow:

- **pwreport**: provides a high-level overview of your code: summary of parallelized regions, defects, recommendations and opportunities..
- **pwcheck**: looks for **defects** such as race-conditions and issues **recommendations** on best-practices and performs **data-race analysis.**
- **pwloops**: provides insight into the **parallel properties of loops** found in the code which constitute opportunities for parallelism.
- **pwdirectives**: provides guided generation of parallel code for multicore CPUs and GPUs, with **OpenMP and OpenACC**, using multithreading, offloading and tasking.

What is Parallelware Trainer?

Based on the same Parallelware technology used by Analyzer, Parallelware Trainer provides a graphical Integrated Development Environment designed to help get started parallelizing code and experimenting with different parallel implementations. Many functionalities provided by Parallelware Analyzer are available through graphical interaction in Trainer.

Quickstart with ATMUX

To get started with the Parallelware tools we will use the ATMUX C example. We will start by using Parallelware Analyzer to successfully analyze the code, follow recommendations and get opportunities for parallelization. Then, we will move to Parallelware Trainer to implement different parallel versions of ATMUX and compare their performance through experimentation.

1. Download Parallelware Analyzer and its license file, then uncompress and move inside the license file with name *pwa.lic*:

```
$ tar xvfz pwanalyzer-0.16.0_linux-x64.tgz
$ mv pwanalyzer-eap.lic pwanalyzer-0.16.0_linux-x64/pwa.lic
```

> If you are using a supercomputer, you may already have Parallelware tools available as a Environment modules (e.g., `module load pwanalyzer`, `module load pwtrainer`).

2. Download and uncompress the ATMUX code.

```
$ tar xvfz ATMUX.tar.gz && cd ATMUX
```

3. Use make to build and run ATMUX:

```
$ make run
```

> The Makefile is designed to run on your local machine. If you are using a supercomputer consider adjusting it accordingly to perform the execution on the corresponding node.

4. Run **pwreport** to get a first overview of the code:

```
$ pwreport atmux.c
0 files successfully analyzed and 1 failure in 34 ms (pass
--show-failures for error details)
...
SUGGESTIONS

  1 file could not be analyzed, get more information by enabling error
reporting:
    pwreport --show-failures atmux.c
```

> Some files fail due to missing includes. You should pay attention to the suggestions outputted by the different tools. In this case you are instructed to use the `--show-failures` flags.

5. Re-run with the `--show-failures` flag:

```
$ pwreport atmux.c --show-failures
...
error: file was not processed correctly: 'atmux.c'
atmux.c:6:10: fatal error: 'CRSMatrix.h' file not found
```

```
#include <CRSMatrix.h>
         ^~~~~~~~~~~~~
...
```

6. You need to add the *lib* directory to the include path. Compiler flags are passed to all Parallelware Analyzer tools after all other arguments and separated by "--", following the GCC/Clang syntax:

```
$ pwreport atmux.c -- -I lib
...
CODE COVERAGE
  Analyzable files:              1 / 1 (100 %)
  Analyzable functions:          0 / 9 (0 %)
  Analyzable loops:              0 / 4 (0 %)
  Parallelized SLOCs:            0 / 117 (0 %)

SUMMARY
  Total defects:                 0
  Total recommendations:         3
  Total opportunities:           0
  Total data races:              0
  Total data-race-free:          0
```

> Notice that the code coverage is quite poor. There are also no opportunities for parallelization. However, there are 3 recommendations available and, in many cases, following recommendations can increase the code coverage which in turn yields more parallelization opportunities.

7. Run **pwcheck** to get information about those recommendations:

```
$ pwcheck atmux.c -- -I lib
...
[PWR011] atmux.c:31:5 outline loop to increase compiler and tooling
code coverage

       31:     for (i = 0; i < n; i++) {

       SUGGESTION: consider extracting the loop to a dedicated
function


...
```

> The PWR011 recommends outlining loops to dedicated functions to improve tooling coverage. Doing this for hotspots should help to detect more parallelization opportunities.

