Ranger Optimization

Release 0.3

Drew Dolgert

May 20, 2011

Contents

1 Introduction

This document accompanies a talk on optimization and scalability on the Ranger cluster at the Texas Advanced Computing Center.

1.1 Goals, Prerequisites, Resources

The goal of the talk is to help researchers use the Ranger system efficiently. Ranger is a large, distributed system made to run parallel, distributed code. We usually address how to accelerate applications in two steps, making code faster on one node, and making code faster in parallel across several nodes. The former is optimization, the latter scalability.

You should have the following skills by the end:

- Know how to write an allocation request.
- Know when to focus your efforts on writing code, compiling it, choosing a new algorithm, or finding another computer.
- Predict a parallel algorithm's effect on your program's speed from its complexity.

Prerequisites:

- An account on Ranger.
- Access to opti.tgz, a file with source materials.
- You must know how to compile and run codes on Ranger.

Resources:

To complete these exercises, there are no external resources you need. For further reading, you might seek the following:

- Designing and Building Parallel Programs, http://www.mcs.anl.gov/~itf/dbpp/ The task-channel method is relevant. The subsequent discussion of programming languages is less current.
- What Every Programmer Should Know About Memory - http://lwn.net/Articles/250967/ Memory is one of the most important things to understand for optimization.
- Quinn, Michael J., *Parallel Programming in C with MPI and OpenMP*, McGraw Hill, 2004. This is one of the better introductory books on parallel programming.
- Golub, Gene; Ortega, James M., *Scientific Computing: an introduction with parallel computing*, Academic Press, 1993. This is a gentle mathematial introduction to parallel algorithms.

1.2 Optimization and Scalability

Computational simulations are built on a series of choices.

1. What *model* would inform my research?
2. Which set of *algorithms* could express this model on a computer?
3. How should I *implement* those algorithms in code?
4. What *compiler* will best interpret that code, and with what hints?
5. On what *runtime environment* will this executable run?

These questions don't happen in order, though. You already know what model you want to compute and have probably chosen what algorithms, even what applications, you want to use to do that computation. On the other hand, you have a certain set of computers or clusters available, from Ranger to your laptop. It's the in-between steps, of compilation, implementation, and algorithm, where you get to make the most choices.

We cannot cover, in this talk, how to write good code. If writing code is crucial to your work, you can likely already construct the computational equivalent of a NASCAR engine. What we will do is remind you Formula 1 engines exist and take you on a tour equivalent libraries and applications on Ranger that might help you avoid doing your own coding.

Modern compilers can now outmatch all but the finest of hand-tuned efforts at constructing machine code. The key to using a compiler is to help it understand your code and inform it of all the capabilities of the architecture on which you will run. Once you know the options, finding the best compiler flags will still be an experimental process.

Scalability is whether an algorithm can remain efficient when run on many nodes. While we won't cover algorithm choice, we will learn how to measure scalability and how to use it in an argument about whether a particular algorithm is a good choice to run on Ranger.

1.3 The Ask

You can't get a research allocation on Ranger unless you can explain the scalability of your program, so how do you do that?

One of the Cornell groups working on Ranger was kind enough to let us use part of a draft of their winning application for a research allocation on Ranger.

The first section of the application is about the science, the second an explanation of what computations they do. Let's take a look at the third section, Preliminary Results. They offer two figures, here labeled Fig. 1 and Fig. 2. Their text follows.

> This section presents preliminary results obtained with stand-alone usage of all three components of our LES/FDF/ISAT methodology along with some information on their scalability. The scalability studies have been performed on TACC Ranger and on our CATS cluster.
>
> The parallel performance of the LES code in a variable-density flow has been studied for the case of a simple, non-premixed jet flame at Re=22,400 on a cylindrical grid with resolution 128 x 96 x 64. A flamelet table has been employed to provide the chemical parameterization of the filtered density as a function of mixture fraction. Preliminary scalability results are shown in Figs. 1 (a,b). It is seen that for the chosen grid resolution the LES code exhibits linear scalability up to 128 processors and reasonable scalability up to 256 processors. We expect improvements in scalability with increasing problem size.

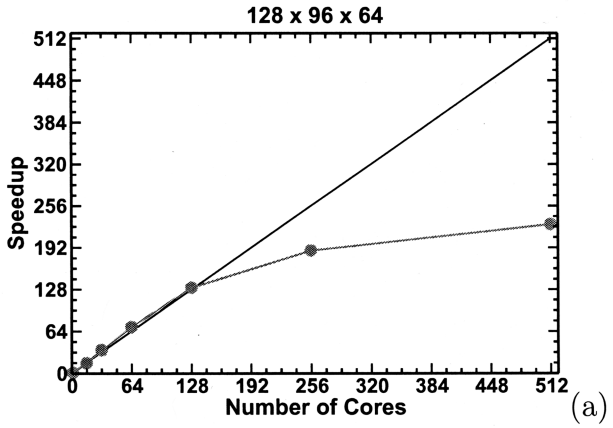


Figure 1: Speedup graph for example code.

