Optimization & Scalability

Carlos Rosales
carlos@tacc.utexas.edu

February 7th, 2012
Introduction to Parallel Computing

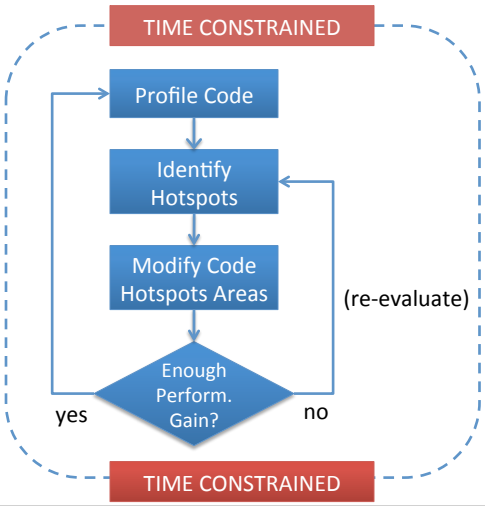
What this talk is about

- Simple but efficient optimization strategies
- Serial code optimization (Part I)
  - The optimization process
  - compiler-based optimization
  - manual optimization
- Parallel code optimization (Part II)
  - Strong and weak scaling
  - MPI performance as a function of the message size
  - Eager limit

TACC
THE UNIVERSITY OF TEXAS AT AUSTIN
TEXAS ADVANCED COMPUTING CENTER

Overview

Optimization Process



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

TACC
THE UNIVERSITY OF TEXAS AT AUSTIN
TEXAS ADVANCED COMPUTING CENTER

Compiler Option Based Optimization

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

Types of compiler Options

- Information
  - Assembly listing
  - Optimization Reports (Intel/PGI)
- Optimization level
  - Scalar replacement
  - Loop transformation
  - Prefetching
  - Cache Blocking
  - Vectorization
- Architecture specification
  - Cache size
  - Pipeline length
  - Available Streaming SIMD Extensions (SSE)
- Interprocedural Optimization
  - Inline
  - Reorder routines

Useful compiler options

GNU (gcc/gfortran)	Intel (icc/ifort)	PGI (pgcc/pgf90)	Description
(Not available)	-opt-report	-Minfo=ipo	Optimization Report
-S	-S	-S	Assembly Output
-O3	-O3	-fast	Aggressive Optimization
-mtune=barcelona -march=barcelona	-xW	-tp barcelona-64	Architecture Specific Optimization
(Partially included in -O3)	-ipo	-Mipa=fast,inline	Inter Procedural Optimization

```
icc  -O3   -xW                  -ipo                 prog.c
pgcc -fast -tp barcelona-64     -Mipa=fast,inline    prog.c
gcc  -O3   -mtune=barcelona -march=barcelona         prog.c
```

Optimization level Architecture Specification Interprocedural Optimization

Useful compiler options (TACC)

```
Intel 10.1
Ranger   : icc -O3 -xW      -ipo prog.c
Longhorn: icc -O3 -xT      -ipo prog.c

Intel 11.1 / 12.1
Ranger   : icc -O3 -xSSE2   -ipo prog.c
Longhorn: icc -O3 -xSSE4.2 -ipo prog.c
Lonestar: icc -O3 -xSSE4.2 -ipo prog.c
Auto     : icc -O3 -xHOST   -ipo prog.c

NVIDIA nvcc
Longhorn: nvcc -Xptxas -O3 -arch=sm_13 prog.cu
Lonestar: nvcc -Xptxas -O3 -arch=sm_23 prog.cu

GNU gcc
Auto     : gcc -O3 -mtune=native -march=native prog.c
```

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.

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 SSE instructions

- x87 instruction sets are now replaced by SSE "Vector" instruction sets, and soon by AVX.
- SSE is 128bit wide (2 doubles) / AVX is 256 wide (4 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 = W, P, T, O, S} directs the compiler to use most advanced SSE instruction set for the target hardware.

