Notebook #3 - Pathfinder Workflow with SAXPY

Initial steps: We must import specific environment variables to point to the user's notebook code director and to the Lucata tools.

In [1]:
```
import os
os.environ["USER_NOTEBOOK_CODE"]=os.environ["HOME"]+"/pearc-tutorial/code"
os.environ["PATH"]=os.pathsep.join(["/tools/emu/pathfinder-sw/21.06/bin",os.environ["PA
```

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 MIT's Tapir.

Lucata Workflow

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.

```
//saxpy-1d-workflow.c
#include <stdio.h>
#include <stdlib.h>
#include <cilk/cilk.h>

//If x86 is specified use the x86 compatibility headers with Cilk; otherwise
use the Lucata toolchain
#ifdef X86
  #include <stdio.h>
  #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)
{
  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;
}
```

We can then compile this code for execution on an x86 system as follows:

In [2]:

```
%%bash
export EMU_BASE=/tools/emu/pathfinder-sw/21.06
gcc-7 -I ${EMU_BASE}/x86/include/emu_c_utils -I ${EMU_BASE}/include -fcilkplus -DX86 sa
```

```
In file included from saxpy-1d-workflow.c:14:0:
/tools/emu/pathfinder-sw/21.06/x86/include/emu_c_utils/memoryweb_x86.h:21:0: warning: "c
ilk_migrate_hint" redefined
 #define cilk_migrate_hint(X) (void)(X)

In file included from saxpy-1d-workflow.c:9:0:
/tools/emu/pathfinder-sw/21.06/include/cilk/cilk.h:84:0: note: this is the location of t
he previous definition
 # define cilk_migrate_hint(x) _Cilk_migrate_hint(x)

In file included from saxpy-1d-workflow.c:14:0:
/tools/emu/pathfinder-sw/21.06/x86/include/emu_c_utils/memoryweb_x86.h:22:0: warning: "c
ilk_spawn_at" redefined
 #define cilk_spawn_at(X) (void)(X); cilk_spawn

In file included from saxpy-1d-workflow.c:9:0:
/tools/emu/pathfinder-sw/21.06/include/cilk/cilk.h:85:0: note: this is the location of t
he previous definition
```

```
# define cilk_spawn_at(x) cilk_migrate_hint(x); cilk_spawn
```

In [3]:

```
%%bash
CILK_NWORKERS=4 ./saxpy-1d-workflow-x86 8 128 5
```

```
SAXPY complete!
```

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.

Lucata Workflow

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 [4]:

```
!emu-cc -o saxpy-1d-workflow.mwx saxpy-1d-workflow.c -lemu_c_utils
```

In [5]:

```
%%bash
emusim.x -- saxpy-1d-workflow.mwx 8 128 5.0
```

```
Start untimed simulation with local date and time= Mon Jul 19 02:25:26 2021

End untimed simulation with local date and time= Mon Jul 19 02:25:26 2021

SysC Enumeration done. Program launching...
Simulation @0 s with local date and time= Mon Jul 19 02:25:26 2021

SAXPY complete!

Info: /OSCI/SystemC: Simulation stopped by user.
        SystemC 2.3.3-Accellera --- Jun 22 2021 17:09:43
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED
```

In [6]:

```
%%bash
emusim.x --total_nodes 4 --untimed_short_trace -- saxpy-1d-workflow.mwx 8 128 5.0
```

```
Start untimed simulation with local date and time= Mon Jul 19 02:25:33 2021

End untimed simulation with local date and time= Mon Jul 19 02:25:33 2021

SysC Enumeration done. Program launching...
Simulation @0 s with local date and time= Mon Jul 19 02:25:33 2021
```

```
SAXPY complete!

Info: /OSCI/SystemC: Simulation stopped by user.
        SystemC 2.3.3-Accellera --- Jun 22 2021 17:09:43
        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.

Lucata Workflow

In [ ]:

```
%%bash
ssh pathfinder1.crnch.gatech.edu
ssh n0
cd ${USER_NOTEBOOK_CODE}/03-saxpy-workflow

emu_handler_and_loader saxpy-1d-workflow.mwx 8 128 5.0

emu_multinode_exec saxpy-1d-workflow.mwx 8 128 5.0
```

Debugging 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. As an added important note, Eric Hein shares this debugging strategy that builds on the usage of the x86 and simulator models to ensure correct program execution and scaling.

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

4) Compile for Emu 5) Run on emusim in untimed mode (--ignore_starttiming) 6) Run on emusim with randomly initialized memory (--initialize_memory)

7) Run on emusim in timed mode. (starttiming() in code)

8) Run on single-node HW 9) Run 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!

Postcript

Once we've finished our testing, we can clean up some of the logfiles that we used for this example with `make clean` . Uncomment the following line to clean this directory.

In [ ]:
```
#!make clean
```

In [ ]:
```
```