NB03-saxpy-basics

September 21, 2022

1 Lucata Cilk Programming Basics with SAXPY

First we set environment variables to point to the user's notebook code and to the Lucata tools.

We'll also set SIMFLAGS to simulate 16 MiB of memory on 4 simulated nodes for each simulation.

[1]:
```
import os
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["PATH"]=os.pathsep.join([os.path.join(LUCATA_BASE,"bin"),os.
 ↪environ["PATH"]])
os.environ["CCFLAGS"]="-I"+LUCATA_BASE+"/include/"+" -L"+LUCATA_BASE+"/lib␣
 ↪-lmemoryweb"
os.environ["SIMFLAGS"]="-m 24 --total_nodes 4"
```

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

1.1 SAXPY with Cilk Spawn

Our first example shows an example of Single-precision AX Plus Y (SAXPY), a basic linear algebra kernel that combines scalar multiplication and vector addition. As shown in the saxpy kernel, the output, `y`, is equal to the sum of the constant `a` multiplied by the elements of a vector `x`.

This first example shows how to implement SAXPY using the Cilk functions, `cilk_spawn` and `cilk_sync`. Note that the Lucata architecture operates on a particular "grain size", which is specified by the number of threads (the first argument to the program).

*For examples of SAXPY in other parallel languages please check out this NVIDIA developer blog on SAXPY.

[2]:
```
Code('saxpy.c')
```

[2]:

```c
#include <stdio.h>
#include <stdlib.h>
#include <cilk/cilk.h>
#include <memoryweb/timing.h>

void saxpy(long n, float a, float *x, float *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
  float aval = atof(argv[3]); // constant
  float *x = malloc(size * sizeof(*x));
  float *y = malloc(size * sizeof(*y));
  for (long i = 0; i < size; i++) {
    x[i] = i; y[i] = 0;
  }
  long grain = size / nth; // elts per thread

  lu_profile_perfcntr(PFC_CLEAR, "CLEAR COUNTERS");
  lu_profile_perfcntr(PFC_START, "START COUNTERS");

  for (long i = 0, j = 0; i < nth; i++, j += grain)
    cilk_spawn saxpy(grain, aval, &x[j], &y[j]);
  cilk_sync;

  lu_profile_perfcntr(PFC_STOP, "STOP COUNTERS");
}
```

We'll test compiling and running this example with a 4 and 8 threads, an array with 32 or 128 elements, and a constant value `a` of 5.0. Then we will check the .cdc files for the simulated runtime of the profiled region.

This notebook will use 8 threads and an array of size 128 for all examples after this one.

The line that says "Emu system run time" is the simulated time to complete the SAXPY operation.

[3]:
```bash
%%bash
emu-cc -o saxpy.mwx $CCFLAGS saxpy.c
emusim.x $SIMFLAGS -- saxpy.mwx 4 32 5.0 2>/dev/null
less saxpy.cdc | grep "Emu system run time"
```

```
Start untimed simulation with local date and time= Wed Sep 21 21:33:45 2022

Timed simulation starting…
```

```
End untimed simulation with local date and time= Wed Sep 21 21:33:49 2022


Info: /OSCI/SystemC: Simulation stopped by user.
Emu system run time 0.000122 sec==122232000 ps
```

[4]:
```
%%bash
emusim.x $SIMFLAGS -- saxpy.mwx 8 128 5.0 2>/dev/null
less saxpy.cdc | grep "Emu system run time"
```

```
Start untimed simulation with local date and time= Wed Sep 21 21:33:50 2022

Timed simulation starting…
End untimed simulation with local date and time= Wed Sep 21 21:33:56 2022


Info: /OSCI/SystemC: Simulation stopped by user.
Emu system run time 0.000172 sec==172236000 ps
```

1.2 SAXPY with Cilk For

`cilk_for` can be used to launch one thread per loop iteration in a fashion similar to traditional OpenMP pragma-based `omp parallel for` loops. Note here that the programmer must explicitly specify a grainsize to partition up the input array.