This application is explicit about time from a previous application having been used to do scaling studies. The studies shown are for *strong scaling*, which means they examine how the solution time varies with the core count for fixed problem size. The last sentence of the application points out that *weak scaling,* meaning the change in wall time for increasing problem size, should remain efficient to much larger core counts.

1.4 Exercise: Allocation MadLib

Goal: Know how to write this section of an allocation proposal.

Demonstration of the current scaling capabilities of your code can be fairly straightforward. With the data given for a sample FFTW2 code, which solves a three-dimensional real transform, try filling in the MadLib below.

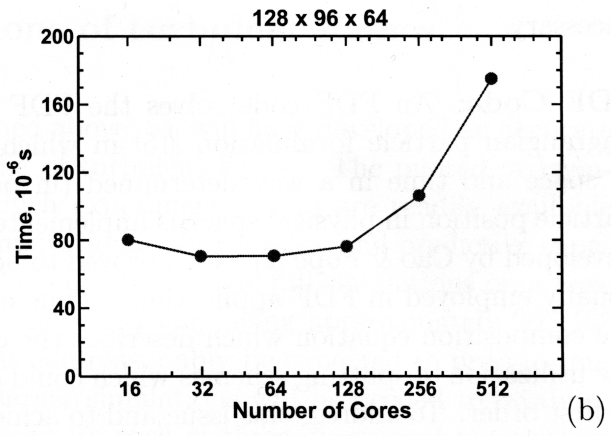
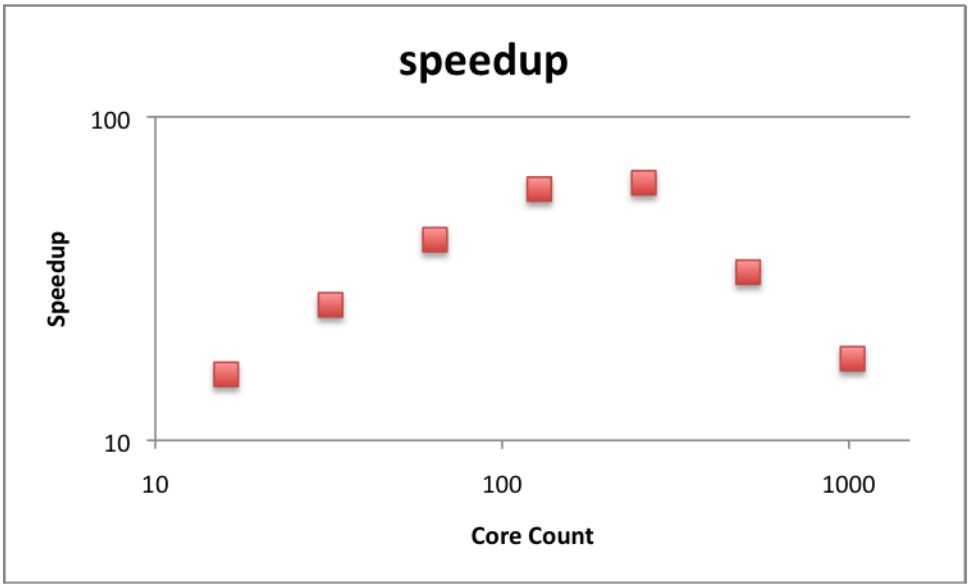
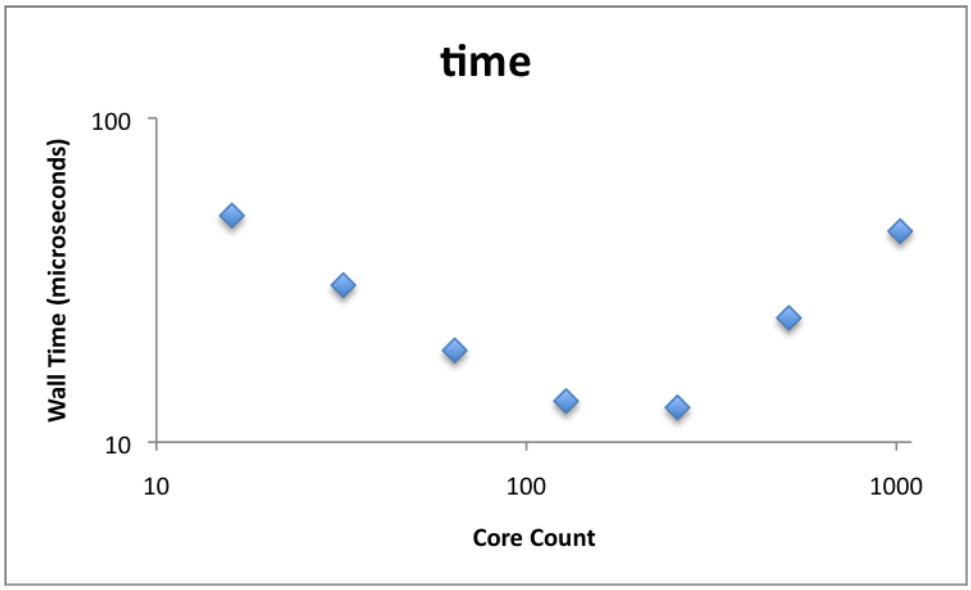


