

Advanced Crash Course in Supercomputing: Parallelism



Rebecca Hartman-Baker
Oak Ridge National Laboratory
hartmanbakrj@ornl.gov

© 2004-2011 Rebecca Hartman-Baker. Reproduction permitted for non-commercial, educational use only.



Outline

- I. Parallelism
- II. Supercomputer Architecture
- III. Basic MPI
- IV. MPI Collectives
- V. Debugging and Performance Evaluation



I. PARALLELISM

Parallel Lines by Blondie. Source:
<http://xponentialmusic.org/blogs/885mmm/2007/10/09/403-blondie-hits-1-with-heart-of-glass/>

I. Parallelism

- Concepts of parallelization
- Serial vs. parallel
- Parallelization strategies

Parallelization Concepts

- When performing task, some subtasks depend on one another, while others do not
- Example: Preparing dinner
 - Salad prep independent of lasagna baking
 - Lasagna must be assembled before baking
- Likewise, in solving scientific problems, some tasks independent of one another

Serial vs. Parallel

- Serial: tasks must be performed in sequence
- Parallel: tasks can be performed independently in any order



Serial vs. Parallel: Example

- Example: Preparing dinner

- Serial tasks: making sauce, assembling lasagna, baking lasagna; washing lettuce, cutting vegetables, assembling salad



- Parallel tasks: making lasagna, making salad, setting table



7 OLCF ●●●●

OAK
CRIDGE
National Laboratory

Serial vs. Parallel: Example

- Could have several chefs, each performing one parallel task
- This is concept behind parallel computing



8 OLCF ●●●●

OAK
CRIDGE
National Laboratory

Parallel Algorithm Design: PCAM

- *Partition*: Decompose problem into fine-grained tasks to maximize potential parallelism
- *Communication*: Determine communication pattern among tasks
- *Agglomeration*: Combine into coarser-grained tasks, if necessary, to reduce communication requirements or other costs
- *Mapping*: Assign tasks to processors, subject to tradeoff between communication cost and concurrency

(taken from Heath: *Parallel Numerical Algorithms*)

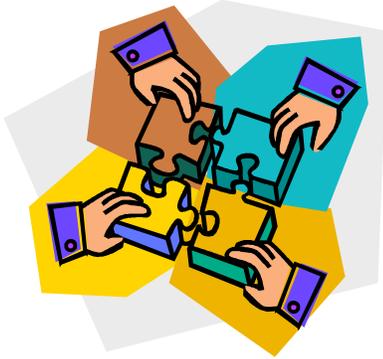
Discussion: Jigsaw Puzzle*

- Suppose we want to do 5000 piece jigsaw puzzle
- Time for one person to complete puzzle: n hours
- How can we decrease walltime to completion?



* Thanks to Henry Neeman

Discussion: Jigsaw Puzzle



- Add another person at the table
 - Effect on wall time
 - Communication
 - Resource contention
- Add p people at the table
 - Effect on wall time
 - Communication
 - Resource contention

11 OLCF ●●●●

OAK
BRIDGE
National Laboratory

Discussion: Jigsaw Puzzle



- What about: p people, p tables, $5000/p$ pieces each?
- What about: one person works on river, one works on sky, one works on mountain, etc.?

12 OLCF ●●●●

OAK
BRIDGE
National Laboratory

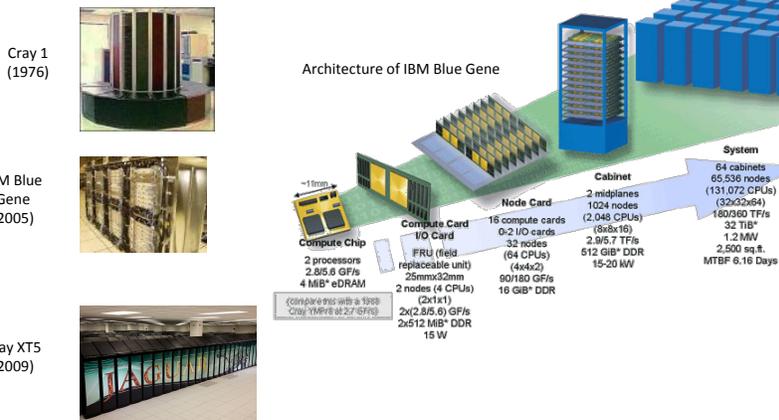


II. ARCHITECTURE

Image: Louvre Abu Dhabi – Abu Dhabi, UAE, designed by Jean Nouvel, from <http://www.inhabitat.com/2008/03/31/jean-nouvel-named-2008-pritzker-architecture-laureate/>

II. Supercomputer Architecture

- What is a supercomputer?
- Conceptual overview of architecture



Cray 1 (1976)



IBM Blue Gene (2005)



Cray XT5 (2009)



What Is a Supercomputer?