8. Follow the recommendation by outlining the loop at line 31 to a function called *compute* and replace the loop by an invocation to it:

```
$ vim atmux.c
...
void compute(double *val, double *x, double *y, int *col_ind, int
*row_ptr, int n) {
  int i, k;

  // y = A^T x
  for (i = 0; i < n; i++) {
    for (k = row_ptr[i]; k < row_ptr[i + 1]; k++) {
        y[col_ind[k]] = y[col_ind[k]] + x[i] * val[k];
    }
  }
}
...

  compute(val, x, y, col_ind, row_ptr, n);
 }

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

This is a really simple code for which we know that the hotspot corresponds to the loop at line 31. For real applications, profiling will be required to locate hotspots.

9. Run **pwreport** again and notice how following one recommendation has enabled the detection of two new parallelization opportunities in two loops of the code:

```
$ pwreport atmux.c -- -I lib
...
CODE COVERAGE
  Analyzable files:              1 / 1 (100 %)
  Analyzable functions:          1 / 10 (10 %)
  Analyzable loops:              2 / 4 (50 %)
  Parallelized SLOCs:            0 / 122 (0 %)

SUMMARY
  Total defects:                 0
  Total recommendations:         2
  Total opportunities:           2
  Total data races:              0
  Total data-race-free:          0
```

10. Run **pwloops** to get information about those parallelization opportunities:

```
$ pwloops atmux.c -- -I lib
...
Loop                Analyzable Patterns Opportunity Auto-Parallelizable Parallelized
------------------- ---------- -------- ----------- ------------------- ------------
atmux.c:compute:20:3     x        sparse    multi           x
atmux.c:compute:21:5     x        sparse    simd            x
atmux.c:atmux:38:5
atmux.c:main:78:5
...
```

11. You can also use **pwloops** to visualize the source code annotated with opportunities. You can do so filtering by function to narrow the output to the relevant functions:

```
$ pwloops atmux.c --code --function atmux.c:compute -- -I lib
...
Line Opp  atmux.c
---- ---
-----------------------------------------------------------------
  16   void compute(double *val, double *x, double *y, int *col_ind,
int *row_ptr, int n) {
  17      int i, k;
  18
  19      // y = A^T x
  20  P   for (i = 0; i < n; i++) {
  21        for (k = row_ptr[i]; k < row_ptr[i + 1]; k++) {
  22            y[col_ind[k]] = y[col_ind[k]] + x[i] * val[k];
  23        }
  24      }
  25   }