Figure 2: How wall time scales with number of cores, but the wall time is multiplied by the number of cores.

The main algorithm of our application is a spectral code which relies on fast fourier transforms. We use the FFTW2 library because it runs over MPI and is optimized for our target platform.





For a problem of size 1200 cubed, we make a benchmark that executes 100 iterations.

problem size	wayness	cores	time (s)
1200	16	16	50.01
1200	16	32	30.49
1200	16	64	19.18
1200	16	128	13.38
1200	16	256	12.79
1200	16	512	24.16
1200	16	1024	44.74

We tested the application on the Ranger system with 16 cores per node and Infiniband among nodes. For our smallest runs, with ___ cores, we see a wall time of ___. For larger runs, maintaining a problem size of 1200 cubed, the efficiency drops below 60% between ___ and ___ cores.

Given these results, we feel we can compute 10000 iterations on ___ cores, within __ SUs.

2 Numerical Libraries on Ranger

2.1 Discussion: Find a Relevant Math Library

Pick from the list below a library that you don't already know. See if you can find something interesting about it from its documentation. Compare with your neighbor, and we'll share that with the group. Typical questions are

- Is it multi-threaded?
- Can it run distributed (which means across nodes, which means with MPI)?
- What languages can use it?

There are a lot of domain-specific applications on Ranger. Let's just look at some of the math libraries.

- acml - AMD Core Math Library, lapack.
- arpack - Fortran subroutines to solve large scale eigenvalue problems.
- fftw2 - A fast, free C FFT library; includes real-complex, multidimensional, and parallel transforms. Version 2 can use MPI.

- fftw3 - A fast, free C FFT library; includes real-complex, multidimensional, and parallel transforms.
- glpk - GNU linear programming kit.
- gotoblas - Hand-tuned Basic linear algebra subroutines.
- hypre - library for solving large, sparse linear systems of equations
- mkl - Intel Math Kernel Library.
- mpfr - C library for multiple-precision floating point.
- numpy - Tools for numerical computing on Python.
- petsc - data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations.
- plapack - Parallel linear algebra matrix manipulations.
- pmetis - parallel graph partitioning.
- scalapack - linear algebra routines using MPI.
- scalasca - open-source analysis of parallel applications.
- slepc - Scalable library for eigenvalue problems.
- sprng - scalable parallel psuedo random number generation.
- trilinos - algorithms for large-scale, complex multi-physics engineering and scientific problems.

2.2 Exercise: Compare Libraries and Hand-written Code

Goal: Understand when to use numerical libraries in your code.

Here, you compare three versions of the same code to solve a matrix. The sample code has all three versions, a makefile to compile them, and a script to submit them to the scheduler. Embedded in that script is code to time how long each runs and put it into the output file.

Unpack the source code with:

```
$ mkdir labs
$ cd labs
$ tar zxf ~train100/opti.tgz
```

You will find a directory called `ludecomp-gsl-nr`. In it are three codes, one using Numerical Recipes and one with Gnu Scientific Libraries.

- The `nr` version uses code copied from a book called Numerical Recipes. It does not need any external libraries.
- The `gsl` version uses Gnu Scientific Libraries. You need to load these libraries in the Ranger environment.
- The `lapack` version uses either the Math Kernel Libraries if you are using the Intel Compiler, or the PGI Lapack libraries if you are using the PGI compiler. Load the appropriate one for whichever compiler you choose.

The Makefile has two commands you might use:

- "make" - This makes all three versions of the program: nr, gsl, and lapack. If make fails, it is likely because it wants libraries loaded that are not currently loaded.
- "make clean" - This deletes all binaries you compiled for a clean start if you are switching from Intel to PGI, or vice versa.

