A Preliminary Performance Comparison of Chapel to MPI and MPI/OpenMP

Laura Brown
US Army Corps of Engineers Engineer Research and Development Center, Vicksburg, MS
CHIUW 2015, Portland, OR
13 June 2015
Overview

- Background/Motivation
- Methodology
- Coding
- Results
- Conclusions
- Work Left To Do
- Acknowledgements
Background/Motivation

As the DoD HPC community works towards petascale and exascale computing, we face several challenges as users scale codes to larger core counts:

► Inefficient programming techniques
► Inefficient memory utilization
► Increased communication overhead

We are exploring Chapel, along with other HPCS and PGAS languages, to determine:

► Does this parallel language have the potential capability to perform more efficiently than (or at least as well as) MPI or MPI/OpenMP as core counts increase?
► Does this parallel language have the potential to be adopted by the HPCMP user community?
Methodology

- Translate a small, practical program into Chapel
  - Iterative Conjugate gradient using diagonal sparse matrix storage format
  - Originally written in Fortran
- Execute at 6 different processor counts…
  - 4 nodes/128 cores
  - 16 nodes/512 cores
  - 32 nodes/1024 cores
  - 64 nodes/2048 cores
  - 128 nodes/4096 cores
  - 256 nodes/8192 cores
- … with 4 different matrix problem sizes
  - 150 x 150
  - 1000 x 1000
  - 10000 x 10000
  - 20000 x 20000
- Compare results with observed performance of existing versions
  - Serial
  - MPI with 1-D decomposition
  - MPI/OpenMP with 1-D decomposition
Methodology

- All runs were performed on GARNET at ERDC DSRC in Vicksburg, MS
  - Cray XE6
  - 2 16-core 2.5 GHz AMD Interlagos chips per node (32 procs/node)
    - 2 GB memory per core (64 GB per node)
  - Gemini interconnect
  - Cray Compiler Environment
  - Chapel version 1.10.0
Coding

- MPI-style domain decomposition replaced with a single \texttt{dmapped} distributed array for each array of coefficients
  - All distributed arrays declared over the same domain
allocate (ac(n), ae(n), an(n), aw(n), as(n), rhs(n),
& u(1 - nx:n + nx), wksp(3*n + 2*nx), stat - istat)
ncal = (20*n + 4*nx)*nbyte
if (istat .eq. 0) then
  write (6, 110) nstore, nyl, istat
  call mpi_finalize (mpierf)
  stop
endif
  call mpi_reduce (nstore, maxmem, 1, mpi_integer,
& mpi_max, 0, comm, mpierr)

... set number of processors.

call omp_set_num_threads (nthreads)

c$omp parallel default (shared) private (i)
c$omp do
  do i = 1, n
    ac(i) = 4.0d0
    ae(i) = -1.0d0
    an(i) = -1.0d0
    aw(i) = -1.0d0
    as(i) = -1.0d0
  enddo
  c$omp do
    do i = nx, n, nx
      ac(i) = 0.0d0
    enddo
  c$omp end do nowait
  if (myid .eq. (nprocs - 1)) then
    c$omp do
      do i = n - nx + 1, n
        an(i) = 0.0d0
      enddo
    c$omp end do nowait
  endif
  c$omp do
    do i = 1, n - nx + 1, nx
      aw(i) = 0.0d0
    enddo
  c$omp end do nowait
  if (myid .eq. 0) then
    c$omp do
      do i = 1, nx
        ac(i) = 0.0d0
      enddo
    c$omp end do nowait
  endif
  c$omp do
    do i = 1 - nx, n + nx
      u(i) = 0.0d0
    enddo
  c$omp do
    do i = 1, n
      rhs(i) = ac(i) + ae(i) + an(i) + aw(i) + as(i)
    enddo
  c$omp end parallel

/* ac, ae, an, aw, as, rhs, u, wksp all dmapped */
const Problemspace: domain([1..n]) = [1..n];
const USpace = ProblemSpace.expand(nx);

/* INITIAL ARRAY ALLOCATION */
var p_err: [Problemspace] real = 0.0;
var ac: [Problemspace] real = 4.0;
var ae: [Problemspace] real = -1.0;
var an: [Problemspace] real = -1.0;
var aw: [Problemspace] real = -1.0;
var as: [Problemspace] real = -1.0;
var rhs: [Problemspace] real = 0.0;
var u: [USpace] real = 0.0;

/* SERIAL ARRAY INITIALIZATION PORTION - DIAGONAL SPARSE MATRIX*/
for i in nx..n by nx do ae[i] = 0.0;
for i in n-nx+1..n do an[i] = 0.0;
for i in 1..(n-nx+1) by nx do aw[i] = 0.0;
for i in 1..nx do as[i] = 0.0;

