Pathfinder Workflow with SAXPY

Lesson Objectives

Upon completing this notebook you should be able to understand and apply the following concepts:

- Understand how the Lucata Pathfinder workflow supports x86 execution, simulation, and execution in multiple configurations on the hardware.
- Use basic Slurm commands to submit a job on the Pathfinder hardware.
- Understand the debugging flow for the Pathfinder platform
  - Use OpenCilk and the memoryweb library to test code on x86 systems
  - Simulate your Lucata Cilk code with `emusim` and profile with `emusim_profile`
  - Run a single node hardware job on the Pathfinder with Slurm
  - Run multi-node and multi-chassis jobs

Environment Setup

We rerun our environment setup for the notebook as with our previous notebooks. However, notice that we have added a new line with `X86FLAGS`, which we will use to compile our code on an x86 platform with OpenCilk.

In [1]:
```
%load_ext slurm_magic
import os
#Used to display the code file directly within the notebook
from IPython.display import Code

#Set the path to the latest toolset
LUCATA_BASE="/tools/emu/pathfinder-sw/22.09-beta"

#Get the path to where all code samples are
os.environ["USER_NOTEBOOK_CODE"]=os.path.dirname(os.getcwd())
os.environ["X86FLAGS"] = "-I/tools/lucata/pathfinder-sw/22.09-beta/include/cilk/ -I/tools/emu/path
os.environ["PATH"]=os.pathsep.join([os.path.join(LUCATA_BASE,"bin"),os.environ["PATH"]])
os.environ["FLAGS"]="-I"+LUCATA_BASE+"/include/memoryweb/"+" -L"+LUCATA_BASE+"/lib -lmemoryweb"
os.environ["LUCATA_BASE"]="/tools/lucata/pathfinder-sw/22.09-beta/"

!printf "Lucata Cilk compilation flags are $FLAGS\n\n"
!printf "x86 flags are $X86FLAGS\n"
```

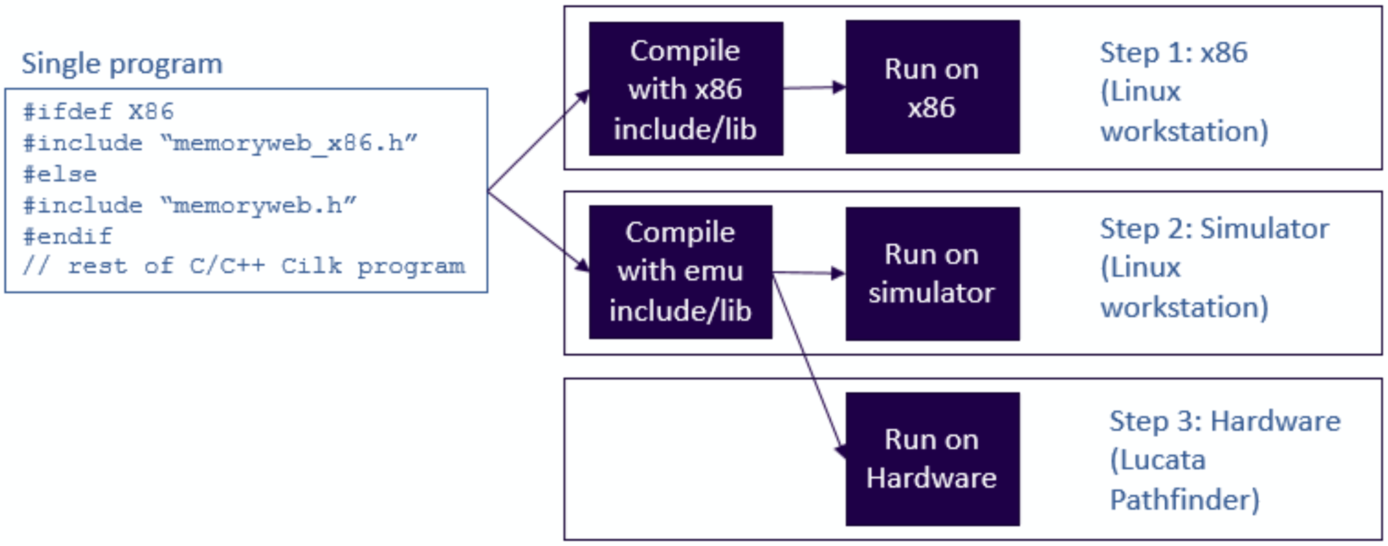
```
Lucata Cilk compilation flags are -I/tools/emu/pathfinder-sw/22.09-beta/include/memoryweb/ -L/tool
s/emu/pathfinder-sw/22.09-beta/lib -lmemoryweb

x86 flags are -I/tools/lucata/pathfinder-sw/22.09-beta/include/cilk/ -I/tools/emu/pathfinder-sw/2
2.09-beta/x86/include/emu_c_utils /tools/emu/pathfinder-sw/22.09-beta/x86/lib/libemu_c_utils.a
```