1. Add `gsl` to the currently-loaded modules.

2. Ensure `pgi` is the current compiler, again using the module command.
3. Type `make` in the `~/labs/ludecomp-gsl-nr` directory.
4. Submit the job.sge script with your account in it. One way to do this is `qsub -A your_account job.sge`. The results of your runs will be in results.txt.

There are three versions of the same code. They are in nr.c, gsl.c and lapack.c. Evaluate each one on these criteria:

1. How many lines of code did it take to implement the solution?
2. How fast does it run? Look at results.txt when it finishes.
3. For each version, how hard would it be to use an alternative algorithm, for instance, to use an iterative solver or to solve a sparse matrix?
4. Which codes can run multithreaded? Can they run distributed, using MPI? You may need to Google the library to figure this out.

3 Compiler Optimization

3.1 Common Compiler Options

Standard Choices are:

- PGI: `-O3 -fast tp barcelona-64 Mipa=fast`
- Intel: `-O3 -xW -ipo`

Don't exceed -O2 without checking whether your output is correct.

PGI on Ranger

- -O3 - Performs some compile time and memory intensive optimizations in addition to those executed with -O2, but may not improve performance for all programs.
- -Mipa=fast,inline - Creates inter-procedural optimizations. There is a loader problem with this option.
- -tp barcelona-64 - Includes specialized code for the barcelona chip.
- -fast - Includes: -O2 -Munroll=c:1 -Mnoframe -Mlre -Mautoinline -Mvect=sse -Mscalarsse -Mcache_align Mflushz
- -g, -gopt - Produces debugging information.
- -mp - Enables the parallelizer to generate multi-threaded code based on the OpenMP directives.
- -Minfo=mpi,ipa - Provides information about OpenMP, and inter-procedural optimization.

Intel on Ranger and Lonestar

- -O3 - More than O2, but maybe not faster
- -ipo - Creates inter-procedural optimizations.
- -vec_report[0|..|5] - Controls the amount of vectorizer diagnostic information.
- -xW - Includes specialized code for SSE and SSE2 instructions (recommended for Ranger).
- -xSSE4.2 - Recommended for Lonestar so that it uses SSE4.2.
- -xO - Enables the parallelizer to generate multi-threaded code based on the OpenMP directives.
- -fast - Includes: -ipo, -O2, -static DO NOT USE – static load not allowed because only dynamic loading is allowed.

- -g -fp - debugging information produced.
- -openmp - Enable OpenMP directives
- -openmp_report[0|1|2] - OpenMP parallelizer diagnostic level.

3.2 Exercise: Test Numerically-intensive Code

Goal: Understand the process of optimizing compilation.

In this exercise, you return to the ludecomp-gsl-nr directory to experiment with how different compilers and compiler options affect the codes. Everybody's results are recorded on The Eric Chen Scoreboard.

1. Return to the `~/labs/ludecomp-gsl-nr` directory.
2. As we did in the Ranger Environment talk, use the `module` command to select a compiler (PGI or Intel) and Lapack library (ACML or MKL).
3. Edit the first two lines of the Makefile so that the COMPILER matches your compiler choice in the shell: pgicc or icc for PGI or Intel. Use options in sections above to pick values for FFLAGS.
4. Compile the three codes with `make`.
5. Submit the codes to the scheduler with `qsub -A 20100714HPC job.sge`.
6. Examine results.txt, which will appear after the job runs, to see how long it ran.
7. Iterate through a few choices of compiler and optimizations to see what is fastest.
8. What columns would you need in a chart to show your choices, and how do you choose what represents the results? Our main result is speed. What other choices could matter, depending on your problem?

4 Scalability

4.1 Exercise: Analyze a Parallel System

Goal: Understand how to identify serial and parallel parts of an algorithm and analyze how these change with problem size and number of threads.

1. Think of a problem where several actors, or workers, or individuals, work together to do a task. Make it a task where each worker does the same kind of activity, so a poor example would be people building a house. A better examle would be people building a fence. You can use actual computer algorithms or any other example you like.
2. Identify who are the actors and what is the work to be done. You will need some measure of how much work there is: feet of fencing, stacks of sticks, pounds of pork.
3. Identify communication patterns among workers. Communication often requires synchronization. If there is no part where workers communicate, you have the option to make your model problem more complex so that it requires some kind of communication at some point.
4. Are there parts of this task that each worker has to do separately so that the task cannot be shared? These are serial parts.
5. Write an equation to express the time it would take one worker to complete this problem.
6. Write an equation to express the time it would take N workers to complete this problem. Is there some part of the problem, often at the beginning or end, that does *not* get faster when there are many workers?