                                                              ...
```

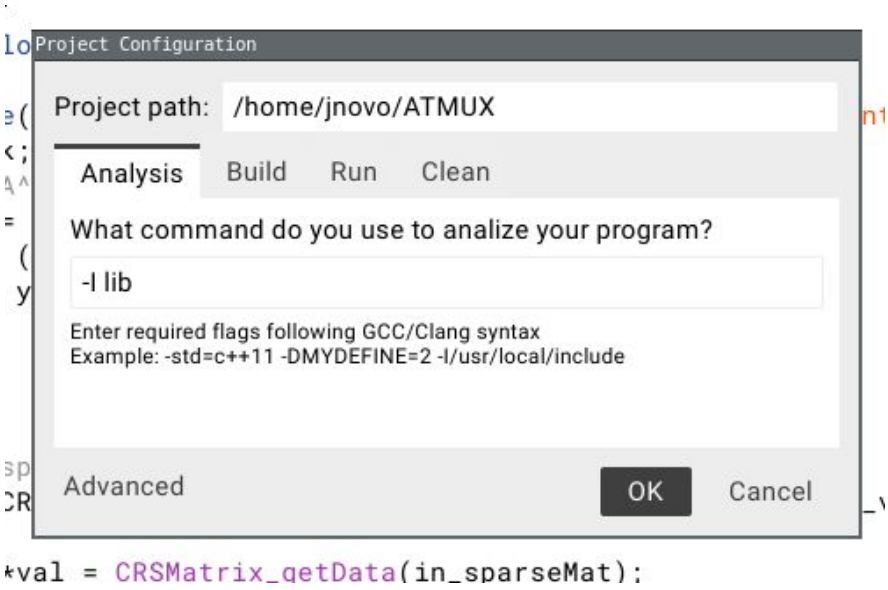
> Filtering by function comes very handy to focus on the hotspots detected through profiling.

12. You can use **pwdirectives** to generate parallel versions of your code. For instance, to parallelize the loop at line 20 of *atmux.c* using defaults settings (OpenMP multithreading):

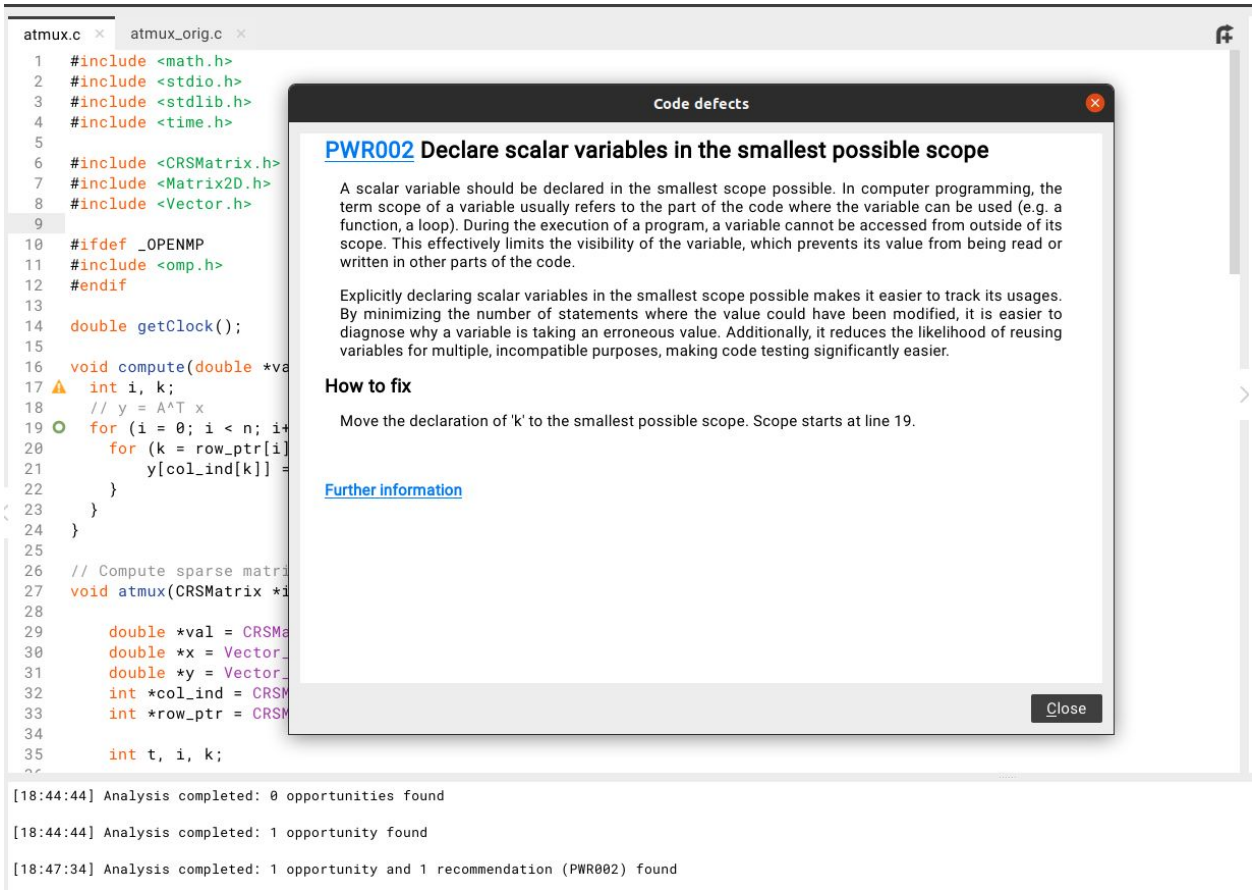
```
$ pwdirectives atmux.c:20 -o atmux_omp.c -- -I lib
Compiler flags: -I lib
Warning: defaulting to OpenMP CPU multithreading paradigm since no
explicit options were provided.
Attempting to parallelize loop at 'atmux.c:20
Parallel sparse reduction pattern identified for variable 'y' with
associative, commutative operator '+'
Available parallelization strategies for variable 'y'
  #1 OpenMP atomic access (* implemented)
  #2 OpenMP explicit privatization
Loop parallelized with multithreading using OpenMP directive 'for'
Complete access range for variables: 'row_ptr', 'col_ind', 'val', 'y'
Parallel region defined by OpenMP directive 'parallel'
Make sure there is no aliasing among arguments in 'compute': val, x,
y, col_ind, row_ptr, n
Successfully created atmux_omp.c
```

