Optimization & Scalability

Carlos Rosales
carlos@tacc.utexas.edu

September 20th, 2013
Parallel Computing in Stampede
What this talk is about

Highlight main performance and scalability bottlenecks

Simple but efficient optimization strategies
What this talk is about

Highlight main performance and scalability bottlenecks

Simple but efficient optimization strategies

• Serial code optimization
  – The optimization process
  – Compiler-based optimization
  – Manual optimization

• Multi-node optimization
  – Blocking vs non-blocking
  – Bi-directional comms
  – Eager limit
  – Collective comms

• On-node optimization
  – Resource contention
  – Load balance
SERIAL CODE OPTIMIZATION

Optimization and scalability
Optimization Process

1. **Profile Code**
2. **Identify Hotspots**
3. **Modify Code**
   - Hotspots Areas

   - **Enough Perform. Gain?**
     - yes
     - no (re-evaluate)

**TIME CONSTRAINED**

- Iterative process
- Application dependent
- Different levels
  - Compiler Options
  - Performance Libraries
  - Code Optimizations
Performance Analysis Basics

• Controlled measurements are essential
• Repeat
  – No single measurement is reliable
• Document
  – Save all these together: code version(s), compilers & flags, user inputs, system environment, timers & counters, code output
  – Record Details: important routines should track work done and time taken
• Automate
  – Don’t try to remember how you did something – script it!
• Control
  – Don’t let the O/S decide process & memory affinity – control it!
Compiler Option Based Optimization

- Significant improvements in execution time.
- Type `<compiler>` --help or `man <compiler>` for available options.
- There is **no universal flag set** optimal for all applications.
- Hardware vendors recommend sets of flags for optimal performance:
  - Intel Compiler optimization Guide
- Listings and reports suggest optimization opportunities.
Types of compiler Options

- Information
  - Assembly listing
  - Optimization reports

- Optimization level
  - Scalar replacement
  - Loop transformation
  - Prefetching
  - Cache blocking
  - Vectorization

- Architecture specification
  - Cache size
  - Pipeline length
  - Available streaming SIMD extensions

- Interprocedural Optimization
  - Inline
  - Reorder routines
Optimization level -On

- **-O0**: Disable optimization. Good for debugging.
- **-O1**: Optimize for speed. Disable some optimizations which increase code size.
- **-O2**: Optimize for speed. Default.
- **-O3**: Optimize for speed. Enable aggressive optimization. May alter results.
- **-fast**: Optimize for speed. Recommended optimizations from compiler vendor. Equivalent to `-O3 -xHOST -ipo -no-prec-div -static`. Not recommended because the -static option is incompatible with our MPI stacks.

The higher the level of optimization the longer the compilation stage takes, and the more memory required to do it.

The highest optimization levels (-O3 and -fast) do not always provide the fastest executable, and may change the semantics of the code.
Vectorization and SIMD instructions

- x87 instruction sets are now replaced by a variety of SIMD instruction sets
  - SSE is 128bit wide (2 doubles) / AVX is 256 wide (4 doubles) / MIC is 512 wide (8 doubles)
  - (S)SSE = (Supplemental) Streaming SIMD Extension
  - SSE instructions sets are chip dependent
  - (SSE instructions pipeline and simultaneously execute independent operations to get multiple results per clock period.)

- The –x<codes> { code = SSE4.2, AVX, ...} directs the compiler to use most advanced instruction set for the target hardware.
Vectorization

- Write loops with independent iterations, so that SSE/AVX instructions can be employed.
- SIMD (Single Instruction Multiple Data)
- SSE/AVX instructions operate on multiple data arguments simultaneously.
Performance Libraries

- Optimized for specific architecture
- Use when possible (can’t beat the performance)
- Numerical Recipes books **DO NOT** provide optimized code!
- Relevant to TACC:
  - Intel Math Kernel Libraries (MKL)
  - FFTW
  - PETSC
Best Practices