- “The biggest, fastest computer right this minute.” -- Henry Neeman
- Generally 100-10,000 times more powerful than PC
- This field of study known as *supercomputing*, *high-performance computing (HPC)*, or *scientific computing*
- Scientists use really big computers to solve really hard problems

SMP Architecture

- Massive memory, shared by multiple processors
- Any processor can work on any task, no matter its location in memory
- Ideal for parallelization of sums, loops, etc.

Cluster Architecture

- CPUs on racks, do computations (fast)
- Communicate through myrinet connections (slow)
- Want to write programs that divide computations evenly but minimize communication

State-of-the-Art Architectures

- Today, hybrid architectures gaining acceptance
- Multiple {quad, 8, 12}-core nodes, connected to other nodes by (slow) interconnect
- Cores in node share memory (like small SMP machines)
- Machine appears to follow cluster architecture (with multi-core nodes rather than single processors)
- To take advantage of all parallelism, use MPI (cluster) and OpenMP (SMP) hybrid programming



III. MPI

MPI also stands for Max Planck Institute for Psycholinguistics. Source: <http://www.mpi.nl/WhatWeDo/institute-pictures/building>

III. Basic MPI

- Introduction to MPI
- Parallel programming concepts
- The Six Necessary MPI Commands
- Example program

Introduction to MPI

- Stands for *Message Passing Interface*
- Industry standard for parallel programming (200+ page document)
- MPI implemented by many vendors; open source implementations available too
 - ChaMPion-PRO, IBM, HP, Cray vendor implementations
 - MPICH, LAM-MPI, OpenMPI (open source)
- MPI function library is used in writing C, C++, or Fortran programs in HPC
- MPI-1 vs. MPI-2: MPI-2 has additional advanced functionality and C++ bindings, but everything learned today applies to both standards

Parallelization Concepts

- Two primary programming paradigms:
 - SPMD (single program, multiple data)
 - MPMD (multiple programs, multiple data)
- MPI can be used for either paradigm

SPMD vs. MPMD

- SPMD: Write single program that will perform same operation on multiple sets of data
 - Multiple chefs baking many lasagnas
 - Rendering different frames of movie
- MPMD: Write different programs to perform different operations on multiple sets of data
 - Multiple chefs preparing four-course dinner
 - Rendering different parts of movie frame
- Can also write hybrid program in which some processes perform same task

The Six Necessary MPI Commands

- `int MPI_Init(int *argc, char **argv)`
- `int MPI_Finalize(void)`
- `int MPI_Comm_size(MPI_Comm comm, int *size)`
- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
- `int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
- `int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`

Initiation and Termination

- `MPI_Init(int *argc, char **argv)`
initiates MPI
 - Place in body of code after variable declarations and before any MPI commands
- `MPI_Finalize(void)` shuts down MPI
 - Place near end of code, after last MPI command

Environmental Inquiry

- `MPI_Comm_size(MPI_Comm comm, int *size)`
 - Find out number of processes
 - Allows flexibility in number of processes used in program
- `MPI_Comm_rank(MPI_Comm comm, int *rank)`
 - Find out identifier of current process
 - $0 \leq \text{rank} \leq \text{size}-1$

Message Passing: Send

- `MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
 - Send message of length `count` bytes and datatype `datatype` contained in `buf` with tag `tag` to process number `dest` in communicator `comm`
 - E.g. `MPI_Send(&x, 1, MPI_DOUBLE, manager, me, MPI_COMM_WORLD)`

Message Passing: Receive

- `MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
 - Receive message of length `count` bytes and datatype `datatype` with tag `tag` in buffer `buf` from process number `source` in communicator `comm` and record status `status`
 - E.g. `MPI_Recv(&x, 1, MPI_DOUBLE, source, source, MPI_COMM_WORLD, &status)`

Message Passing

- **WARNING!** Both standard send and receive functions are *blocking*
- `MPI_Recv` returns only after receive buffer contains requested message
- `MPI_Send` may or may not block until message received (usually blocks)
- Must watch out for deadlock