7. How much faster would N workers do the same problem? This is speedup. Express this algebraicly and simplify it. You might find that expressing the serial part of the work as a fraction of the total will simplify the equation. In some cases, it might not, and that's OK, too.
8. What is the minimum time to do this work if you have lots of workers? What does that make the maximum speed? This is the source of Amdahl's Law.
9. If you increase the amount of work as fast as you increase the number of workers, what happens to the speedup?
10. When there are many, many workers on the same problem, adding more workers doesn't make the problem faster, so they are inefficient. How would you measure efficiency? Could you come up with an equation for this, using what we have written so far? As a guideline, one worker would always be one hundred percent efficient by our estimate. When would N workers be one hundred percent efficent?

4.2 Exercise: Excel Demo of Fluent Model

Goal: Understand the different regimes of scaling and their dependence on latency and bandwidth.

For this exercise, we will look at an Excel spreadsheet that has analysis similar to what you would get from mpiP. We will use it to understand how Fluent might behave on different kinds of computers.

The Fluent program calculates fluid dynamics. It is a spectral code, like fourier transforms. To construct the spreadsheet, we ran Fluent in parallel using MPI and measured the number of messages sent and the size of those messages, all as a function of the number of parallel tasks.

Given this, we assume:

- The main work speeds up according to the number of cores.
- Every message sent incurs a time penalty for latency.
- The time to send each message depends on its size, according to network bandwidth.

The data, on the second page of the spreadsheet, show that, as you increase core count, Fluent sends disproportionately more messages and longer messages.

Download FluentEstimate.zip to the desktop. Double-click it and open the Excel 2007 file inside.

There are two graphs, one showing the speed on the right, and one showing the relative contribution of work, latency, and bandwidth on the left. The two sliders change latency and bandwidth.

1. Is our simplistic model enough to explain rougly at what core count the code becomes less efficient?
2. Infiniband and Gigabit ethernet have similar bandwidths, but Infiniband has much lower latency (under 2.3 microseconds on Ranger). Would using a cluster with this feature help run Fluent at higher core counts?
3. At what core count does the time to send messages become commensurate with the time to do the main work? How does this change as you lower latency? As you increase bandwidth?

4.3 Exercise: Measure Strong Scaling of Fourier Transform

Goal: Know how to estimate your code's scalability.

You will find code in `~/labs/fftw_mpi` which contains a parallel MPI FFTW2 code, the same one we used for the chart earlier. You have to add the modules for `pgi` and `fftw2` in order to compile it, which is done by typing `make`.

We ask, in this exercise, how much faster a parallel program can complete the same amount of work given an increasing number of processors. We need to choose an appropriate amount of work. That might normally be determined by the research, but here, we are limited by the amount of memory required to store an N x N x N array. A side length of 1200 seems to work well.

1. Run it on 1, 4, 16, 64, 256 processors by changing the makefile.
2. Make a chart on your local computer using a spreadsheet. It should be a log-log chart. Is the chart smooth?
3. You could fit the chart in order to estimate it.
4. What is the efficiency of running 16-way parallel on one node?
5. What is the efficiency of runnning 256-way parallel?

4.4 Exercise: Measure Memory Usage of Fourier Transform

Goal: See how memory per task changes with problem size and core count.

We ask how efficiently a parallel program can complete an *increasing* amount of work given an *increasing* number of processors. Most of the challenge here is finding what problem sizes fit into available memory. You will have to decrease wayness in order to get problems to fit in smaller core counts.

1. What is the largest size problem you can run 16way with 16 cores? What if you keep it 16way but increase core count? Note that the program prints its memory usage for the main work arrays.
2. Reduce wayness to 1way or 4way. How does it behave now at different core counts?
3. First make a standard log-log chart of the wall time at different core counts.
4. Look at memory usage per core for different core counts and problem sizes. This might be a more important metric for running this kind of code.

4.5 What Machines Are Appropriate for Your Work?

Look at the TeraGrid Portal Resource List. If you click on a name on the left, you will see a description of each machine. What are you looking for in those descriptions? It depends on what you need for your program, but the capabilities of those computers might also dictate what you imagine for your research.

- How much memory per core is on the machine? How much memory can a single process access?
- How many cores can you use for a multi-threaded portion of the code?
- Is the interconnect slow, like Gigabit Ethernet, or fast, like Infiniband?
- Does the interconnect have a special topology to support faster collective communications for large core counts?
- Are the filesystems parallel? What are the read and write rates?
- Will the chipset run regular Linux applications or is it special purpose, such as the BlueGene?
- Is the queue full for days?