“premature optimization is the root of all evil” - Donald E. Knuth

• Clear & Simple (through structure and comments)
• Portable

• Access memory in the preferred order (row-major in C, column-major in Fortran)
• Minimize number of complicated mathematical operations
• Avoid excessive program modularization
  – Use macros
  – Write functions that can be inlined
• Avoid type casts and conversions
• Minimize the use of pointers
• Avoid the following within loops:
  – Constant expression evaluations
  – Branches, conditional statements
  – Function calls & IO operations
Code Optimization

– Improve memory usage
  • Calculation is often faster than memory access – recalculate!
  • Block loops to ensure maximum data reuse from L1 and L2
  • Allow compiler to unroll/fuse/fission loops
  • Minimize stride
    – Decreases access times
    – Unit stride is optimal in most cases
    – Power of 2 strides are typically bad

– Write loops with independent iterations
  • Can be unrolled and pipelined
  • Can be sent to vector or SIMD units
    – MMX, SSE, SSE2, SSE3, SSE4, AVX (AMD, Intel)
    – Vector Units (Cray, NEC)

– Reduce unnecessary IO
  • Disk access is expensive and disruptive
Stampede Processors
Intel Xeon E5 “Sandy Bridge” 2.7 GHz

Latency | 4 Cycles | 12 Cycles | 26-31 Cycles | ~175-350 Cycles
--- | --- | --- | --- | ---
Bandwidth | 40-100 GB/s | 30-60 GB/s | 20-40 GB/s | 4-10 GB/s

Peak FLOP rate is 8 64-bit FP ops/cycle
L2 latency = ~120 FLOPS
L3 latency = ~200 FLOPS
Memory latency = ~1500-3000 FLOPs

Streaming accesses are less expensive:
L1 DAXPY = ~33% of peak
L2 DAXPY = ~10% of peak
L3 DAXPY = <8% of peak
Local memory DAXPY = 2%-4% of peak
Minimizing Stride

The following snippets of code illustrate the correct way to access contiguous elements. i.e. stride 1, for a matrix in both C and Fortran.

**Fortran Example:**

```fortran
real*8 :: a(m,n), b(m,n), c(m,n)
...
do i=1,n
   do j=1,m
      a(j,i)=b(j,i)+c(j,i)
   end do
end do
```

**C Example:**

```c
double a[m][n], b[m][n], c[m][n];
...
for (i=0;i < m;i++){
   for (j=0;j < n;j++){
      a[i][j]=b[i][j]+c[i][j];
   }
}
```
Procedure Inlining

```
program MAIN
  integer :: ndim=2, niter=10000000
  real*8 :: x(ndim), x0(ndim), r
  integer :: i, j
  ...
  do i=1,100000
    ...
    r=dist(x,x0,ndim)
    ...
  end do
...
end program

real*8 function dist(x,x0,n)
  real*8 :: x0(n), x(n), r
  integer :: j,n
  r=0.0
  do j=1,n
    r=r+(x(j)-x0(j))**2
  end do
  dist=r
end function
```

```
program MAIN
  integer, parameter :: ndim=2
  real*8 :: x(ndim), x0(ndim), r
  integer :: i, j
  ...
  do i=1,100000
    ...
    r=0.0
    do j=1,ndim
      r=r+(x(j)-x0(j))**2
    end do
    ...
  end do
...
end program

function dist is expanded inline inside loop

Loop j is called niter times
```
Loop Unrolling

The objective of loop unrolling is to perform more operations per iteration in the loop. This optimization is typically best left to the compiler, but we will show you how it looks so you can identify it later.

\[
\begin{align*}
\text{do } & i=1, n \\
& A(i) = A(i) + B(i) * C \\
\text{end do}
\end{align*}
\]

\[
\begin{align*}
\text{do } & i=1, n, 4 \\
& A(i) = A(i) + B(i) * C \\
& A(i+1) = A(i+1) + B(i+1) * C \\
& A(i+2) = A(i+2) + B(i+2) * C \\
& A(i+3) = A(i+3) + B(i+3) * C \\
\text{end do}
\end{align*}
\]
Loop Fusion