[5]:
```
Code('saxpy-for.c')
```

[5]:
```c
#include <stdio.h>
#include <stdlib.h>
#include <cilk/cilk.h>
#include <memoryweb/timing.h>

int main(int argc, char **argv)
{
  long size = atol(argv[1]); // array size
  float aval = atof(argv[2]); // constant

  float *x = malloc(size * sizeof(*x));
  float *y = malloc(size * sizeof(*y));

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

  lu_profile_perfcntr(PFC_CLEAR, "CLEAR COUNTERS");
  lu_profile_perfcntr(PFC_START, "START COUNTERS");
```

```
  #pragma cilk grainsize = 8
  cilk_for (long i = 0; i < size; i++) {
    y[i] += aval * x[i];
  }

  lu_profile_perfcntr(PFC_STOP, "STOP COUNTERS");
}
```

[6]:

```
%%bash
emu-cc $CCFLAGS -o saxpy-for.mwx  saxpy-for.c
emusim.x $SIMFLAGS -- saxpy-for.mwx 8 128 5.0 2>/dev/null
less saxpy-for.cdc | grep "Emu system run time"
```

```
Start untimed simulation with local date and time= Wed Sep 21 21:34:03 2022

Timed simulation starting…
End untimed simulation with local date and time= Wed Sep 21 21:34:08 2022


Info: /OSCI/SystemC: Simulation stopped by user.
Emu system run time 0.000122 sec==122232000 ps
```

1.3 SAXPY with Distributed Allocation (1D)

In this example, `memoryweb.h` is included for Lucata-specific distributed allocation strategies while `mw_malloc1dlong` is used to distribute data across different nodes within the system.

[7]:

```
Code('saxpy-1d.c')
```

[7]:

```
#include <stdlib.h>
#include <cilk/cilk.h>
#include <memoryweb/memoryweb.h>
#include <memoryweb/timing.h>

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

  lu_profile_perfcntr(PFC_CLEAR, "CLEAR COUNTERS");
  lu_profile_perfcntr(PFC_START, "START COUNTERS");

  for (long i = 0, j = 0; i < nth; i++, j += grain)
    cilk_spawn saxpy(grain, aval, &x[j], &y[j]);
  cilk_sync;

  lu_profile_perfcntr(PFC_STOP, "STOP COUNTERS");
}
```

[8]:

```
%%bash
emu-cc $CCFLAGS -o saxpy-1d.mwx  saxpy-1d.c
emusim.x $SIMFLAGS -- saxpy-1d.mwx 8 128 5.0 2>/dev/null
less saxpy-1d.cdc | grep "Emu system run time"
```

```
Start untimed simulation with local date and time= Wed Sep 21 21:34:14 2022

Timed simulation starting…
End untimed simulation with local date and time= Wed Sep 21 21:34:21 2022


Info: /OSCI/SystemC: Simulation stopped by user.
Emu system run time 0.000194 sec==194460000 ps
```

1.4 SAXPY Distributed Spawn with migrate_hint

The `migrate_hint` allows the programmer to pass a pointer that is then used by the next `cilk_spawn` operation to efficiently jump to a specific part of a distributed array. Here the migration hint is specifying a "directed spawn" to the location where `y[j]` is located. Note that `cilk_spawn_at` provides a similar purpose by combining a spawn and migration hint operation into one call.

[9]:

```
Code('saxpy-1d-hint.c')
```

[9]:

```
#include <stdlib.h>
#include <cilk/cilk.h>
#include <memoryweb/memoryweb.h>
#include <memoryweb/timing.h>

void saxpy(long n, long a, long *x, long *y)
{
```

```c
  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

  lu_profile_perfcntr(PFC_CLEAR, "CLEAR COUNTERS");
  lu_profile_perfcntr(PFC_START, "START COUNTERS");

  for (long i = 0, j = 0; i < nth; i++, j += grain) {
    cilk_migrate_hint(&y[j]);
    cilk_spawn saxpy(grain, aval, &x[j], &y[j]);
  } cilk_sync;

  lu_profile_perfcntr(PFC_STOP, "STOP COUNTERS");
}
```

[10]:

```bash
%%bash
emu-cc $CCFLAGS -o saxpy-1d-hint.mwx  saxpy-1d-hint.c
emusim.x $SIMFLAGS -- saxpy-1d-hint.mwx 8 128 5.0 2>/dev/null
less saxpy-1d-hint.cdc | grep "Emu system run time"
```

```
Start untimed simulation with local date and time= Wed Sep 21 21:34:28 2022

Timed simulation starting…
End untimed simulation with local date and time= Wed Sep 21 21:34:35 2022


Info: /OSCI/SystemC: Simulation stopped by user.
Emu system run time 0.0002 sec==200016000 ps
```

1.5 SAXPY with 2D Distributed Allocation

This example shows the usage of `cilk_spawn_at` and 2D block allocation of data across the Lucata nodes. In this case, the number of threads matches the number of blocks and the work done by each thread is the block size.