Deadlocking Example (Always)

```

#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
    int me, np, q, sendto;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    if (np%2==1) return 0;
    if (me%2==1) {sendto = me-1;}
    else {sendto = me+1;}
    MPI_Recv(&q, 1, MPI_INT, sendto, sendto, MPI_COMM_WORLD,
            &status);
    MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
    printf("Sent %d to proc %d, received %d from proc %d\n",
           me, sendto, q, sendto);
    MPI_Finalize();
    return 0;
}

```

Deadlocking Example (Sometimes)

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
    int me, np, q, sendto;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    if (np%2==1) return 0;
    if (me%2==1) {sendto = me-1;}
    else {sendto = me+1;}
    MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
    MPI_Recv(&q, 1, MPI_INT, sendto, sendto, MPI_COMM_WORLD,
            &status);
    printf("Sent %d to proc %d, received %d from proc %d\n",
           me, sendto, q, sendto);
    MPI_Finalize();
    return 0;
}
```

31 OLCF



Deadlocking Example (Safe)

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv) {
    int me, np, q, sendto;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    if (np%2==1) return 0;
    if (me%2==1) {sendto = me-1;}
    else {sendto = me+1;}
    if (me%2 == 0) {
        MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
        MPI_Recv(&q, 1, MPI_INT, sendto, sendto, MPI_COMM_WORLD,
                &status);
    } else {
        MPI_Recv(&q, 1, MPI_INT, sendto, sendto, MPI_COMM_WORLD,
                &status);
        MPI_Send(&me, 1, MPI_INT, sendto, me, MPI_COMM_WORLD);
    }
    printf("Sent %d to proc %d, received %d from proc %d\n", me,
           sendto, q, sendto);
    MPI_Finalize();
    return 0;
}
```

32 OLCF



Explanation: Always Deadlock Example

- Logically incorrect
- Deadlock caused by blocking `MPI_Recv`s
- All processes wait for corresponding `MPI_Send`s to begin, which never happens

Explanation: Sometimes Deadlock Example

- Logically correct
- Deadlock could be caused by `MPI_Send`s competing for buffer space
- Unsafe because depends on system resources
- Solutions:
 - Reorder sends and receives, like safe example, having evens send first and odds send second
 - Use non-blocking sends and receives or other advanced functions from MPI library (see MPI standard for details)



IV. MPI COLLECTIVES

"Collective Farm Harvest Festival" (1937) by Sergei Gerasimov. Source:
<http://max.mmlc.northwestern.edu/~mdenner/Drama/visualarts/neorealism/34harvest.html>

MPI Collectives

- Communication involving group of processes
- Collective operations
 - Broadcast
 - Gather
 - Scatter
 - Reduce
 - All-
 - Barrier

Broadcast

- Perhaps one message needs to be sent from manager to all worker processes
- Could send individual messages
- Instead, use broadcast – more efficient, faster
- `int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`

Gather

- All processes need to send same (similar) message to manager
- Could implement with each process calling `MPI_Send(...)` and manager looping through `MPI_Recv(...)`
- Instead, use gather operation – more efficient, faster
- Messages concatenated in rank order
- `int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
- Note: `recvcount` = number of items received from each process, not total

Gather

- Maybe some processes need to send longer messages than others
- Allow varying data count from each process with `MPI_Gatherv(...)`
- `int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm)`
- `recvcounts` is array; entry `i` in `displs` array specifies displacement relative to `recvbuf[0]` at which to place data from corresponding process number

Scatter

- Inverse of gather: split message into NP equal pieces, with `i`th segment sent to `i`th process in group
- `int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
- Send messages of varying sizes across processes in group: `MPI_Scatterv(...)`
- `int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs, MPI_datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`

Reduce

- Perhaps we need to do sum of many subsums owned by all processors
- Perhaps we need to find maximum value of variable across all processors
- Perform global reduce operation across all group members
- `int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`

Reduce: Predefined Operations

MPI_Op	Meaning	Allowed Types
MPI_MAX	Maximum	Integer, floating point
MPI_MIN	Minimum	Integer, floating point
MPI_SUM	Sum	Integer, floating point, complex
MPI_PROD	Product	Integer, floating point, complex
MPI_LAND	Logical and	Integer, logical
MPI_BAND	Bitwise and	Integer, logical
MPI_LOR	Logical or	Integer, logical
MPI BOR	Bitwise or	Integer, logical
MPI_LXOR	Logical xor	Integer, logical
MPI_BXOR	Bitwise xor	Integer, logical
MPI_MAXLOC	Maximum value and location	*
MPI_MINLOC	Minimum value and location	*