Loop fusion combines two or more loops of the same iteration space (loop length) into a single loop:

```
for (i=0; i<n; i++) {
    a[i] = x[i] + y[i];
}
for (i=0; i<n; i++) {
    b[i] = 1.0 / x[i] + z[i];
}
```

Costly (multiple CP)

```
for (i=0; i<n; i++) {
    a[i] = x[i] + y[i];
    b[i] = 1.0 / x[i] + z[i];
}
```

Only n memory accesses for X array.
Five streams created.
Division many not be pipelined!
Array Blocking

- Works with small array blocks when expressions contain mixed-stride operations.
- Uses complete cache lines when they are brought in from memory.
- Avoids possible eviction that would otherwise ensue without blocking.

```
  do i=1,n
    do j=1,n
      A(j,i)=B(i,j)
    end do
  end do

  do i=1,n,2
    do j=1,n,2
      A(j,i)=B(i,j)
      A(j+1,i)=B(i+1,j)
      A(j,i+1)=B(i,j+1)
      A(j+1,i+1)=B(i+1,j+1)
    end do
  end do
```
Array Blocking

```
real*8 a(n,n), b(n,n), c(n,n)
do ii=1,n,nb
  do jj=1,n,nb
    do kk=1,n,nb
      do i=ii,min(n,ii+nb-1)
        do j=jj,min(n,jj+nb-1)
          do k=kk,min(n,kk+nb-1)
            c(i,j)=c(i,j)+a(j,k)*b(k,i)
          end do; end do; end do; end do; end do; end do
```

Much more efficient implementations exist, in HPC scientific libraries (ATLAS, MKL, ACML, ESSL ...).
Optimization and Scalability

ON-NODE OPTIMIZATION
A Prototypical “Node”

Each link has its own bandwidth and latency characteristics.
Basic Scalability Issues

• Resource contention effects
  – Shared L3 cache
  – Shared DRAM access
  – Prefetcher

• PCIe bandwidth bottleneck
  – Synchronization of co-processor and host creates hot spot
  – Can be minimized using asynchronous execution
  – Can be minimized minimizing data exchange to co-processor

• Best single-thread performance may not give best on-node throughput
  – Shared BW (recalculate instead of accessing memory)
  – Too many array references (split loops, reduce number of simultaneous memory streams)
Additional Scalability Issues

• Serial sections in the code
  – Startup and shutdown
  – Data dependencies preventing parallelization

• Imperfect load balance
  – Uneven static decomposition
  – Changes in local workload during run
  – Different data layouts slow down some tasks/threads

• Imperfect process / memory affinity
  – Processes may move between chips
  – Memory may be accessed remotely

• Communication costs
  – Explicit (MPI, OMP Barriers)
  – Implicit (OMP)
Setting Processor/Memory Affinity

- Tools to prevent, limit, or control process migration
  - `numactl` and `taskset` are command-line utilities
    - `tacc_affinity` works with `ibrun` on TACC systems
  - `sched_setaffinity()` can be called in-line in pthreads or OpenMP code
  - Intel compiler uses `KMP_AFFINITY` environment variable
  - MVAPICH2 uses `MV2_CPU_BINDING_POLICY` environment variable
  - Don’t attempt to mix process affinity control mechanisms!