13. Note that most of the functionality you've used from Parallelware Analyzer's **pwcheck**, **pwloops** and **pwdirectives** tools is also available in Trainer. We will now switch to **Parallelware Trainer** to generate several parallel versions, building and running them through the graphical user interface in order to quickly compare their performance.

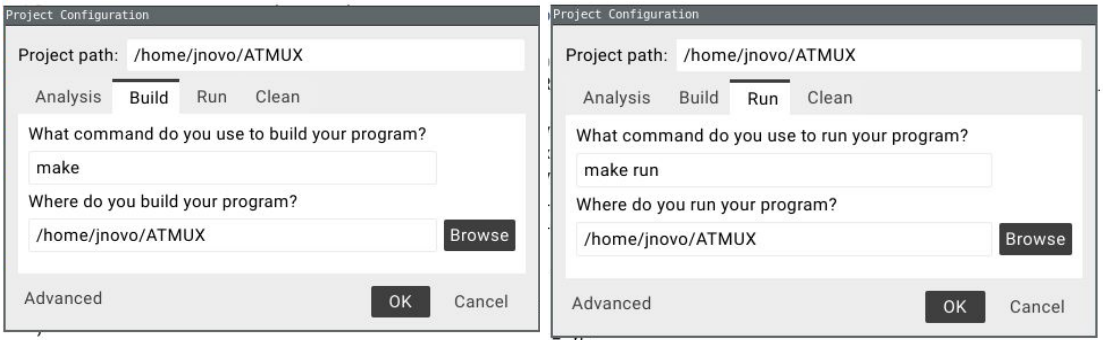
Launch Parallelware Trainer by invoking **pwtrainer**, open the ATMUX directory through the *File > Open Project* menu entry and double click *atmux.c* to open it. Notice the header file error in the Parallelware output console. In Trainer, you can add the compiler flags in the *Analysis* tab of the project configuration, so select the *Project > Configuration* menu entry and add `-I lib`.



14. Now notice a yellow warning icon next to the 17 line number. This is a recommendation just like those outputted by Analyzer's **pwcheck** tool. Click it to get information on what it is about and how to address it. In this case it is advising that you should move the declarations of *i* and *k* to the loop header. Please, do so.



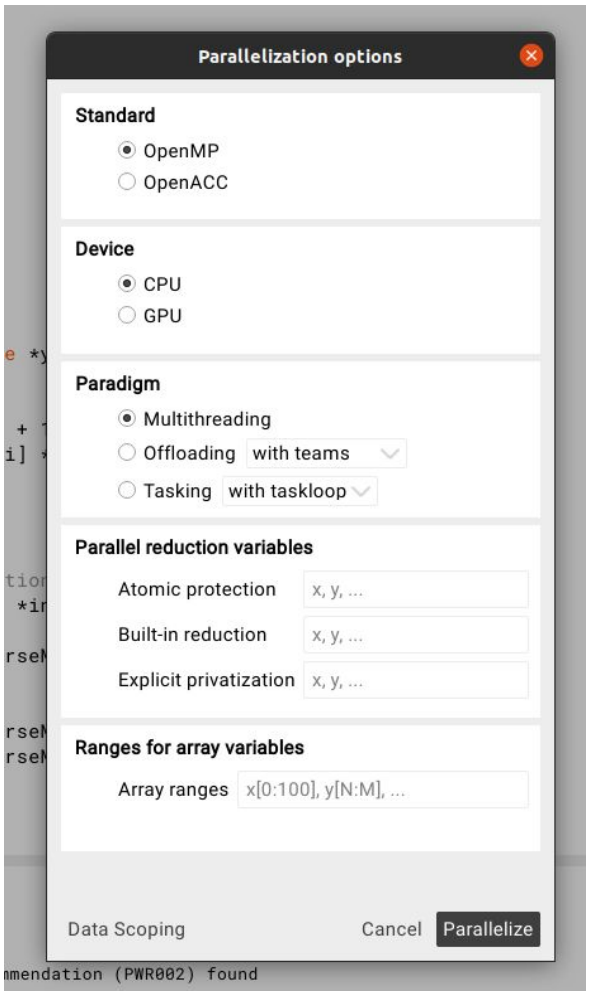
15. Press F6 or select the *Project > Run* menu entry to run the sequential version of the code. If you haven't done so yet, you will need to fill the *Build* and *Run* tabs with *make* and *make run*, respectively. Once you do so, Parallelware Trainer will execute the ATMUX program and you will get the results in the *Execution output* console.