/* PARALLEL INITIALIZATION OF RHS */
rhs = ac + ae + an + aw + as;
Coding

- Concurrency handled via `forall` loops

```c
... Compute initial residual, residual norm, and rhs norm.

if (myid < nprocs - 1) then
    call mpi_sendrecv (u(n - nx + 1), nx, mpi_r, myid + 1, 0,
    & u(n + 1), nx, mpi_r, myid + 1, 1,
    & comm, stat, mpierr)
end if
if (myid > 0) then
    call mpi_sendrecv (u(1), nx, mpi_r, myid - 1, 1,
    & u(1 - nx), nx, mpi_r, myid - 1, 0,
    & comm, stat, mpierr)
end if
bsum = 0.000
gamma = 0.000
c$omp parallel default (shared) private (i)
c$omp do reduction (+, bsum, gamma)
    do i = 1, n
        r(i) = rhs(i) - ac(i)*u(i) - ae(i)*u(i+1) - an(i)*u(i+nx)
        & - aw(i)*u(i-1) - as(i)*u(i-nx)
        bsum = bsum + r(i)**2
        gamma = gamma + r(i)**2
    end do
c$omp end parallel
sendbuf(1) = bsum
sendbuf(2) = gamma
call mpi_allreduce (sendbuf, recvbuf, 2, mpi_r, mpi_sum,
& comm, mpierr)
bsum = recvbuf(1)
gamma = recvbuf(2)
bnorm = sqrt(bsum)
```

```c
forall i in ProblemSpace {
    r(i) = rhs(i) - ac(i)*u(i) - ae(i)*u(i+1) - an(i)*u(i+nx)
    & - aw(i)*u(i-1) - as(i)*u(i-nx);
    p_gamma(i) = r(i)**2;
    p_bsum(i) = rhs(i)**2;
}
gamma = 0.0;
gamma = + reduce p_gamma;
bsum = + reduce p_bsum;
bnorm = sqrt(bsum);
```
Results

- Size of each implementation
  - Serial: 394 LOC
  - MPI: 492 LOC
  - MPI/OpenMP: 525 LOC
  - Chapel: 256 LOC
Results

Chapel Performance at Increasing Node Counts

Time to Solution (sec)

Grid size

- CHAPEL @ 16 nodes
- CHAPEL @ 4 nodes
- SERIAL
Results

Chapel vs. MPI and MPI/OpenMP at 4 nodes

Time to Solution (sec)

Grid size

- **CHAPEL**
- **MPI**
- **MPI/OpenMP**
- **SERIAL**
Results

Chapel vs. MPI and MPI/OpenMP at 4 nodes (zoomed)
Results

Chapel vs. MPI and MPI/OpenMP at 16 nodes
Results

Chapel vs. MPI and MPI/OpenMP at 16 nodes (zoomed)
Results

What’s going on?

► Chapel does not scale well (for this code)!
  • Increasing the number of nodes increases the runtime, regardless of problem size
  • Increasing the problem size increases the runtime at an exponential rate

<table>
<thead>
<tr>
<th>NX</th>
<th>NY</th>
<th>RUNTIME (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>150</td>
<td>150</td>
<td>4.64201</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>183.006</td>
</tr>
<tr>
<td>10000</td>
<td>10000</td>
<td>43200+**</td>
</tr>
</tbody>
</table>

**Timed out after 12 hours
Results

What’s going on?

- Contacted Ben Harshbarger and Brad Chamberlain (Cray) in order to determine the issue
- Updating compiler versions (in this case, from 1.9.0 to 1.10.0) and adding compiler optimization flags positively impacted performance
- We determined that the main factor impacting performance was the use of Chapel’s implementation of reductions (can be used in manner equivalent to `mpi_allreduce`)
  - Example: `err = + reduce p_err;`
- There are 6 instances of `reduce` in this code. Two of them occur within the main iteration loop
  - The number of times this iteration loop executes increases as the problem size increases
    - 150x150: 314 iterations
    - 1000x1000: 1934 iterations
    - 10000x10000: 18133 iterations
    - 20000x20000: 35690 iterations
  - As problem size increases, runtime becomes dominated by these reductions
Conclusions, currently

- Chapel is not ready for our use in a production environment
  - Developers are working to modify/add features to make language more useful to average user, but they’re not there yet.
  - Documentation/tutorials can be an unorganized mixture of useful and outdated
    - Direct guidance from Chapel developers was extremely helpful, but not every user would have access to this
  - Even with code-tuning assistance, Chapel does not impress when compared to MPI and MPI/OpenMP
    - Does not seem to scale well with large problem sizes or large core counts
    - While Chapel itself is easy to read/use, will our code developers want to spend the effort learning/implementing a new programming language only to get similar or worse results than with MPI?
Conclusions, currently

- However…
  - As a language, Chapel is clean, concise and easy to understand (even after parallelization is implemented)
    - This could attract portions of our user base starting new coding projects
  - Once the performance improves, others looking to get gains from existing Fortran/C/C++ with MPI could follow
Work Left to Do

- Rerun tests with Chapel v.1.11.0
- When able, make changes necessary to Chapel code in order to increase performance
Acknowledgements

- DoD High Performance Computing Modernization Program (HPCMP)
  - This work was performed using computer time from the DoD HPCMP at the ERDC DoD Supercomputing Resource Center (DSRC).
- Ben Harshbarger and Brad Chamberlain (Cray) for Chapel support