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:
  - AMD Core Math Libraries (ACML)
  - Intel Math Kernel Libraries (MKL)
  - GotoBLAS (highly optimized BLAS, by Kazsushige Goto)

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

Optimization Methods

: focus on serial code optimization

Code Optimization

- Exploit the underlying hardware:
  - Minimize stride
    - Decreases cache access times
    - Unit stride is optimal in most cases
    - Power of 2 strides are 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)

TACC
THE UNIVERSITY OF TEXAS AT AUSTIN
TEXAS ADVANCED COMPUTING CENTER

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:

```
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:

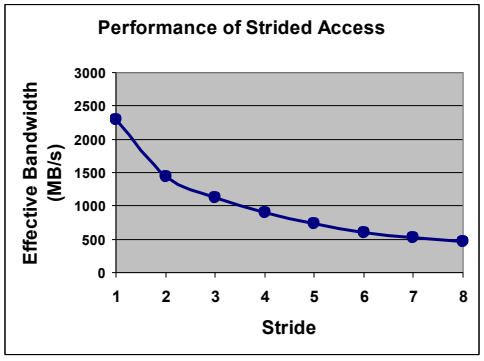
```
double a[m][n], b[m][n], c[m][n];
...
for (i=0;i < m;i++){
   for (j=0;j < n;j++){
      a[i][j]=b[i][j]+c[i][j];
   }
}
```


Minimizing Stride

Also, for large and small arrays, always try to arrange data so that structures are arrays with a unit (1) stride.

Bandwidth Performance Code:

```
do i = 1,10000000,istride
sum = sum + data( i )
end do
```



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
```

function dist is called *niter* times

```
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.

```
do i=1,n
   A(i)=A(i) + B(i)*C
end do
```

```
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
end do
```

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];
}
```

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

Costly (at least 30 CP)

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
```

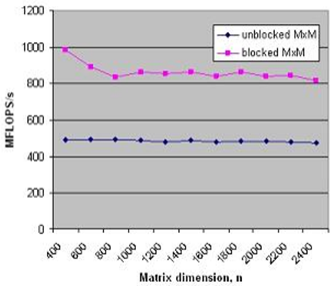
→

```
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
```

TACC
THE UNIVERSITY OF TEXAS AT AUSTIN
TEXAS ADVANCED COMPUTING CENTER

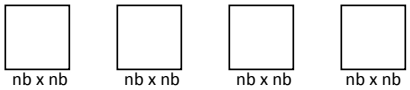
Array Blocking

matrix multiplication example




```
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)
```

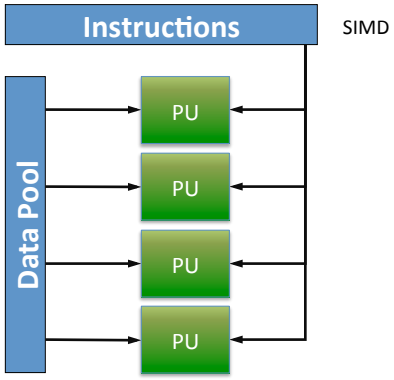



```
end do; end do; end do; end do; end do; end do
```

Much more efficient implementations exist, in HPC scientific libraries (ESSL, MKL, ACML,…).

Vectorization

- Write loops with independent iterations, so that SSE instructions can be employed
- SIMD (Single Instruction Multiple Data)
- SSE (Streaming SIMD Extensions) instructions operate on multiple data arguments simultaneously



TACC
THE UNIVERSITY OF TEXAS AT AUSTIN
TEXAS ADVANCED COMPUTING CENTER

Vectorization (cont)

In some cases, an entire loop can be replaced with a single call to a vector function:

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

→

```
vdInvSqrt(n,x,y);
```

```
for (i=0;i<n;i++) {
   y[i]=a*sin(x[i]) + b*cos(x[i]);
}
```

→

```
vdSinCos(n,x,s,c);
for (i=0;i<n;i++) {
   y[i]=a*s[i] + b*c[i];
}
```

But, how do you make something like this portable?
-- "ifdef", in C and F90.

#ifdef example

```
program main
integer, Parameter :: n=100, nn=2*n, nap=nn*(nn+1)/2
real(8), Parameter :: xmax=20.0, xmin=-xmax