```
[15:28:47] Starting 'make run'...
cc atmux.c lib/Matrix2D.c lib/Vector.c lib/CRSMatrix.c -std=c99 -O2 -Ilib -fopenmp -o atmux

./atmux 17000

- Input parameters
size      = 17000
- Executing test...
time (s)= 1.286307
size      = 17000
sparsity = 0.66
chksum    = 244110871193
iters     = 10

[15:28:58] Execution completed successfully (11.197 seconds)
```

Build output | Execution output | Parallelware output

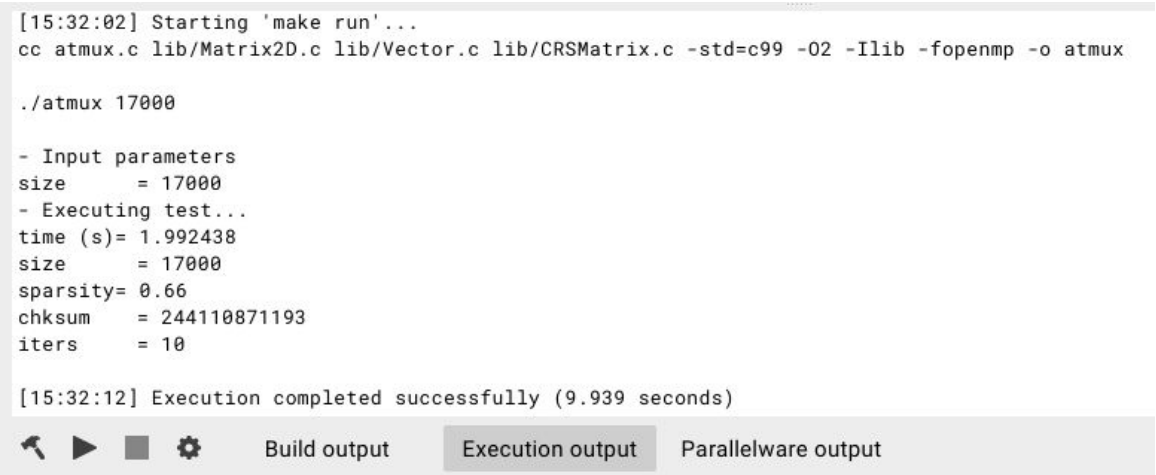
16. Green circles constitute parallelization opportunities, just like those reported by Analyzer's **pwloops** tool. Clicking it will display the parallelization options dialog in which you can select the target standard, hardware and paradigm that you want to parallelize for. The first time, you can just leave the default options and click *Parallelize*. By default the parallelization strategy "*Parallel Loop w/ Atomic*" (or simply "*atomic*") is generated.



The code is updated with the proper pragmas implementing the parallelization. Notice that a new version called Original is automatically created in the version manager on the right. You can create different versions of the file for the different parallel implementations and restore them to perform experimentations.

