OpenMP Booth Talk @ SC'17

STUDYING OPENMP WITH VAMPIR

Ronny Tschüter
Case Studies
Sparse Matrix Vector Multiplication – Load Imbalances
Sparse Matrix Vector Multiplication

\[
\begin{pmatrix}
  y_1 \\
  \vdots \\
  y_m
\end{pmatrix}
= 
\begin{pmatrix}
  a_{11} & \cdots & a_{n1} \\
  \vdots & \ddots & \vdots \\
  a_{m1} & \cdots & a_{mn}
\end{pmatrix}
\cdot
\begin{pmatrix}
  x_1 \\
  \vdots \\
  x_n
\end{pmatrix}
\]

- A sparse matrix is a matrix populated primarily with zeros
- Only non-zero elements of \(a_{ij}\) are saved efficiently in memory
- Algorithm

```c
foreach row r in A
  y[r.x] = 0
  foreach non-zero element e in row
    y[r.x] += e.value * x[e.y]
```
Sparse Matrix Vector Multiplication

- **Naive OpenMP algorithm**

```c
#pragma omp parallel for
define row r in A
    y[r.x] = 0
define non-zero element e in row
    y[r.x] += e.value * x[e.y]
```

- Distributes the rows of A evenly across the threads in the parallel region
- The distribution of the non-zero elements may influence the load balance in the parallel application
Sparse Matrix Vector Multiplication
Improved OpenMP algorithm

```c
#pragma omp parallel for schedule(dynamic,1000)
foreach row r in A
    y[r.x] = 0
    foreach non-zero element e in row
        y[r.x] += e.value * x[e.y]
```

- Distributes the rows of A dynamically across the threads in the parallel region
Sparse Matrix Vector Multiplication

November 15, 2017

Studying OpenMP with Vampir
Sparse Matrix Vector Multiplication

November 15, 2017  Studying OpenMP with Vampir
Case Studies

Heat Conduction – Propagation of Load Imbalances
Heat Conduction
Case Studies

Trinity RNA-Seq Assembler – Comparing Performance between Different Process Numbers
Trinity RNA-Seq Assembler – Comparing Performance between Different Process Numbers

- Analyzes and optimization of the RNA-Seq assembler Trinity [1]
- Trinity is a pipeline of up to 27 individual components invoked by a main perl script
- One main performance issue was the poor intra-node scaling of the GraphFromFasta module
- Intra-node parallelism (OpenMP) achieved a speed up of only 2.27 with a full 16-core node, the figure shows traces in comparison for 1, 2, 4, 8, and 16 threads
- The first part of GraphFromFasta increases nearly linearly with the number of OpenMP threads; there is practically no parallel speed up with more than two threads
- Time spent **stringstreams** increases from about 25s with one thread to 260s with 16 threads

Trinity RNA-Seq Assembler – Comparing Performance between Different Process Numbers

- Root cause was the frequent creation and destruction of string stream objects within an inner loop
- The creation was internally locked by a mutex, which produced excessive wait times
- Solution was to move the string stream object creation before the loop and only clear the string streams in the inner loop
- This resulted in better scaling (parallel speed up increased from 2.3 to 8.9) and reduced serial runtime (for the test data set from 72s to 45s)
- The introduced modifications resulted in a 22% improvement in overall run time
Case Studies

LSMS – Comparing Performance between Different Hardware
LSMS – Comparing Performance between Different Hardware

- OLCF ports applications from Titan to Summit (early development system called Summitdev)
- Summitdev contains NVIDIA P100 GPUs providing 4x the theoretical DPLOPS peak performance than the Tesla K20X in Titan
- One Summitdev node (20 cores) has four GPUs instead of one for Titan (16 cores). The system supports CUDA MPS, which allows sharing of GPUs between multiple processes
- This case study explores how these differences affect the performance of the CORAL benchmark code LSMS
- The faster GPU and better GPU/CPU pairing factor of 5 (20 cores/4 GPUs) cause the GPU-accelerated function `zblock_lu` to speed up on Summitdev, while the non-GPU-enabled function `buildKKRMatrix` gains in relative execution time

Overview of a 80-GPU LSMS run on Titan (80 nodes, white background) and Summitdev (20 nodes, blue background)

Detailed comparison of one iteration on Titan vs. roughly 2.5 on Summitdev
To evaluate if CUDA MPS can speed up LSMS, we run it with varying numbers of threads and processes per node.

LSMS is most resource efficient if the total number of threads and processes divides the number of simulated atoms evenly.

Using all 20 cores is faster than the other variants, although it adds occasional waiting time on the “left-over” threads.

The increase in MPI waiting time (more red in the green and cyan timelines) is negated by better GPU utilization.

GPU MPS uses the GPU more efficiently. But not using four cores per node negates this advantage.

Exploratory comparison of different process vs. thread setups.
- White: 4 processes times 4 threads per node (16 total) (1 process per GPU)
- Blue: 4 processes times 5 threads per node (20 total) (1 process per GPU)
- Green: 8 processes times 2 threads per node (16 total) (2 processes per GPU)
- Cyan: 16 processes times 1 thread per node (16 total) (4 processes per GPU)