This notebook goes along with the Lucata workflow slides, so please follow along with the slides for a supplemental resource.

Lucata Pathfinder Workflow

This figure shows the workflow for using the Pathfinder ecosystem and hardware. Since the Pathfinder is programmed using a variant of the Cilk programming language, code written for this platform can be run on x86 systems using the Lucata toolchain, some GCC versions (5-7), or an appropriate Clang branch like OpenCilk.


Single program
#ifdef X86
#include "memoryweb_x86.h"
#else
#include "memoryweb.h"
#endif
// rest of C/C++ Cilk program
Compile with x86 include/lib
Run on x86
Step 1: x86 (Linux workstation)
Compile with emu include/lib
Run on simulator
Step 2: Simulator (Linux workstation)
Run on Hardware
Step 3: Hardware (Lucata Pathfinder)

This notebook takes one of our previous Saxpy examples and uses it as part of a workflow that shows how to run code on the x86 system, simulator, and finally on the hardware. Just to revisit, we are using the basic SAXPY "1D allocation" kernel from Notebook 2.

X86 Execution

As a first step, we need to update the code slightly to allow it to be compiled for x86 platforms using "memoryweb_x86.h". This compatibility header tells the compiler to compile for an x86 variant of Cilk rather than the Lucata version. The differences are that some Lucata-specific commands like `cilk_spawn_at` don't exist in most standard x86 Cilk APIs.

Also, note the inclusion of the "emu_c_utils" header, which provides additional helper functions.

In [2]:
```
Code('saxpy-1d-workflow.c')
```

Out[2]:
```
//saxpy-1d-workflow.c
#include <stdio.h>
#include <stdlib.h>
#include <cilk.h>

//If x86 is specified use the x86 compatibility headers with Cilk; otherwise use the Luc
ata toolchain
#ifdef X86
  #include "memoryweb_x86.h"
  #include <emu_c_utils.h>
#else
  #include "memoryweb.h"
  #include <emu_c_utils/emu_c_utils.h>
#endif

void saxpy(long n, long a, long *x, long *y)
{
  for (long i = 0; i < n; i++)
    y[i] += a * x[i];
}

int main(int argc, char **argv)
{
  if(argc < 4){
      printf("Pass at least 3 arguments!\n");
      printf("saxpy-1d-workflow <num_threads> <array_size> <constant>\n");
      exit(1);
  }

  long nth = atol(argv[1]); // number threads
  long size = atol(argv[2]); // array size
  long aval = atol(argv[3]); // constant

  long *x = mw_malloc1dlong(size);
  long *y = mw_malloc1dlong(size);

  for (long i = 0; i < size; i++) {
    x[i] = i; y[i] = 0;
  }

  long grain = size / nth; // elts per thread

  //Timing flag to simulation only; ignored on x86 and HW
  starttiming();
  for (long i = 0, j = 0; i < nth; i++, j += grain)
    cilk_spawn saxpy(grain, aval, &x[j], &y[j]);
  cilk_sync;

  printf("SAXPY complete!\n");
}
```

We can then compile this code for execution on an x86 system as follows. Note that we are using the OpenCilk version of Clang, which is not Lucata-specific! Note that we have added both `${X86FLAGS}` and `-DX86` to our compilation string so that `memoryweb_x86.h` is used instead of `memoryweb.h`.

