A Study of High Performance Computing and the Cray SV1 Supercomputer

Michael Sullivan
TJHSST Class of 2004

June 2004

0.1 Introduction

"A supercomputer is a device for turning compute-bound problems into I/O-bound problems."
–Seymour Cray

High Performance Computing (HPC) is the branch of computational research that focuses on developing systems that provide more computing performance, power, or resource than is generally available. From the software perspective, this usually means developing parallel processing algorithms and programs that can be processed simultaneously by multiple processors. From the hardware perspective, supercomputers generally focus on distributed systems, with multiple processors and interconnected memory. Depending on the expected cost and final use the system is designed for, supercomputers may be comprised of cutting-edge technology, with massive memory banks, or may be a distributed system of off-the-shelf components. There are advantages and disadvantages to each approach – no one solution is ideal for all purposes.

0.2 Uses

Supercomputers are mainly used for applications which fit into one or more of the following categories.

1. Long: A process which requires many CPU hours to complete.
2. Large: Requires large memory and/or disk space.
3. Distributed: Process composed of many independent tasks.
4. Time critical: The amount of processor time it takes to complete is important.

For any application which does not fit into these criteria, a lower-cost solution may be advisable.

0.3 Types

There are four main types of supercomputers:

1. SMP clusters: Shared-memory clusters utilizing non-vector processors, most using RISC microprocessors.
2. Vector clusters (or Vector SMPs): Shared-memory clusters utilizing vector or array processors.
3. COWs (Clusters of Workstations): Distributed memory computers, often comprised of hundreds or thousands of separate off-the-shelf PCs.

4. SIMD / Special purpose array processors: Single instruction, multiple data computation popularized by early vector processors and now implemented in commercial CPUs.

The Cray-SV1 is considered to be a Vector cluster, due to its 16 SMP vector processors and central memory bank.

0.3.1 Grid Computing

"If you were plowing a field, which would you rather use? Two strong oxen or 1024 chickens?"
– Seymour Cray

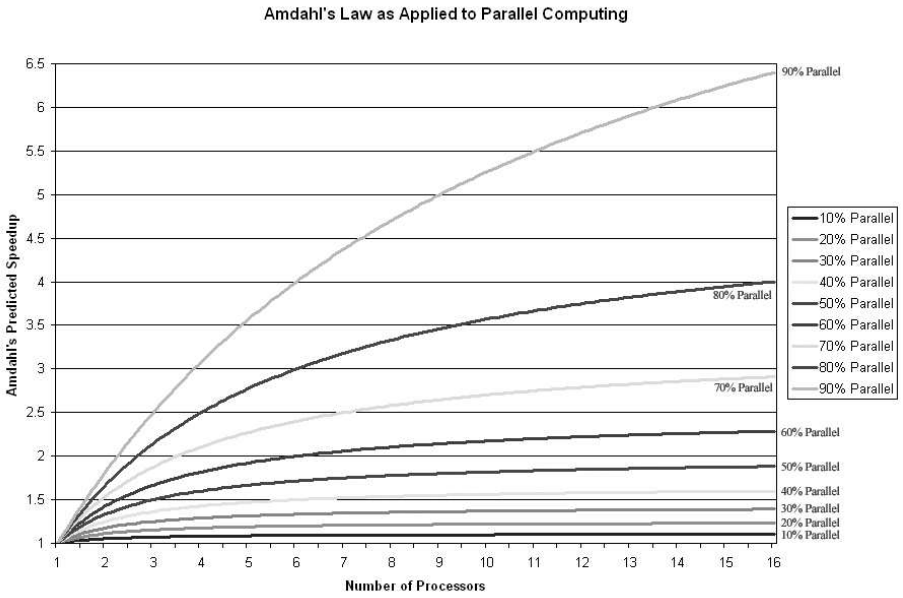
A low-cost alternative to traditional cluster or distributed computing models is grid computing. Grid computing utilizes large numbers of computers arranged as clusters embedded in an open telecommunications infrastructure. This approach to distributed computing involves the sharing of heterogeneous resources running all different platforms and architectures located in different places all over the globe. The ability of grid computing to transcend administrative domains, software platforms, and hardware architectures sets it apart from standard distributed computing approaches.

0.4 Amdahl's Law as Applied to Parallel Computing

As the number of processors increases, even highly parallelized code does not increase proportionally in speed (thus, 2 processors do not run code at twice the speed of 1 processor).

Amdahl's Law states that if P is the fraction of a calculation that is parallelized, and thus (1-P) is the fraction that is sequential, then the maximum speedup that can be achieved by using N processors is:

$$1/((1-P)+P/N) \tag{1}$$



Notice that if it were possible to parallelize 100% of all code, then there would be a 1:1 relationship between number of processors and speedup, and 2 processors would run at twice the speed of 1 processor. However, this is not generally possible, because there is often a need to have sequential operation in a program which is difficult to parallelize. Also, small processes do not benefit from parallelization, since the overhead of passing data between processors can outweigh the computational boost of distribution if the application or data size is too small.

0.5 Parallelization

In order to parallelize sequential code, one needs to partition it into smaller pieces that can be run independently and simultaneously on separate processors. The partitioning process is called decomposition, and it can be achieved in two distinct ways: Data Parallelization, and Task Parallelization.

