OpenSHMEM Implementation of HPCG Benchmark

Ed D’Azevedo (dazevdof@ornl.gov)
Sarah Powers (powersss@ornl.gov)
Neena Imam (imamn@ornl.gov)

Oak Ridge National Laboratory

OpenSHMEM 2016
August 3, 2016
Background on HPCG 3.0

• High Performance Conjugate Gradient benchmark (with emphasis on **sparse** matrix computations) available at http://hpcg-benchmark.org

• HPCG was designed to be different from floating point intensive High Performance Linpack (HPL) TOP500 benchmark for LU factorization of **dense** matrices

• C++ code using OpenMP and two-sided MPI

• Solve elliptic partial differential equations (PDE) on $n_x \times n_y \times n_z$ processor grid, each processor has $L_x \times L_y \times L_z$ local grid
Background on HPCG (2)

- Symmetric positive linear system stored as unstructured sparse matrix in compressed sparse row format, interior nodes have 27 non-zeros
- Linear system solved by preconditioned conjugate gradient (CG) iterative method
- Multi-grid preconditioner with symmetric Gauss-Seidel smoother (sparse triangular solves)
- Communication kernels:
  - Sparse matrix vector multiplication performs boundary Halo exchange with immediate neighbor processors, requires data rearrangement in message buffers
  - MPI_Irecv + MPI_Send used in Halo exchange
  - Global reductions in computing dot products and determining convergence
Background on HPCG (3)

- Benchmark prints out major components:
  - **DDOT**: Global reduction in computing vector dot products
  - **SPMV**: Sparse matrix-vector multiplication used in iterative method, includes communication in boundary halo exchange (y(:) <- A * x(:))
  - **WAXPBY**: Vector operations (W(:) <- a*X(:) + b*Y(:))
  - **MG**: Multi-grid preconditioner including Gauss-Seidel smoothing (sparse triangular solve)
Domain Decomposition

Domain Decomposition - sub-domains & boundary values

Transmit the field values from processor to processor

Boundary values from neighbor sub-domain (processor)

Solve for field value at interior points in each sub-domain
27-point Stencil Operator

27-point stencil operator
Details of One-sided Implementation

- Initialization phase precomputes sizes and offsets into message buffers
- Buffers allocated in shared heap using `shmalloc()` (or `MPI_Alloc_mem()`)
- `shmem_double_put()` (or `MPI_Put()`) used instead of `MPI_Send()`
- SHMEM reduction (e.g. `shmem_double_sum_all()`) instead of MPI reduction
- `MPI_Win_create`+`MPI_Win_fence`+`MPI_Win_free` used in MPI one-sided, data transfer using `MPI_Put()`
Code for Two-sided MPI

```c
for (int i = 0; i < num_neighbors; i++) {
    local_int_t n_recv = receiveLength[i];
    MPI_Recv(x_external, n_recv, MPI_DOUBLE, neighbors[i],
             MPI_MY_TAG, MPI_COMM_WORLD, request+i);
    x_external += n_recv;
}

// Fill up send buffer
for (local_int_t i=0; i<totalToBeSent; i++) {
    sendBuffer[i] = xv[elementsToSend[i]];
}

for (int i = 0; i < num_neighbors; i++) {
    local_int_t n_send = sendLength[i];
    MPI_Send(sendBuffer, n_send, MPI_DOUBLE, neighbors[i],
             MPI_MY_TAG, MPI_COMM_WORLD);
    sendBuffer += n_send;
}

MPI_Status status;
for (int i = 0; i < num_neighbors; i++) {
    if ( MPI_Wait(request+i, &status) ) { std::exit(-1); }
}
```
Code for SHMEM Halo Exchange (1)

- Synchronize neighbor processors for halo exchange using atomic `shmem_int_inc()` and `shmem_int_wait_until()`

```c
for (int i=0; i < num_neighbors; i++) {
    int pe = neighbors[i];
    shmem_int_inc(&nreceivers_ready, pe);
}
shmem_fence();
shmem_quiet();

shmem_int_wait_until(&nreceivers_ready, SHMEM_CMP_EQ, num_neighbors);
shmem_int_swap(&nreceivers_ready, 0, shmem_my_pe());
```
Code for SHMEM Halo Exchange (2)