[11]:
```
Code('saxpy-2d-spawn-at.c')
```

[11]:
```
#include <stdlib.h>
#include <cilk/cilk.h>
#include <memoryweb/memoryweb.h>
#include <memoryweb/timing.h>

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

int main(int argc, char **argv)
{
  long num = atol(argv[1]); // number blocks
  long size = atol(argv[2]); // block size
  float aval = atof(argv[3]); // constant
  float **x = mw_malloc2d(num, size * sizeof(*x));
  float **y = mw_malloc2d(num, size * sizeof(*y));

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

  lu_profile_perfcntr(PFC_CLEAR, "CLEAR COUNTERS");
  lu_profile_perfcntr(PFC_START, "START COUNTERS");

  for (long i = 0; i < num; i++) {
    cilk_spawn_at (y[i]) saxpy(size, aval, x[i], y[i]);
  }
  cilk_sync;

  lu_profile_perfcntr(PFC_STOP, "STOP COUNTERS");
}
```

[12]:
```
%%bash
emu-cc $CCFLAGS -o saxpy-2d-spawn-at.mwx  saxpy-2d-spawn-at.c
emusim.x $SIMFLAGS -- saxpy-2d-spawn-at.mwx 8 128 5.0 2>/dev/null
less saxpy-2d-spawn-at.cdc | grep "Emu system run time"
```

```
Start untimed simulation with local date and time= Wed Sep 21 21:34:40 2022

Timed simulation starting…
```

```
End untimed simulation with local date and time= Wed Sep 21 21:35:00 2022


Info: /OSCI/SystemC: Simulation stopped by user.
Emu system run time 0.000539 sec==538932000 ps
```

1.6 SAXPY with Local Allocation

This example shows a variation of the previous 2D code with a local allocation for the output. You will notice that the local allocation for the output (as opposed to 2D allocation) results in more migrations overall.

[13]:
```
Code('saxpy-local-spawn-at.c')
```

[13]:
```c
#include <stdlib.h>
#include <cilk/cilk.h>
#include <memoryweb/memoryweb.h>
#include <memoryweb/timing.h>

void saxpy4(long n, float a, float *x, float *y)
{
  for (long i = 0; i < n; i++)
    y[i] += a * x[i];
}

int main(int argc, char **argv)
{
  long num = atol(argv[1]); // number blocks
  long size = atol(argv[2]); // block size
  float aval = atof(argv[3]); // constant
  float **x = mw_malloc2d(num, size * sizeof(*x));
  float *y = mw_localmalloc(num * size * sizeof(*y), x[0]);

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


  lu_profile_perfcntr(PFC_CLEAR, "CLEAR COUNTERS");
  lu_profile_perfcntr(PFC_START, "START COUNTERS");

  for (long i = 0; i < num; i++) {
    cilk_spawn_at (x[i]) saxpy4(size, aval, x[i], &y[i * size]);
  }
  cilk_sync;
```

```
  lu_profile_perfcntr(PFC_STOP, "STOP COUNTERS");
}
```

[14]:
```
%%bash
emu-cc $CCFLAGS -o saxpy-local-spawn-at.mwx  saxpy-local-spawn-at.c
emusim.x $SIMFLAGS -- saxpy-local-spawn-at.mwx 8 128 5.0 2>/dev/null
less saxpy-local-spawn-at.cdc | grep "Emu system run time"
```

```
Start untimed simulation with local date and time= Wed Sep 21 21:35:06 2022

Timed simulation starting…
End untimed simulation with local date and time= Wed Sep 21 21:35:49 2022


Info: /OSCI/SystemC: Simulation stopped by user.
Emu system run time 0.00113 sec==1133424000 ps
```

1.7 SAXPY with Replicated Data Structures

Finally, we look at using replication to create copies of the constant variable, `a` across all the nodes. This prevents migrations to access this common variable if it were located only on a single node. Note that replication can be a powerful tool for optimized allocation but it should be used primarily with small data structures and variables that are read-only (for coherency reasons).