Reduce: Operations

- **MPI_MAXLOC** and **MPI_MINLOC**
 - Returns {max, min} and rank of first process with that value
 - Use with special MPI pair datatype arguments:
 - **MPI_FLOAT_INT** (float and int)
 - **MPI_DOUBLE_INT** (double and int)
 - **MPI_LONG_INT** (long and int)
 - **MPI_2INT** (pair of int)
 - See MPI standard for more details
- **User-defined operations**
 - Use **MPI_Op_create(...)** to create new operations
 - See MPI standard for more details

All- Operations

- Sometimes, may want to have result of gather, scatter, or reduce on all processes
- **Gather operations**
 - `int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`
 - `int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, MPI_Comm comm)`

All-to-All Scatter/Gather

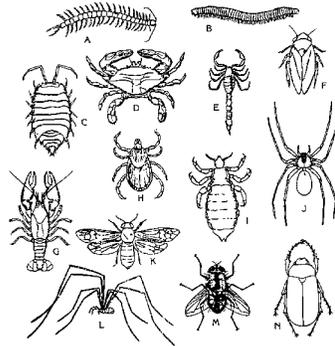
- Extension of `Allgather` in which each process sends distinct data to each receiver
- Block `j` from process `i` is received by process `j` into `i`th block of `recvbuf`
- `int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`
- Also corresponding `AlltoAllv` function available

All-Reduce

- Same as `MPI_Reduce` except result appears on all processes
- `int MPI_Allreduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`

Barrier

- In algorithm, may need to synchronize processes
- Barrier blocks until all group members have called it
- `int MPI_Barrier(MPI_Comm comm)`



Source: <http://www.uky.edu/Aq/Entomology/ythfacts/4h/unit1/f&tr.htm>

V. DEBUGGING AND PERFORMANCE EVALUATION

V. Debugging and Performance Evaluation

- Common errors in parallel programs
- Debugging tools
- Overview of benchmarking and performance measurements



Common Errors

- Program hangs
 - Send has no corresponding receive (or vice versa)
 - Send/receive pair do not match in source/recipient or tag
 - Condition you believe should occur does not occur
- Segmentation fault
 - Trying to access memory you are not allowed to access/ memory you should not have been allowed to access has been altered (e.g. array index out-of-bounds, uninitialized pointers, using non-pointer as pointer)
 - Trying to access a memory location in a way that is not allowed (e.g. overwrite a read-only location)

Debugging Tools

- Debugging parallel codes is particularly difficult
- Problem: figuring out what happens on each node
- Solutions:
 - Print statements, I/O redirection into files belonging to each node
 - Debuggers compatible with MPI

Print Statement Debugging Method

- Each processor dumps print statements to `stdout` or into individual output files, e.g. `log.0001`, `log.0002`, etc.
- Advantage: easy to implement, independent of platform or available resources
- Disadvantage: time-consuming, extraneous information in log files

MPI-Compatible Debuggers

- Commercial debuggers
 - Easy-to-use GUI, intuitive
 - Installed on production systems such as Crays, probably not installed on local machines
 - Examples: TotalView, Allinea DDT
- Free debuggers + `mpirun`
 - Use `mpirun` command and specify your favorite debugger, e.g.
`mpirun -dbg=ddd -np 4 ./myprog`
 - This option available with MPICH and most other MPI implementations
 - Not as “pretty” as commercial debugger but it gets job done

Why Use Debuggers?

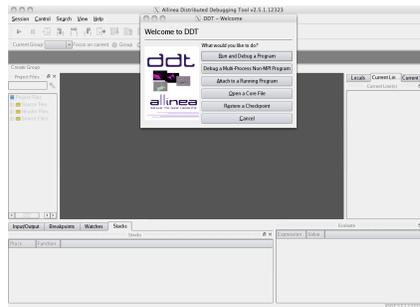
- Debuggers can save time
 - With print-statement debugging, must insert print statements into code, sift through print statements, and find error
 - Debugger allows you to find the line where problem occurs in a single trial
- Complexity of bugs grows with complexity of code
 - More lines of code, more potential for bugs
 - More complicated algorithm, more potential for errors
 - Parallelism only adds to complexity