- On extreme cases setting affinity can improve execution time by a factor 2X

- Preventing process migration also makes hardware performance counters much more useful
KMP_AFFINITY

compact

balanced

scatter
MULTI-NODE OPTIMIZATION

Optimization and Scalability
Testing Scalability

• How many CPUs can I use **efficiently**?
  – Don’t waste your own resources!
  – Solve larger or more complex problems

• Three typical tests:
  – Strong Scaling: performance with fixed problem size, adding nodes
  – Fixed Time to Solution: add nodes, increase problem size
  – Weak Scaling: fixed “work” per node, add nodes

• Tools available:
  – MPI or other inline timers: source code changes, low-overhead
  – mpiP, IPM: summary MPI information, simple to use, low-overhead
  – prof/gprof: summary function information, simple to use
  – HPCToolkit, Scalasca, Tau: detailed information, complex to use, can have high overheads
Communication Bottlenecks

- Bandwidth limited communication
  - Large amounts of data exchanged
  - Algorithm is not data-local (typical FT)
  - Change algorithm or try hybrid OMP/MPI

- Latency limited communication
  - Large number of small messages exchanged
  - Message setup overhead dominates communication
  - Pack data to send larger, less frequent messages

- Others
  - Unbalanced workload
  - Excessive IO
Strong Scaling Example

Parallel efficiency reduced to 60%

Comms overtake serial work
Blocking vs non-blocking

- Blocking messages prevent task from proceeding until message transmission is complete
- Non-blocking messages allow task to do other work while message transmission completes
- Using non-blocking messages allows overlap of communication and work
- Work/communication overlap hides communication latencies
- Requires compilation with asynchronous mode active

**Blocking**

Task is not available in between calls

**Non-Blocking**

Task can process additional work in between posting the calls and their
Overlapping Work and Communication

- Use non-blocking or one-sided exchanges
- Break down computational kernels
- Interleave partial kernels with communication

Time saved by overlapping
Bi-direction Example

Ghost Cell Exchange
Bi-direction Example

if (irank == left) then
  call mpi_send(a,N,MPI_REAL8,1,1,ICOMM,ierr)
  call mpi_recv(b,N,MPI_REAL8,1,0,ICOMM,istatus,ierr)
else if (irank == right) then
  call mpi_recv(b,N,MPI_REAL8,0,1,ICOMM,ierr)
  call mpi_send(a,N,MPI_REAL8,0,0,ICOMM,istatus,ierr)
endif

if (irank == left) then
  call mpi_isend(a,N,MPI_REAL8,1,1,ICOMM,ireqall(1),ierr)
  call mpi_irecv(b,N,MPI_REAL8,1,0,ICOMM,ireqall(2),ierr)
else if (irank == right) then
  call mpi_irecv(b,N,MPI_REAL8,0,1,ICOMM,ireqall(1),ierr)
  call mpi_isend(a,N,MPI_REAL8,0,0,ICOMM,ireqall(2),ierr)
endif

call mpi_waitall(2,ireqall,istatall,ierr)
Bi-Directional Bandwidth

Uni-Directional Exchange

Bi-Directional Exchange

Stampede - MPI Bandwidth

Bandwidth (GB/s) vs Message Size (Bytes)
Message Matching Protocols

**Eager Protocol**

- Single exchange of envelope, followed by data
- Requires queue to handle unexpected messages
  - Requires memory for data and envelopes
  - Queue fits 1000s of messages
  - Problems for large messages
- Faster method as long as there is memory for the queue

**Rendezvous Protocol**

- Multiple exchanges of envelope
  - Initial send
  - Acknowledgment of readiness
  - Final send
- Does not require queue buffers
- Still requires memory for the envelopes
Eager/Rendezvous Costs

• Elements contributing to communication costs
  – Latency (lat)
  – Message size (msg)
  – Envelope size (env)
  – Bandwidth (bw)
  – Copying from a temporary buffer (copy)

<table>
<thead>
<tr>
<th>protocol</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eager (expected)</td>
<td>lat + (msg + env)/bw</td>
</tr>
<tr>
<td>Eager (unexpected)</td>
<td>lat + (msg + env)/bw + copy*msg</td>
</tr>
<tr>
<td>Rendezvous (any)</td>
<td>lat + (msg + env)/bw + 2*(lat + env/bw)</td>
</tr>
</tbody>
</table>
Tailoring Protocol Matching Limits