[15]:
```
Code('saxpy-1d-replicated.c')
```

[15]:
```
#include <stdlib.h>
#include <cilk/cilk.h>
#include <memoryweb/memoryweb.h>
#include <memoryweb/timing.h>

long aval;
replicated long a;

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
  aval = atol(argv[3]); // constant
  mw_replicated_init(&a, aval);
  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

  lu_profile_perfcntr(PFC_CLEAR, "CLEAR COUNTERS");
  lu_profile_perfcntr(PFC_START, "START COUNTERS");

  for (long i = 0, j = 0; i < nth; i++, j += grain)
    cilk_spawn saxpy(grain, a, &x[j], &y[j]);
  cilk_sync;

  lu_profile_perfcntr(PFC_STOP, "STOP COUNTERS");
}
```

[16]:

```
%%bash
emu-cc $CCFLAGS -o saxpy-1d-replicated.mwx  saxpy-1d-replicated.c
emusim.x $SIMFLAGS -- saxpy-1d-replicated.mwx 8 128 5.0 2>/dev/null
less saxpy-1d-replicated.cdc | grep "Emu system run time"

#%%bash
#FLAGS="-I/tools/lucata/pathfinder-sw/22.02/include/memoryweb/ -L/tools/lucata/
↪pathfinder-sw/22.02/lib -lmemoryweb"
#emu-cc -o saxpy-1d-replicated.mwx $FLAGS saxpy-1d-replicated.c
#emusim.x -m 24 --total_nodes 1 -- saxpy-1d-replicated.mwx 8 128 5 2>/dev/null
```

```
Start untimed simulation with local date and time= Wed Sep 21 21:35:55 2022

Timed simulation starting…
End untimed simulation with local date and time= Wed Sep 21 21:36:02 2022


Info: /OSCI/SystemC: Simulation stopped by user.
Emu system run time 0.000194 sec==194460000 ps
```

1.8 Visualization

As we learned in Notebook 1.1, we can use `emusim_profile` to generate a number of charts to help us understand the performance of the profiled regions. Below is an example of how to do that for the last example, `saxpy-1d-replicated`.

[17]:

```
%%bash
emusim_profile saxpy-profile $SIMFLAGS -- saxpy-1d-replicated.mwx 8 128 5.0 2>/
↪dev/null
```

```
Generating profile in saxpy-profile/saxpy-1d-replicated
emusim.x  -m 24 --total_nodes 4
saxpy-1d-replicated.mwx 8 128 5.0
Start untimed simulation with local date and time= Wed Sep 21 21:36:03 2022

Timed simulation starting…
End untimed simulation with local date and time= Wed Sep 21 21:36:10 2022


Info: /OSCI/SystemC: Simulation stopped by user.
Generating saxpy-profile/saxpy-1d-replicated_total_instructions.png
Generating saxpy-profile/saxpy-1d-replicated_total_migrations.png
Generating saxpy-profile/saxpy-1d-replicated.Thread_Enqueue_Map.png
Generating saxpy-profile/saxpy-1d-replicated.Memory_Read_Map.png
Generating saxpy-profile/saxpy-1d-replicated.Memory_Write_Map.png
Generating saxpy-profile/saxpy-1d-replicated.Atomic_Transaction_Map.png
Generating saxpy-profile/saxpy-1d-replicated.Remote_Transaction_Map.png
Generating saxpy-profile/saxpy-1d-replicated.MSP_Activity.png
Generating saxpy-profile/saxpy-1d-replicated.SRIO_Outgoing_Activity.png
Generating saxpy-profile/saxpy-1d-replicated.SRIO_Incoming_Activity.png
Generating saxpy-profile/saxpy-1d-replicated.Live_Threads.png
saxpy-profile/saxpy-1d-replicated.hpc exists
Find all graphs in: saxpy-profile/saxpy-1d-replicated_21-09-2022_21:36:31
The last hpc call to analyze will be 0
Program called lu_profile_perfcntr with message: STOP COUNTERS
Generating Graphs for [STOP COUNTERS]…
Stopping here after read 0
hpc_file_name_base: saxpy-1d-replicated.hpc
Report written to saxpy-profile/saxpy-1d-replicated-report.html, you may open it
in your browser now
```

The output for the above simulation is summarized in the file saxpy-profile/saxpy-1d-replicated.html. Note that some plots will not be generated when running with fewer than 2 nodes (set in `$SIMFLAGS`).

1.8.1 Postcript

Here we have investigated several different strategies for spawning threads and allocatin data with the Pathfinder's distributed layout.

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.

[18]:
```
#!make clean
```