Allinea DDT

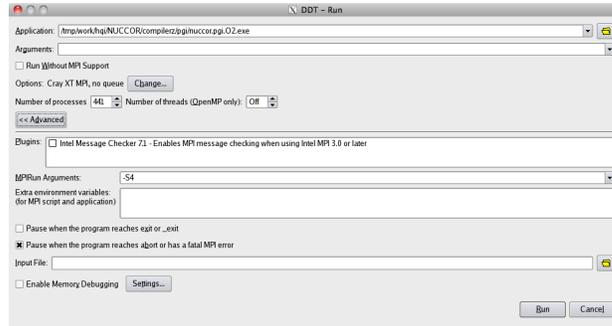
- Distributed Debugging Tool
- Capable of debugging codes written with MPI, OpenMP, threading, GPGPU (in beta)
- Allinea collaborating with ORNL to create petascale debugging tool
- Easy to use, intuitive

Using DDT

- Compile code with `-g` flag
- On OLCF systems:
 - `module load ddt`
 - `ddt &`
- Launch DDT from scratch directory
- Can run it within interactive job, or have DDT launch job



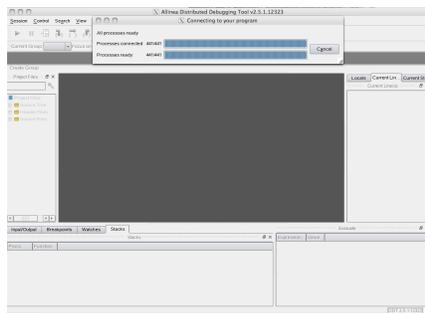
Using DDT



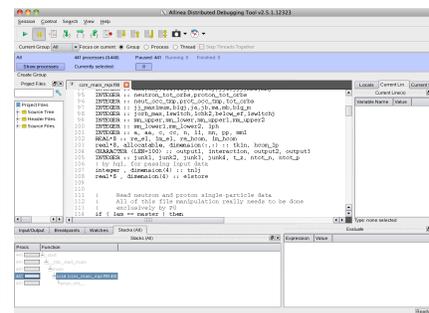
- Running a job
 - Enter application name
 - Can have DDT launch job, or run interactive job
 - Set arguments as necessary

Using DDT

DDT Starting up

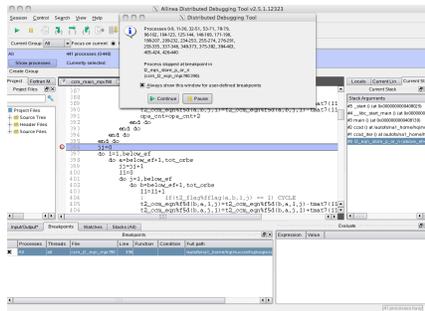


Opening Screen

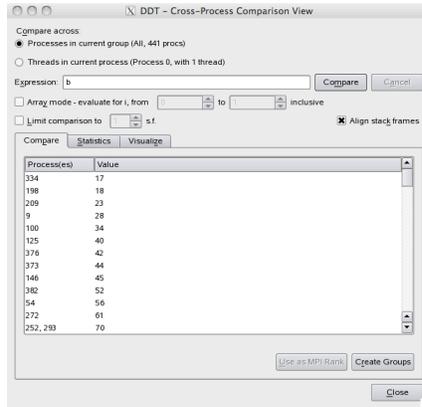


Using DDT

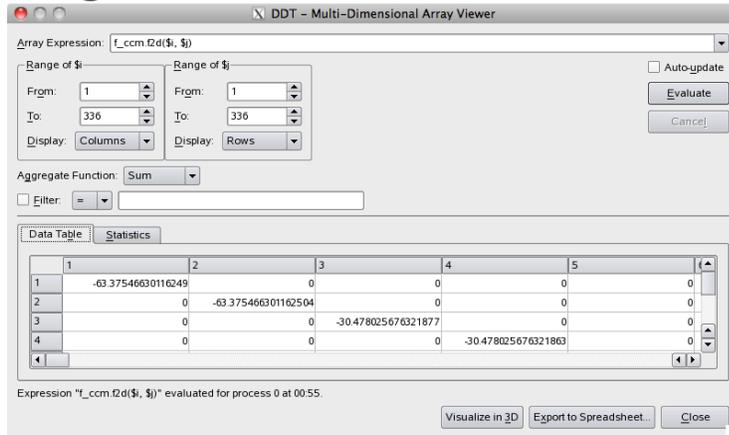
Insert Breakpoints, and Pause at that Point



Can view the value of scalar variable across all processes



Using DDT



- View array on single process
- Can also view statistics, visualize, evaluate subsets

Benchmarking and Performance

- Efficiency
- Scalability
- Performance modeling

Efficiency

- How well does parallel program perform compared to serial program (or parallel program on 1 processor)?

