Tools

John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center

Copyright 2017
Given that the philosophy of these workshops is to teach portable programming techniques and avoid platform or version specific information, I have always reluctantly avoided talking about tools. The best ones are licensed, and in parallel programming there are no open source debuggers worth speaking of.

However, we always get feedback requesting this, and there are two types of tools that are wildly useful in this field:

- Debuggers
- Profilers

So we will carve out a few minutes to expose you to two of the best examples.
A good debugger is an invaluable time saver. It is shameful how under-exploited they are in general serial programming. In parallel programming this is no less true. However there is one excuse: there are only two decent parallel debuggers (Totalview and DDT) and they are both proprietary.

They are very easy to use
Just compiler with the \(-\text{g}\) option and your executable will include everything needed to interact with the debugger at runtime. Note that this does impact performance substantially, so you wouldn’t want to do this all the time.

DDT allows us to skip the X Windows complication
Most unix-land tools with nice GUIs require an X server driven display. DDT now has a desktop client that avoids what have been tricky configuration and maddening performance issues.
This is what makes it a parallel db.
for (i = 0; i <= BORDS1; i++) {
    for (j = 0; j <= (COLUMNS1+1); j++) {
        Temperature_last[i][j] = 0.0;
    }
}

// Boundary condition endpoints
kMin = (my_PF_num==1)*100.0/npes;
kMax = (my_PF_num==n)*100.0/npes;

// Left and right boundaries
for (i = 0; i < BORDS1; i++) {
    Temperature_last[i][0] = kMin + (kMax-kMin)/BORDS1*i;
}

// Top boundary (PE 0 only)
if (my_PF_num == 0) {
    for (i = 0; i < BORDS1+1; i++) {
        Temperature_last[0][i] = 0.0;
    }
}

// Bottom boundary (last PE only)
if (my_PF_num == n-1) {
    for (i = 0; i < BORDS1+1; i++) {
        Temperature_last[BORDS1][i] = (100.0/COLUMNS1) * i;
    }
}

// only called by last PE
void track_progress(int iteration) {
    int i;

    print("-------- Iteration number: ", iteration, "\n");

    // output global coordinates so user doesn't have to understand decomposition
    for (i = 0; i < BORDS1+1; i++) {
        print(P(b[i]), P(c[i]), P(bord[i]));
    }
    print("\n");
}
Other Nifty Features

- Can deal with very large numbers of PEs
  - Has ways of coalescing PEs into groups, etc.
  - Has ways of comparing and contrasting across PEs to find anomalies

- Can connect to running jobs

- Also great for OpenMP and CUDA/OpenACC

- Also good for memory debugging

- and MPI tracing
Profilers

Great Open Source Option
TAU

Intel
VTune & Trace Analyzer

Quickly Find Tuning Opportunities
See Results On The Source Code

Tune OpenMP Scalability
Visualize & Filter Data

- Hotspot (Statistical call tree), Call counts
- Thread Profiling -- Concurrency, Lock & Waits Analysis
- Cache miss, Bandwidth analysis...
- GPU Offload and OpenCL™ Kernel Tracing
- View Results on the Source / Assembly
- OpenMP Scalability Analysis
- Visualize Thread & Task Activity on the Timeline