#ifdef _IBM
                   integer            :: iopt=20
                   integer, parameter :: naux=3*nn
                   real(8):: ap(nap), eval(nn), work(naux)
#elif defined _IA32
                   integer            :: info
                   real(8) ::ap(nap), eval(nn), work(3*nn)
#endif
...
#ifdef _IA32
                   call DSPEV('n','u',nn,ap,eval,evec,nn,work,info)
#elif defined _IBM
                   call DSPEV(iopt,ap,eval,evec,nn,nn,work,naux)
#endif
...
end program
```

TACC
THE UNIVERSITY OF TEXAS AT AUSTIN
TEXAS ADVANCED COMPUTING CENTER

Scalability

: issues with parallel code optimization

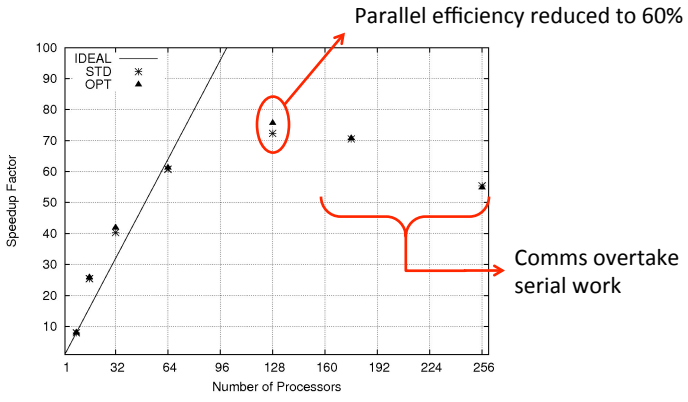
Testing Scalability

- How many CPUs can I use **efficiently**?
  - Don't waste your own resources!
  - Solve larger problems
- Two standard tests:
  - Speedup (more CPUs, fixed problem size) - Strong Scaling
  - Scaleup (problem size increases with CPU #) - Weak Scaling
- Tools available:
  - MPI timers (cumbersome)
  - mpiP, IMP (no detailed information, simple to use)
  - HPCToolkit, Tau (detailed information, complex to use)

Communication Bottlenecks

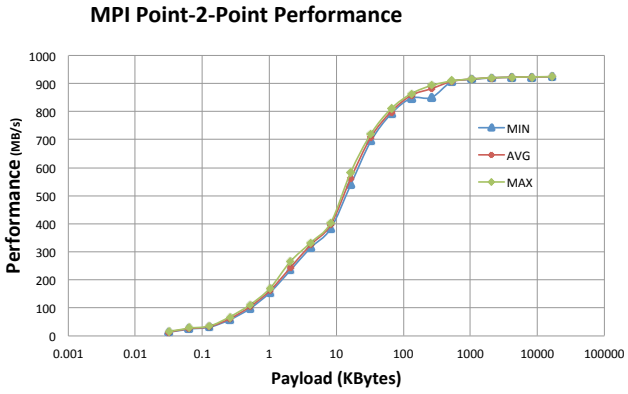
- Bandwidth limited communication
  - Large amounts of data exchanged
  - Algorithm is not data-local (typical FT)
  - Change algorithm or try hybrid OMP/MPI
- Latency limited communication
  - Large number of small messages exchanged
  - Message setup overhead dominates communication
  - Pack data to send larger, less frequent messages
- Others
  - Unbalanced workload
  - Excessive IO

Strong scaling example



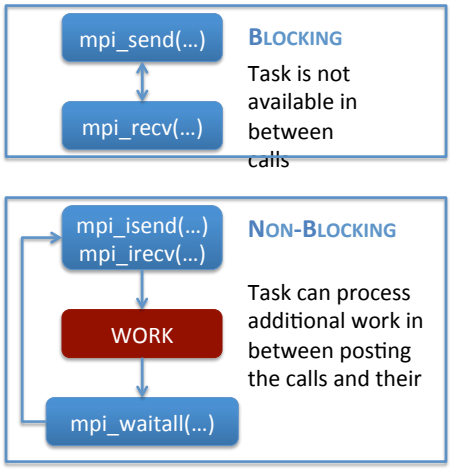
TACC
THE UNIVERSITY OF TEXAS AT AUSTIN
TEXAS ADVANCED COMPUTING CENTER

P-2-P Communication Performance
On-Chassis



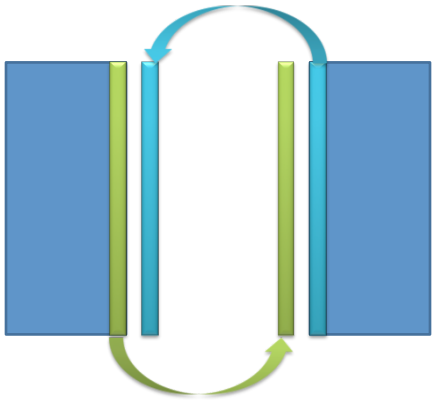
Blocking vs non-blocking

- Blocking messages prevent task to proceed 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



Bi-direction Example
Ghost Cell Exchange

- Non-blocking routines allow for bi-directional data exchange.
- Bi-directional exchange can double the available bandwidth



Bi-direction Example

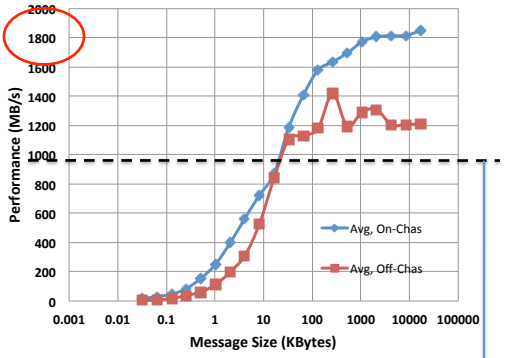
```
if(irank == left) then
   call mpi_send(a,N,MPI_REAL8,1,1,ICOMM,         ierr)
   call mpi_recv(b,N,MPI_REAL8,1,0,ICOMM,istatus,ierr)
else if (irank == right) then
   call mpi_recv(b,N,MPI_REAL8,0,1,ICOMM,         ierr)
   call mpi_send(a,N,MPI_REAL8,0,0,ICOMM,istatus,ierr)
endif
```

NOT Bi-directional