$$E_N = \frac{T_1}{NT_N}$$

- E = efficiency, N = # processors, T_p = time for p processors

Efficiency

- Ideally, $E_N = 1$; realistically, $E_N < 1$.
- Factors influencing efficiency
 - Load balance (evenly distribute work for better efficiency)
 - Concurrency (minimize idle time on all processors)
 - Overhead (minimize work that serial computation would not do, e.g. communication)

Scalability: Speedup

- How well does parallel program take advantage of additional processors?

$$S_N = \frac{T_1}{T_N}$$

- S = speedup, N = # processors, T_p = time for p processors

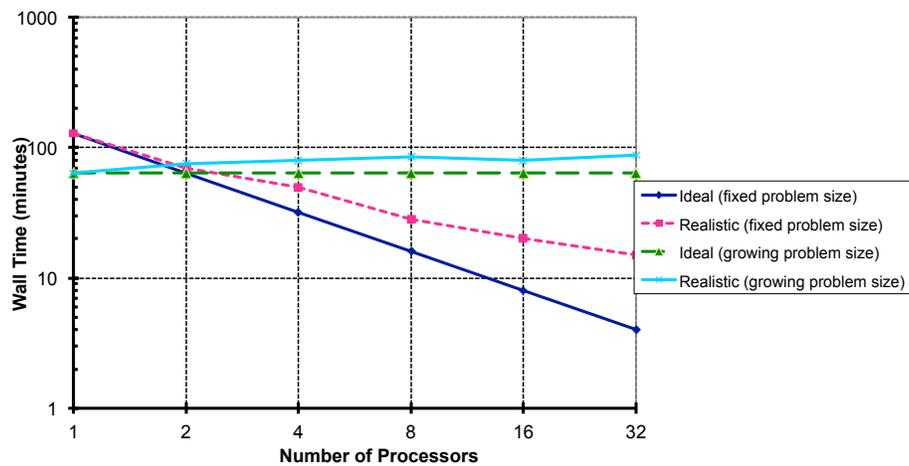
Determining Scalability of Program

- How to measure scalability
 - Fixed problem size, measure T_N for different N 's
 - Increase problem size proportional to N , compare T_N
- Repeat performance runs at least 3 times for each N (ideally >5 times)
- Plot on log-log graph; slope of line determines scalability

65 OLCF



Scalability



66 OLCF



Performance Evaluation

- Create performance model

$$T_N = T_N^{\text{communication}} + T_N^{\text{computation}} + T_N^{\text{serial}}$$

- Examine parallel algorithm and figure out which parts fit in each category
- Perform least-squares fit with scalability data

Bibliography/Resources: Programming Concepts and Debugging

- Heath, Michael T. (2006) *Notes for CS554: Parallel Numerical Algorithms*,
<http://www.cse.illinois.edu/courses/cs554/notes/index.html>
- MPI Deadlock and Suggestions
<http://www.ncsa.uiuc.edu/UserInfo/Resources/Hardware/CommonDoc/MessPass/MPIDeadlock.html>
- TotalView Tutorial
<http://www.llnl.gov/computing/tutorials/totalview/>
- Etnus TotalView page <http://www.etnus.com/>

Bibliography/Resources: MPI/ MPI Collectives

- Snir, Marc, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. (1996) *MPI: The Complete Reference*. Cambridge, MA: MIT Press. (also available at <http://www.netlib.org/utk/papers/mpi-book/mpi-book.html>)
- MPICH Documentation
<http://www-unix.mcs.anl.gov/mpi/mpich/>
- C, C++, and FORTRAN bindings for MPI-1.2
<http://www.lam-mpi.org/tutorials/bindings/>

Bibliography/Resources: MPI/ MPI Collectives

- Message Passing Interface (MPI) Tutorial
<https://computing.llnl.gov/tutorials/mpi/>
- MPI Standard at MPI Forum
 - MPI 1.1:
<http://www.mpi-forum.gov/docs/mpi-11-html/mpi-report.html#Node0>
 - MPI-2:
<http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.htm#Node0>

Bibliography/Resources: Benchmarking and Performance

- Hernandez, Oscar et al. (2010) *Debugging Applications on Jaguar XT5 with DDT*,
<http://www.nccs.gov/wp-content/uploads/2010/02/ddt-ornl-hex-core-workshop.pdf>
- Heath, Michael T. (2006) *Notes for CS554: Parallel Numerical Algorithms*,
<http://www.cse.illinois.edu/courses/cs554/notes/index.html>