In [3]:

```
%%bash
set -x
/tools/lucata/opencilk-2.0.1/bin/clang ${X86FLAGS} -DX86 saxpy-1d-workflow.c -o saxpy-1d-workflow-
set +x
ls
```

```
Makefile
NB04-saxpy-workflow.html
NB04-saxpy-workflow.ipynb
saxpy-1d-workflow.c
saxpy-1d-workflow.cdc
saxpy-1d-workflow.mwx
saxpy-1d-workflow-x86
sbatch-hello-world.sh
sbatch-saxpy-mn.sh
sbatch-saxpy-sn.sh
```

```
+ /tools/lucata/opencilk-2.0.1/bin/clang -I/tools/lucata/pathfinder-sw/22.09-beta/include/cilk/ -I
/tools/emu/pathfinder-sw/22.09-beta/x86/include/emu_c_utils /tools/emu/pathfinder-sw/22.09-beta/x8
6/lib/libemu_c_utils.a -DX86 saxpy-1d-workflow.c -o saxpy-1d-workflow-x86
+ set +x
```

Testing out x86 SAXPY

After compiling with OpenCilk, we run the Saxpy example with 8 threads, an array of size 2^20 and a constant value of 5.

In [4]:
```
%%bash

time CILK_NWORKERS=1 ./saxpy-1d-workflow-x86 8 1048576 5
```

```
SAXPY complete!

real	0m0.022s
user	0m0.013s
sys	0m0.006s
```

Testing SAXPY with additional x86 threads

Rerun the example with more workers - the timing should go down slightly. Remember you may be running this notebook with a limited number of threads (by default usually 2), so adding more workers may not substantially speed this example up unless you run the notebook with more cores!

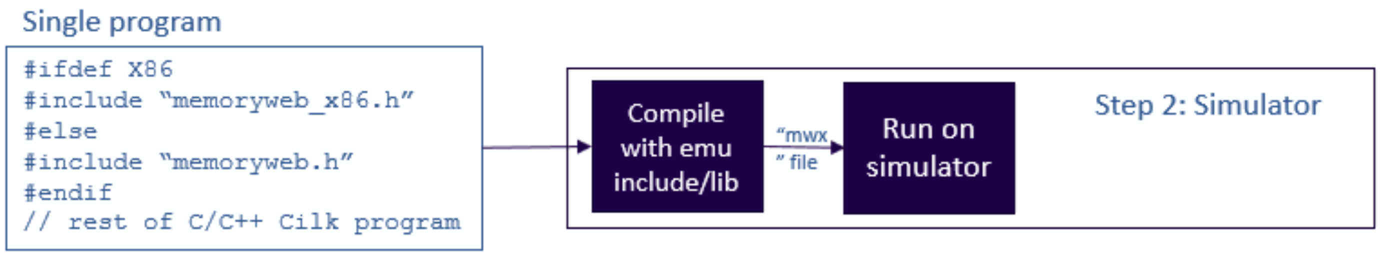
In [5]:
```
%%bash
time CILK_NWORKERS=2 ./saxpy-1d-workflow-x86 8 1048576 5
```

```
SAXPY complete!

real	0m0.020s
user	0m0.011s
sys	0m0.008s
```

Simulator Execution

Once we have tested our program with x86 Cilk execution we can proceed to test with the Lucata simulator, `emusim.x` . This simulator is a single-threaded simulator that operates on a detailed SystemC model of the Pathfinder system. As such, it is somewhat slow and should normally be used in "untimed" mode to verify functionality. Since the Pathfinder hardware does not currently include a debugger or runtime profiler, the simulator should also be used to debug issues and check basic performance profiling.



```
#ifdef X86
#include "memoryweb_x86.h"
#else
#include "memoryweb.h"
#endif
// rest of C/C++ Cilk program
```

The best way to limit the runtime of the simulator is to run in "untimed" mode, meaning that no relative clocks are used to estimate performance of the application. You can do this by either commenting out any `starttiming()` calls in your code or using the `--ignore_starttiming` flag when simulating your code.

In [6]:
```
%%bash
emu-cc -o saxpy-1d-workflow.mwx $FLAGS saxpy-1d-workflow.c -lemu_c_utils
```

```
/net/tools/emu/pathfinder-sw/22.09-beta/bin/clang: /lib64/libtinfo.so.5: no version information av
ailable (required by /net/tools/emu/pathfinder-sw/22.09-beta/bin/clang)
/net/tools/emu/pathfinder-sw/22.09-beta/bin/clang-6.0: /lib64/libtinfo.so.5: no version informatio
n available (required by /net/tools/emu/pathfinder-sw/22.09-beta/bin/clang-6.0)
/net/tools/emu/pathfinder-sw/22.09-beta/bin/opt: /lib64/libtinfo.so.5: no version information avai
lable (required by /net/tools/emu/pathfinder-sw/22.09-beta/bin/opt)
/net/tools/emu/pathfinder-sw/22.09-beta/bin/opt: /lib64/libtinfo.so.5: no version information avai
lable (required by /net/tools/emu/pathfinder-sw/22.09-beta/bin/opt)
```