0.5.1 Data Parallelism

Data, or "fine grain" parallelism, allows many processors to work on one large task simultaneously by assigning each processor a piece of the data to work on. This means that the same code is run on each processor, but each CPU works on its own, unique part of the code, allowing a single task (such as a "for" loop) to be split among many processors as relatively small subtasks.

The defacto standard for data parallelization that has emerged in the past few years is the OpenMP set of compiler directives. This set of directives can be inserted into code in order to parallelize a single loop or task. Although OpenMP can also be used for Task Parallelism, doing so requires a comparitavely large amount of manual reprogramming (comprable to that needed to implement message passing), and OpenMP will not generally scale as well to a large number of processors as will MPI.

The advantages of OpenMP and Data Parallelism:

1. Very efficient for shared memory systems, yet also viable for distributed memory systems.
2. Naturally balances loads among available processors.
3. Can quickly and efficiently be implemented automatically by capable compilers.
4. Locking and synchronization not generally needed on shared memory systems.

The disadvantages of OpenMP and Data Parallelism:

1. The programmer has limited control over the parallelization process.
2. Data placement and scope may have a serious impact on code execution.
3. Parallelization of very small processes may provide negative speedup due to team creation overheads.
4. Scalability is limited as the number of processors increases, which is mainly a result of the automatic parallelization of code.

0.5.2 Task Parallelism

Whereas data parallelism performs the same operations concurrently on different parts of the data, task or "coarse grain" parallelism runs different operations on each processor simultaneously. With task parallelization, a piece of code is split into independent tasks, or subroutines, that can be run on multiple processors simultaneously.

Often, task parallelism offers a higher amount of parallelization than does data parallelism. This is because there is often less data overhead associated with task than data parallelism, and a greater amount of the code can be parallelized since each processor can run sequential processes so long as they operate independent of other subroutines.

The defacto standard for task parallelization on a large scale is MPI, or the Message Passing Interface. Using MPI, processors can be instructed to perform independent tasks simultaneously, and can communicate using a standardized

syntax. Although MPI can also be used for Data Parallelism, it is not generally an efficient solution due to the large amount of manual reprogramming necessary to implement message passing, and the fact that MPI's communication costs may dominate any potential speedup when dealing with smaller tasks.

The advantages of MPI and Task Parallelism:

1. Any independent process can be executed in parallel.
2. Runs well on both shared and distributed memory systems.
3. Scales well to a very large number of processors.
4. Locking memory not necessary.

The disadvantages of MPI and Task Parallelism:

1. Reprogramming is often necessary in order to balance processor loads and synchronize processes.
2. Manual parallelization required.
3. Large, independent tasks are necessary to overcome communication costs.
4. Collective operations and communication are computationally expensive, and may limit the number of processors that can effectively be used.

0.5.3 Explicit Threading

Another possiblity for course or fine grain parallelism is the defintion of explicit threads (such as PThreads or Windows threads) which operate independently on different processors. However, with this method, one has to communicate between processes using shared memory regions, and worry about locking and synchronization is left to the programmer. Because of the need for shared memory regions for collaborative processes, explicit threading is not generally used on distributed memory systems. Explicit threading is often used to parallelize code for a small number of processors (i.e. dual processor workstations), but is not a robust solution for parallelizing code for supercomputing or scientific purposes.

0.5.4 Multi-Level Programming

Both data and task parallelism offer their own architectural and computational advantages and disadvantages. In order to get the maximum performance out of modern supercomputer architectures, it is often advantageous to combine the two styles of parallelism in the same program, which results in multilevel or hybrid code.

The main performance benefit of multi-level programming is the general improvement in program scalability over the exclusive use of course or fine

grain parallelism. Also, when properly implemented, multi-level code tends to balance loads almost equally among a large number of processors, even when relying heavily on coarse grain parallelism.

0.6 Fortran

FORTRAN, short for "Formula Translator/Translation", is a programming language that was developed in the 1950s and is still in use today. Originally capitalized, the language has lost capitalization starting with Fortran 90, and now is referred to as "Fortran".

Fortran was widely adopted by scientists for writing numerically intensive programs, and the language's inclusion of a complex number data type makes it especially suited to scientific computation. The high performance backing of Fortran has encouraged compiler writers over the years to produce compilers that generate very fast code. Many vendors of high performance Fortran compilers, including Cray, have added specialized extensions to the language for special hardware features (instruction cache, CPU pipeline, vector arrays) which are specific to their products' architecture. Many of these extensions have been incorporated into the main standard, making the proprietary extensions obsolete. One notable exception is OpenMP, which, as was noted previously, is a common cross-platform extension for shared memory programming.

0.7 References

1. COSC 3601 Lecture Notes. Available at: http://oldsite.vislab.usyd.edu.au/education/COMP4601/index.html.

2. Dowd, Kevin; Severance, Charles (1998). High Performance Computing, 2nd Edition. O'Reilly & Associates; 2nd edition (July 2, 1998).

3. NCSA/WebCT Courses: Introduction to MPI, Multilevel Parallel Programming, Parallel Computing Explained, Performance Tuning for Clusters. Available at: http://webct.ncsa.uiuc.edu:8900/webct/public/home.pl.

4. Wikipedia Definition: Fortran, Grid Computing, Parallel Computing, RISC, Supercomputer, Vector Processor. Available at: http://en.wikipedia.org.