```
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-direction Communication Performance

- IB (InfiniBand) allows bi-directional messaging
- Use Async. Comms for ~1.6-1.9x speedup.



Maximum uni-directional bandwidth

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

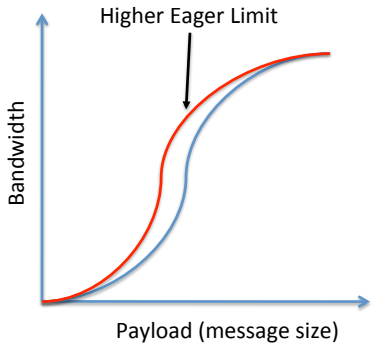
Eager/Rendezvous costs

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

protocol	cost
Eager (expected)	lat + (msg + env)/bw
Eager (unexpected)	lat + (msg + env)/bw + copy*msg
Rendezvous (any)	lat + (msg + env)/bw + 2*(lat + env/bw)

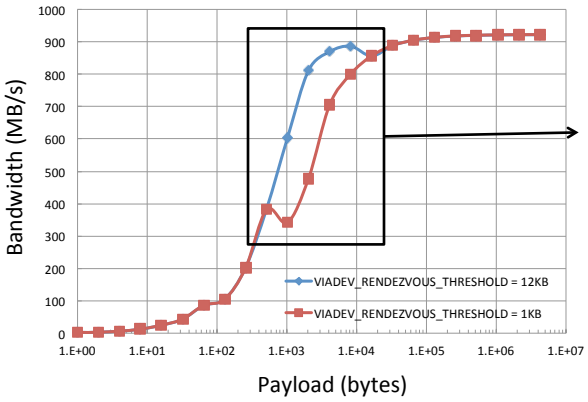
Tailoring protocol matching limits

- Typical implementations
  - Eager for small message sizes
  - Rendezvous for large sizes
- The switch from Eager to Rendezvous protocols can be controlled by `VIADEV_RENDEZVOUS_THRESHOLD`
- Increasing the Eager Limit
  - Increases MPI memory requirements
  - Increases bandwidth for medium size payloads