In [7]:
```
%%bash
emusim.x --ignore_starttiming -m 24 --total_nodes 1 saxpy-1d-workflow.mwx 8 128 5.0
```

```
Start untimed simulation with local date and time= Tue Sep 27 12:23:11 2022

SAXPY complete!
End untimed simulation with local date and time= Tue Sep 27 12:23:11 2022

        SystemC 2.3.3-Accellera --- Sep  7 2022 09:15:59
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED
```

Simulating with more nodes

We can also simulate a larger system size as we have seen in previous notebook examples. Here we test with four simulated Pathfinder nodes.

In [8]:
```
%%bash
emusim.x --ignore_starttiming -m 24 --total_nodes 4 saxpy-1d-workflow.mwx 8 128 5.0
```

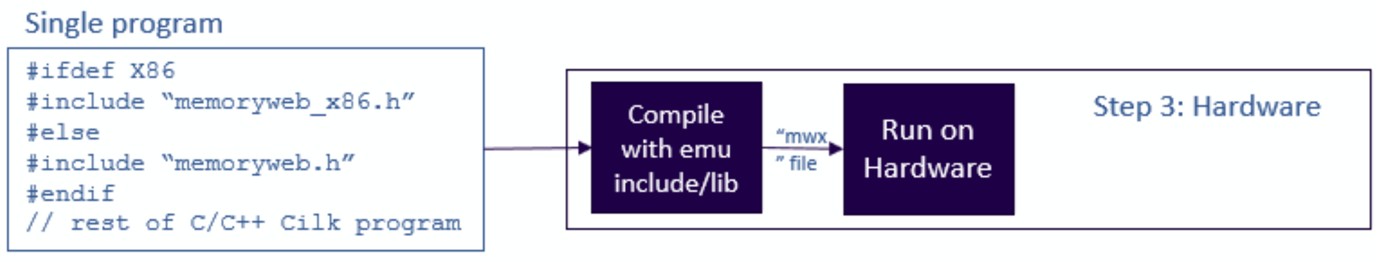
```
Start untimed simulation with local date and time= Tue Sep 27 12:23:11 2022

SAXPY complete!
End untimed simulation with local date and time= Tue Sep 27 12:23:11 2022

        SystemC 2.3.3-Accellera --- Sep  7 2022 09:15:59
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED
```

Hardware Execution

Once our code works with the simulator, and we feel we have optimized it enough it is time to start running on the hardware. You should first try to run your code in single-node fashion on the Pathfinder and then scale up to multi-node execution. Note that the `emu_handler_and_loader` command is meant for single-node execution while `emu_multinode_exec` is meant for multinode execution. All multi-node jobs must be run from node 0 in the Pathfinder system.



```
#ifdef X86
#include "memoryweb_x86.h"
#else
#include "memoryweb.h"
#endif
// rest of C/C++ Cilk program
```

Using Slurm for hardware execution

The Rogues Gallery recently has migrated to using Slurm for execution of code on the Pathfinder system. For more information on how to use Slurm and commands like sbatch, please see our ReadTheDocs page for Slurm.

Here we will cover basic execution of a single node and single-chassis job. For a single node job, we use the quality of service flag `-q single-node` and submit to the Pathfinder cluster `-M pathfinder`. The `--wrap "emu_handler_and_loader 0 0 -- saxpy-1d-workflow.mwx 8 128 5.0"` command just runs the standard `emu_handler_and_loader` as part of the Slurm job submission.

In [9]:

```
%%bash
#Clean up any older Slurm output files
rm -f *.out

#Run the Saxpy command with Slurmemu_handler_and_loader saxpy-1d-workflow.mwx 8 128 5.0
sbatch -M pathfinder -q single-node --wrap "emu_handler_and_loader 0 0 -- saxpy-1d-workflow.mwx 8

#If the job runs successfully, the output file should print out "SAXPY complete!"
sleep 5
#List the Slurm output files.
printf "\nShowing Slurm output files:\n"
ls *.out
#Show the content of the latest output file
printf "\nOutput from the run:\n"
less slurm-*.out
```

```
Submitted batch job 134217841 on cluster pathfinder

Showing Slurm output files:
slurm-134217841.out

Output from the run:
SAXPY complete!
```