- Data transfer using `shmem_double_put()`

```c
for (int i = 0; i < num_neighbors; i++) {
    local_int_t n_send = sendLength[i];
    local_int_t offset = remoteOffset[i];
    int pe = neighbors[i];
    int nelem = n_send;
    double *src = sendBuffer;
    double *dest = &(recvBuffer[offest]);
    shmem_double_put(dest, src, nelem, pe);
    sendBuffer += n_send;
}
shmem_fence();
```
Code for MPI One-sided Halo Exchange

- Note MPI_Win_fence() has implicit synchronization

```c
status = MPI_Win_fence(0, win);
assert(status == MPI_SUCCESS);

for (int i = 0; i < num_neighbors; i++) {
    local_int_t n_send = sendLength[i];
    local_int_t offset = remoteOffset[i];
    status = MPI_Put(sendBuffer, n_send, MPI_DOUBLE,
                     neighbors[i], offset, n_send, MPI_DOUBLE, win);
    assert(status == MPI_SUCCESS);

    sendBuffer += n_send;
}

status = MPI_Win_fence(0, win);
assert(status == MPI_SUCCESS);
```
Numerical Experiments

- A 104 by 104 by 104 local grid was used in all cases
- OpenMP not activated
- HPCG writes out benchmark summary for times in
  - DDOT: dot product
  - WAXPBY: vector operations ($W(\cdot) \leftarrow a X(\cdot) + b Y(\cdot)$)
  - SpMV: sparse matrix times vector ($y(\cdot) \leftarrow A(\cdot,\cdot) * x(\cdot)$)
  - MG: multi-grid preconditioning
- Note time for DDOT is small compared to time in MG. DDOT may have implicit synchronization and include idle time from load imbalance
Results on SGI Turing cluster

- The SGI Turing cluster consists of 16 nodes, each node with two Intel Xeon E5-2660 with 10 cores (total 20 cores or 40 virtual cores with hyper-threading) and 128 GBytes of memory
- Mellanox InfiniBand Edge Switch, 36 QSFP ports, non-blocking capacity of 7.2Tbps, 16 Mellanox passive copper cable with 100Gb/s each
- SGI MPT 2.13 module was used. Note maximum of 640 MPI tasks using 40 MPI tasks per node (with hyper-threading) over 16 nodes
- Results suggest MPI-1, SHMEM, MPI-3 one-sided have similar performance
SGI Turing Cluster, 16 nodes
Cray XK7 (Titan)

• Total of 18,688 nodes, each node has 16 AMD cores, 32 GBytes of memory, 1 Nvidia K20X GPU (not used)

• Gemini interconnect, 3D torus network.

• Can run 1 to 16 MPI tasks per node.

• Only 15 (out of 16 max) MPI tasks on each node

• Cray-shmem 7.2.5

• MG is not communication intensive, difference less than 10%
Cray XK7 (Titan)

- **DDOT**
  - Time in seconds vs. Number of processors
  - Graphs show performance for MPI, MPI_onedside, and OSH

- **SPMV**
  - Time in seconds vs. Number of processors
  - Graphs show performance for MPI, MPI_onedside, and OSH

- **WAXPY**
  - Time in seconds vs. Number of processors
  - Graphs show performance for MPI, MPI_onedside, and OSH

- **MG**
  - Time in seconds vs. Number of processors
  - Graphs show performance for MPI, MPI_onedside, and OSH
**Cray XC30 (EOS)**

- EOS has 736 nodes, each node has Intel Xeon E5-2670 with two sockets, 8 cores each, (total 16 cores or **32 virtual cores** with hyper-threading) and 64 Gbytes of memory.

- Cray Aries interconnect with a network topology called Dragonfly, which has higher bandwidth and lower latency than Gemini.
EOS, 1 MPI task per node

- Run times of SPMV are similar for three implementations
- Run times of MG are similar for three implementations
EOS, 32 MPI tasks per node

- Run times of SPMV for MPI one-sided increase with higher number of processors.
- MG is not communication intensive.
- Run times of MPI one-sided in MG are less than 10% slower.
- Conjecture that a background progress thread might be affecting affinity and performance.
Summary

- Implemented SHMEM version of HPCG-3.0 benchmark for parallel sparse matrix computation
- Implemented MPI-3 one-sided version based on SHMEM version of HPCG
- Performed comparison of original MPI, SHMEM, MPI-3 one-sided versions on SGI Turing cluster, Cray XK7 (Titan), Cray XC30 (Eos)
- Results suggest all three versions have similar performance.
Acknowledgements

This work was supported by the United States Department of Defense (DoD) and used resources of the Computational Research and Development Programs at Oak Ridge National Laboratory.