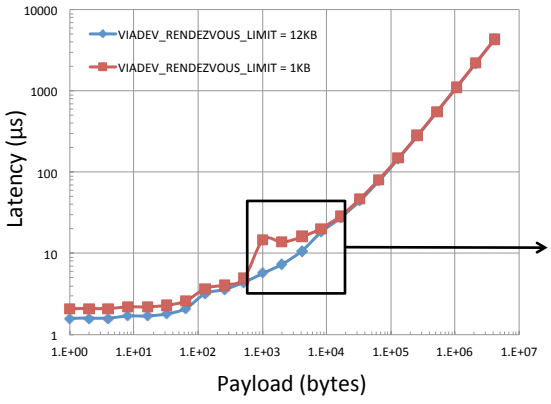
TACC
THE UNIVERSITY OF TEXAS AT AUSTIN
TEXAS ADVANCED COMPUTING CENTER

Eager limit effect on bandwidth



Bandwidth increases with increasing Eager limit

Eager limit effect on latency



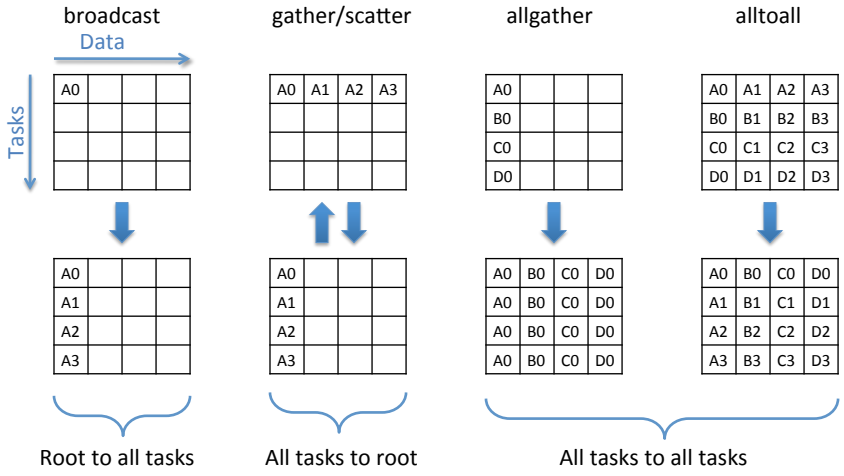
Latency decreases with increasing Eager limit

* Max value for the eager/rendezvous limit is equal to `VIADEV_VBUF_TOTAL_SIZE`

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
- Should be avoided when possible !
- MPI provides implementations for the most common collective operations
- MPI implementation does not use tags
- MPI implementation does not guarantee synchronization at the end of the call (use MPI_Barrier)

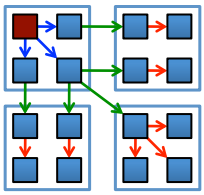
Collective Operation Types



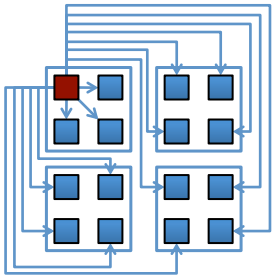
TACC
THE UNIVERSITY OF TEXAS AT AUSTIN
TEXAS ADVANCED COMPUTING CENTER

Collectives Implementation

Provided by MPI

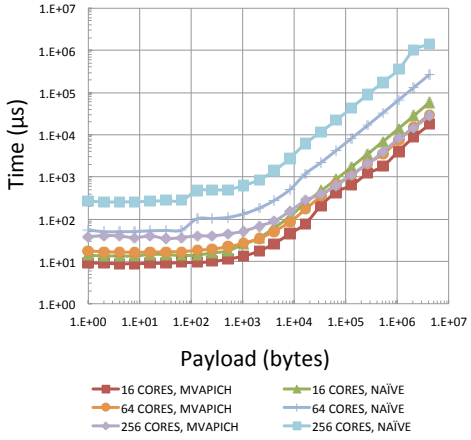


Naïve Implementation



Which one should you use?

Mvapich bcast vs naïve implementation



Speedup vs message size

core #	small	large
16	1.5	3.5
64	3.0	9.0
256	7.0	50.0

Scalability Summary

- Do not optimize a serial version heavily before parallelization
- Use data-local algorithms
- Identify and eliminate communication bottlenecks
  - Network bandwidth
  - Network latency
  - Excessive IO
  - Unbalanced workload