Using sbatch scripts

While we have run the previous two commands from the command-line, it is more effective to create an "batch file" that we can reuse. This file is called `sbatch-saxpy-sn.sh` in the current directory. To run this script you would execute the following: `sbatch sbatch-saxpy-sn.sh` .

```
#!/bin/bash
#==== Specify the time for the reservation (HH:MM:SS)
#SBATCH --time=0:15:00
#==== Create a useful name for output files
#SBATCH --job-name=saxpy-1d-workflow
#==== Specify the cluster to use
#SBATCH -M pathfinder
#==== Specify the QoS parameter. Valid options are single-node, single-chassis, multi-
chassis
#SBATCH -q single-node

# Run the command on a single node and report the time it took
time emu_handler_and_loader 0 0 -- saxpy-1d-workflow.mwx 8 2097152 5.0
```

In [10]:
```
%%bash
#Clean up any older Slurm output files
rm -f *.out

#Run the same command but using a batch file instead of --wrap
sbatch sbatch-saxpy-sn.sh

#If the job runs successfully, the output file should print out "SAXPY complete!"
sleep 10
#Show the content of the latest output file
printf "\nOutput from the run:\n"
less slurm-*.out
```

```
Submitted batch job 134217842 on cluster pathfinder

Output from the run:
SAXPY complete!
```

More information on Slurm queues on CRNCH RG

We can also run commands like `sinfo` and `squeue` to see the entire state of the RG cluster. Here we run `sinfo --federation` because the Pathfinder is set up with a separate Slurm cluster instance.

In [11]:
```
#Show the real-time status of the Pathfinder nodes.
%sinfo --federation -M pathfinder
```

Out[11]:

				CLUSTER:	**pathfinder**
PARTITION	**AVAIL**	**TIMELIMIT**	**NODES**	STATE	NODELIST
rg-pathfinder*	**up**	**180-00:00:**	**1**	alloc	c2n1
			31	idle	c0n[0-7],c1n[0-7],c2n[0,2-7],c3n[0-7]

Debugging and Optimization Strategy for the Pathfinder

If you've made it this far, you likely have some idea of the key steps for compiling and running programs with the Pathfinder. You might notice that there are several steps to compiling and running for the Pathfinder - this is because it is a novel parallel architecture that is constantly evolving!

However, in general there are three phases for running large applications at scale on the Pathfinder: 1) x86 testing for functional correctness and traditional Cilk-based timing information, 2) Lucata emusim testing for functional correctness, 3) Single-node and multi-node testing for timing results. The following steps break down this workflow into more detail and can be used as a reference for your development with the Pathfinder platform.

x86 Testing

1) Compile on x86 (using memoryweb_x86.h)
2) Run on x86 with a single thread (CILK_NWORKERS = 1)
3) Run on x86 multi-threaded (CILK_NWORKERS > 1)

Lucata simulation

4) Compile for Lucata Cilk
5) Simulate with emusim in untimed mode (--ignore_starttiming)

6) Simulate on emusim with randomly initialized memory (--initialize_memory)
7) Simulate with emusim in timed mode. (starttiming() in code)

Pathfinder execution

8) Execute on single-node HW
9) Execute on multi-node HW
10) Increase input size gradually. Always use the smallest input set that will finish/recreate the problem in a reasonable amount of time!

Pathfinder optimization

11) Use the new hardware performance counters to help find areas for optimization.

Exercises

To further your understanding of this topic we encourage you to try the following:
1) Restart the notebook and change the memory size and numbers of nodes that are simulated. How do the statistics change?
2) Investigate the other output files like the `.mps` file and understand how they are different for different applications.
More details on visualizing these files are in the next Notebook.

Multi-node execution (Optional Exercise)

Multi-node jobs run on a different chassis (c0 or c1 for our system), and they run in parallel across all the nodes (8 per chassis) and all the compute engines within a node (currently 24 per node. To run a multi-chassis job, we use the `-q single-chassis` flag which specifies that our code should run on multiple nodes within a chassis.

In [12]:
```
%%bash
#Clean up any older Slurm output files
rm -f *.out

#Note that this may take a bit of time to complete. Uncomment if you'd like to test multi-node exe
#sbatch sbatch-saxpy-mn.sh
#less slurm-*.out
```