• Typical implementations
  – Eager for small message sizes
  – Rendezvous for large sizes

• Control the switch between protocols (Mvapich2) with
  – MV2_SMP_EAGERSIZE
  – MV2_IBA_EAGER_THRESHOLD
  – MV2_VBUF_TOTAL_SIZE

• Rising the Eager Limit increases
  – MPI memory requirements
  – Bandwidth for medium size payloads
Collective Operation Types

- **broadcast**
  - Data
  - Tasks
  - A0
  - A1
  - A2
  - A3
  - Root to all tasks

- **gather/scatter**
  - A0
  - A1
  - A2
  - A3
  - All tasks to root

- **allgather**
  - A0
  - B0
  - C0
  - D0
  - All tasks to all tasks

- **alltoall**
  - A0
  - A1
  - A2
  - A3
  - B0
  - B1
  - B2
  - B3
  - C0
  - C1
  - C2
  - C3
  - D0
  - D1
  - D2
  - D3
Collective Communications

- Data exchange/reduction between multiple tasks in a group
- Can be achieved with P2P exchanges....
- But it can get messy to do it efficiently
- Do not scale well with the number of tasks
- Increasingly expensive with operation complexity
- MPI provides implementations for the most common collective operations
- Should be avoided when possible!
Communicators and Groups

• Often collective comms are not required across the complete task set

• A way to reduce the impact of collective communications is to restrict the number of tasks participating
  – Create a group with a subset of tasks that must use collective comms
  – Create a communicator associated to this group
  – Use the new communicator for all collectives restricted to this group

• Reducing the size of the group is critical because of the relatively poor scalability of collective communications
Communicators and Groups

1. Extract handles of global group from MPI_COMM_WORLD using MPI_COMM_GROUP.
2. Form new group as a subset of global group using MPI_COMM_INCL.
3. Create new communicator for new group using MPI_COMM_CREATE.
4. Determine new rank in new communicator using MPI_COMM_RANK.
5. Conduct communications using any MPI message passing routines.
6. When finished, free up new communicator and group (optional) using MPI_COMM_FREE and MPI_GROUP_FREE.
Summary

• Document, Automate, Control, Repeat

• We have defined multi-node scalability so that performance limiters are due to either load imbalance or communication costs
  • Remember that file system I/O is a form of communication

• Computation is often cheaper than communication
  • Optimum trade-off often varies with scale for the same algorithm!

• Look for most common problems first:
  – File system I/O – either too much, too many, or too serial
  – Network latency – messages too short \(\Rightarrow\) aggregate
  – Network bandwidth – consider algorithmic changes
  – Unbalanced workload – monitor idle times & adjust work per task
A Note on Parallel IO

- Parallel IO is often a mayor obstacle in using data efficiently.
- Parallel IO challenges:
  - Multiple accesses to same file (contention)
  - Limited IO bandwidth
  - Sheer number of files at large scales
- Two main aspects to consider:
  - How many files (one, one per core?)
  - How many simultaneous readers/writers?
- Answers are mostly code-dependent, although very large scale codes usually employ
  - Multiple files with contents spanning several computational cores
  - Standard parallel IO libraries for compatibility (NetCDF, HDF5)
Future Directions?

• Hybrid codes may become a requirement at very large scales
  – Largest machines currently have $O(1)$ million cores
  – MPI message length may be very short if using one MPI task per core
• Nodes now support $O(10)$ to $>O(100)$ threads
  – One MPI task per node or chip increases average message length
  – Recall that network performance is low for short messages!
• On-node memory usage optimization may be critical at large scales
  – Less memory buffer space required with hybrid code
• MPI implementations may not function well at extreme scales
  – Collectives are especially difficult

• Need to cope with heterogeneity, HW/SW failures, extreme scale
  – But not today
Questions ?