```
16  void compute(double *val, double *x, double *y, int *col_ind, int *row_ptr, int n) {
17      // y = A^T x
18      #pragma omp parallel default(none) shared(col_ind, n, row_ptr, val, x, y)
19      {
20      #pragma omp for schedule(auto)
21      for (int i = 0; i < n; i++) {
22          for (int k = row_ptr[i]; k < row_ptr[i + 1]; k++) {
23              #pragma omp atomic update
24              y[col_ind[k]] = y[col_ind[k]] + x[i] * val[k];
25          }
26      }
27      } // end parallel
28  }
```

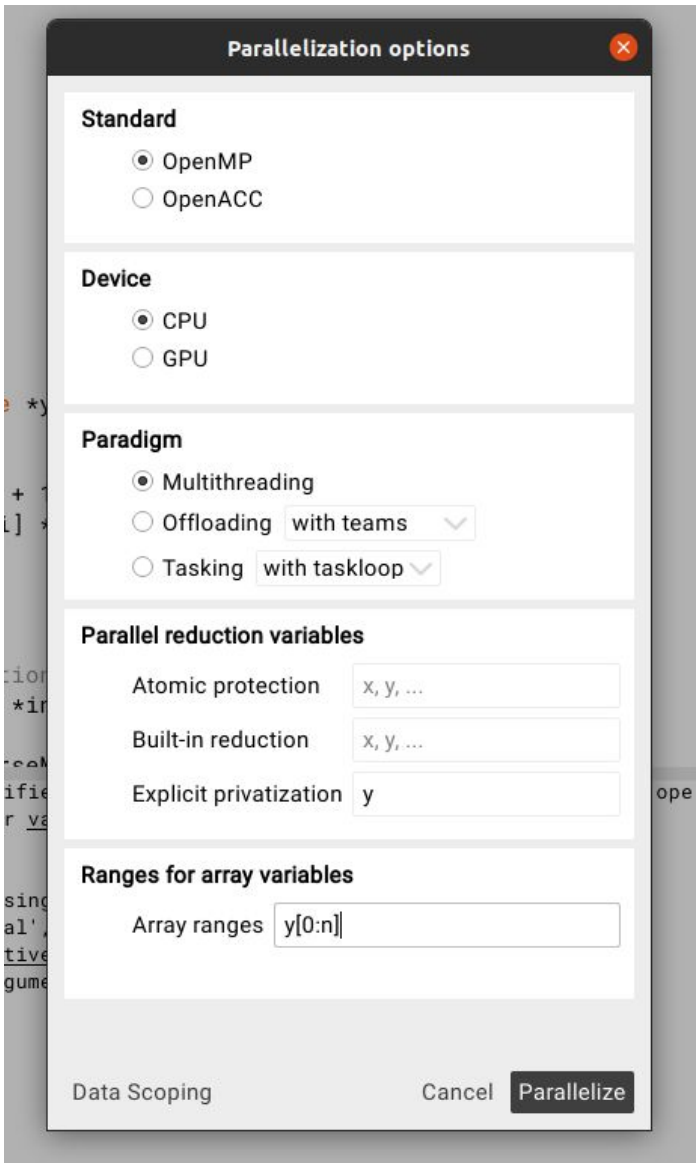
17. Again, press F6 or select the *Project* > *Run* menu entry to run the parallel version of the code.



> Note that the parallel version is correct but it runs slower than the sequential version. So we need to find an alternative parallelization strategy that introduces less parallelization overhead (e.g., create/destroy threads, mutual exclusion mechanisms like atomic operations, synchronization barriers).

18. Save this parallel implementation version by selecting the *File > Create Version* menu entry and giving it a name such as *omp_atomic*.

19. Restore the *Original* version of *atmux.c* and parallelize it again, now using the parallelization strategy "Parallel Loop w/ Explicit Privatization" (or simply "*explicit privatization*"). Enter *y* and *y[0:n]* in the *Explicit privatization* and *Array ranges* input boxes of the parallelization options dialog, respectively.



Clicking again *Parallelize* will generate the following parallel code, for which you can create a new version (e.g. named *omp_privatization*):

```
17  void compute(double *val, double *x, double *y, int *col_ind, int *row_ptr, int n) {
18    // y = A^T x
19    #pragma omp parallel default(none) shared(col_ind, n, row_ptr, val, x, y)
20    {
21    // preamble
22    unsigned int y_length = 0 + n;
23    double *y_private = (double *) malloc(sizeof(double) * y_length);
24    for (int i = 0; i < y_length; ++i) {
25      y_private[i] = 0;
26    }
27    // end preamble
28    #pragma omp for schedule(auto)
29    for (int i = 0; i < n; i++) {
30      for (int k = row_ptr[i]; k < row_ptr[i + 1]; k++) {
31          y_private[col_ind[k]] = y_private[col_ind[k]] + x[i] * val[k];
32      }
33    }
34    // postamble
35    #pragma omp critical
36    for(int i = 0; i < y_length; ++i) {
37      y[i] += y_private[i];
38    }
39    free(y_private);
40    // end postamble
41    } // end parallel
42  }
```

Run again the program by pressing F6 and compare the performance with the previous version.

```
[15:33:26] Starting 'make run'...
cc atmux.c lib/Matrix2D.c lib/Vector.c lib/CRSMatrix.c -std=c99 -O2 -Ilib -fopenmp -o atmux

./atmux 17000

- Input parameters
size      = 17000
- Executing test...
time (s)= 0.629218
size      = 17000
sparsity= 0.66
chksum    = 244110871193
iters     = 10

[15:33:34] Execution completed successfully (8.941 seconds)
```

Build output | Execution output | Parallelware output

> You will notice that the explicit privatization version is faster than the atomic one (even when both are using multithreading on the CPU). Furthermore, it runs twice faster than the sequential version of ATMUX.

20. Now we will try offloading the computation to the GPU. We need to prepare the code as we will need to specify the data ranges to copy between the CPU memory and the GPU memory. Restore the *Original* version, add a new *long long nnz* parameter to the the *compute* function and pass the size of the *in_sparseMat* matrix. Your code should look like the following (see lines 16, 33 and 40):

```c
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include <CRSMatrix.h>
#include <Matrix2D.h>
#include <Vector.h>

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

double getClock();

void compute(double *val, double *x, double *y, int *col_ind, int *row_ptr, int n, long long nnz) {
  // y = A^T x
  for (int i = 0; i < n; i++) {
    for (int k = row_ptr[i]; k < row_ptr[i + 1]; k++) {
        y[col_ind[k]] = y[col_ind[k]] + x[i] * val[k];
    }
  }
}

// Compute sparse matrix-vector multiplication
void atmux(CRSMatrix *in_sparseMat, Vector *in_vec, Vector *out_vec, int n) {

    double *val = CRSMatrix_getData(in_sparseMat);
    double *x = Vector_getData(in_vec);
    double *y = Vector_getData(out_vec);
    int *col_ind = CRSMatrix_colRef(in_sparseMat);
    int *row_ptr = CRSMatrix_rowRef(in_sparseMat);
    long long nnz = CRSMatrix_getSize(in_sparseMat);

    int t, i, k;

    for (t = 0; t < n; t++)
        y[t] = 0;

    compute(val, x, y, col_ind, row_ptr, n, nnz);
}
```

21. Create a new version of the code called *param_nnz*, which will be the starting point to code GPU versions in the following steps.

22. You also need to add offloading flags in the *Makefile*. Double click it to open, add a new line defining *CC = gcc* and append *-foffload=nvptx-none* to the *CFLAGS* variable.

```
SOURCES = atmux.c lib/Matrix2D.c lib/Vector.c lib/CRSMatrix.c
TARGET = atmux
CC = gcc
CFLAGS = -std=c99 -O2 -Ilib -foffload=nvptx-none
...
```

23. Parallelize for OpenMP, GPU and Offloading. You will need to fill the missing information in the *map* clauses, specifically you need to ensure that the following two map clauses are in place:

```
map(to: col_ind[0:nnz], n, row_ptr[0:n+1], val[0:nnz], x[0:n])
map(tofrom:y[0:n])
```

```
16  void compute(double *val, double *x, double *y, int *col_ind, int *row_ptr, int n, long long nnz) {
17    // y = A^T x
18    #pragma omp target teams distribute parallel for shared(col_ind, n, row_ptr, val, x) schedule(auto) \
19    map(to: col_ind[0:nnz], n, row_ptr[0:n+1], val[0:nnz], x[0:n]) map(tofrom:y[0:n])
20    for (int i = 0; i < n; i++) {
21      for (int k = row_ptr[i]; k < row_ptr[i + 1]; k++) {
22          #pragma omp atomic update
23          y[col_ind[k]] = y[col_ind[k]] + x[i] * val[k];
24      }
25    }
```

24. Build and run the OpenMP offloading version. Save it as *omp_gpu* if you like.

25. Open the project configuration and change the build command to *make omp* and the run command to *srun ./atmux 17000*.

```
[08:32:35] Starting 'srun ./atmux 17000'...
- Input parameters
size      = 17000
- Executing test...

time (s)= 3.468869
size      = 17000
sparsity= 0.66
chksum    = 244110871193
iters     = 10

[08:32:47] Execution completed successfully (12.877 seconds)
```

Build output | Execution output | Parallelware output

26. Restore version *param_nnz* and now parallelize for OpenACC, GPU and Offloading. Again, you need to fill the missing information in the *copy* clauses:

```
copyin(col_ind[0:nnz], n, row_ptr[0:n+1], val[0:nnz], x[0:n])
copy(y[0:n])
```

```
16  void compute(double *val, double *x, double *y, int *col_ind, int *row_ptr, int n, long long nnz) {
17    // y = A^T x
18    #pragma acc data copyin(col_ind[0:nnz], n, row_ptr[0:n+1], val[0:nnz], x[0:n]) copy(y[0:n])
19    {
20    #pragma acc parallel
21    {
22    #pragma acc loop
23    for (int i = 0; i < n; i++) {
24      for (int k = row_ptr[i]; k < row_ptr[i + 1]; k++) {
25          #pragma acc atomic update
26          y[col_ind[k]] = y[col_ind[k]] + x[i] * val[k];
27      }
28    }
29    } // end parallel
30    } // end data
31  }
```

27. Open the project configuration, change the build command to *make acc* and ensure that the run command is *srun ./atmux 17000*.

28. Build and run the OpenMP offloading version. Save it as *acc_gpu* if you like.