8-2010

A Study of Improving the Parallel Performance of VASP.

Matthew Brandon Baker
East Tennessee State University

A Study of Improving the Parallel Performance of VASP

A thesis

presented to

the faculty of the Department of Computer Science

East Tennessee State University

In partial fulfillment

of the requirements for the degree

Master of Science in Computer Science

---

by

Matthew Baker

August 2010

Dr. Phillip E. Pfeiffer, IV, Co-chair

Dr. Frank B. Hagelberg, Co-chair

Dr. Christopher D. Wallace

Keywords: Cluster, HPC, Parallelism, MPI, OpenMP, VASP

ABSTRACT

A Study of Improving the Parallel Performance of VASP

by

Matthew Baker

This thesis involves a case study in the use of parallelism to improve the performance of an application for computational research on molecules. The application, VASP, was migrated from a machine with 4 nodes and 16 single-threaded processors to a machine with 60 nodes and 120 dual-threaded processors. When initially migrated, VASP's performance deteriorated after about 17 processing elements (PEs), due to network contention. Subsequent modifications that restrict communication amongst VASP processes, together with additional support for threading, allowed VASP to scale up to 112 PEs, the maximum number that was tested. Other performance-enhancing optimizations that were attempted included replacing old libraries, which produced improvements of about 10%, and prefetching, which degraded, rather than enhanced, VASP performance.

CONTENTS

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure

Page

CHAPTER 1

INTRODUCTION

This thesis involves a case study in the use of parallelism to improve an application's performance. In order to improve the performance of contemporary computers, computer architects are increasingly turning to parallel architectures. Parallelism allows for a program to run many tasks simultaneously, a situation which, as a rule, speeds program completion [7]. The application, VASP, is used for computational research on molecules [3]. VASP simulates the electronic structure of molecules, ranging in size from large molecules to nanostructures: i.e. systems with at least one dimension in the order of nanometers. At East Tennessee State University (ETSU), VASP is used primarily for understanding and designing materials of interest for nanoelectronics, or electronic circuits scaled down to nanoscopic dimensions.

VASP is routinely used to simulate systems are computationally demanding. In order to improve its performance, VASP's authors parallelized its operation using the "Message Passing Interface" (MPI) standard. The MPI standard defines a model of parallelism in which multiple cooperating processes share information by exchanging communications known as messages over a computer network. MPI supports the execution of these processes on the same or different machines as well as communication between processes launched on the same or different systems.

The MPI model of process operation includes a library of functions for inter-process communication, as well as a standard model of execution, where a program is started ("launched") by a command—typically mpiexec. As part of program launch, a user specifies the number of cooperating processes in a program execution. The MPI API supports various types of communication among these processes, including point-to-point, broadcast within user-specified groups of processes, and all-process broadcasts. The version of VASP in use at the outset of this

study made heavy use of collective operations in MPI, especially the MPI_alltoall function, which directs all of a computation's nodes to exchange some data with all other nodes.

1.1 The Problem

This work sought to migrate the version of VASP in use at the outset of this study to a more powerful machine, in a way that effectively exploited that second machine's parallelism. This initial version of VASP was configured for a system consisting of 4 nodes each with 1 quad core AMD Opteron CPU called PQS, using OpenMPI and the AMD HPC libraries (ACML). The target host, Blackpearl (blackpearl.etsu.edu), is a 60-node, 120-CPU cluster with dual hyperthreaded CPUs and native OS support for OpenMPI.

Extensive tests conducted since 2000 show that VASP achieves optimum performance when run in parallel on 50 to 150 CPUs [12]. When initially installed on Blackpearl, VASP exhibited adequate speedup on up to 16 CPUs, similar to its performance on PQS. Adding CPUs beyond 16 would cause VASP to run slower rather than faster.

1.2 Approach

Initially, the VASP source code and other build-related files were transferred to Blackpearl, then built using the Intel compiler and a slightly modified Makefile. The code, when run with 16 CPUs, was nearly twice as fast as when run on PQS, due to the better hardware available in Blackpearl. To determine how to make VASP run still faster, the Blackpearl version of VASP was profiled to determine what portions of the code dominated the CPU use.

The first attempt to optimize VASP replaced AMD libraries for matrix operations with matrix libraries optimized for Blackpearl's Intel CPUs. This change was inspired by an initial round of profiling with Gprof (see 3.2.1) and Valgrind (see 3.2.2), which showed VASP spent much of its execution time in matrix operations.

The second attempt involved prefetching data for a few loops in VASP that showed large numbers of data reads. These changes typically slowed execution. A second analysis of the Gprof and Valgrind profiles showed that these profiles were incomplete: Gprof and Valgrind only accounted for 25% (20 minutes) of the program's actual wall-clock time, due to their failure to measure I/O and operating system calls.

Because VASP's system calls appeared to be dominated by networking operations, network access was analyzed with libraries for profiling MPI communication. VASP's use of MPI_alltoall communication, which worked fine on the smaller 16 node cluster, appeared to overwhelm Blackpearl's Ethernet network.

After networking time was isolated as the scaling problem, two strategies were used to reduce the load on Blackpearl's Ethernet network. One limited the number of processes that could communicate simultaneously by separating running processes into groups of a predefined size, and replacing all-to-all communication with localized, intra-group communication. The other change reduced the number of MPI process per node to one, while directing each MPI process to spawn one thread for each of its node's remaining logical CPUs. A thread is essentially a lightweight computation that shares most of a memory space with a parent process [11]. This second change reduced inter-node communication, which left some CPUs idle when VASP was executing a sequential routine. This penalty, however, was more than offset by the speed gains from more efficient MPI communication.

1.3 Balance of Thesis

The rest of thesis is divided into five sections. Section 2, a background section, describes VASP and reviews performance-related concerns, including strategies for achieving parallelism and measuring performance. Section 3 discusses the optimizations attempted for altering VASP

performance and what was expected of each. Section 4 presents the results of each optimization. Section 5 analyzes the optimizations and their relative value. Finally, section 6 reviews the work and provides suggestions for further research.

CHAPTER 2

BACKGROUND

2.1 Parallel and Distributed Computation: Key Concepts

2.1.1 Platforms.

2.1.1.1 Hardware. Traditionally, the most common strategy for architecting computers supported all computations using one Processing Element (PE) [7]. This processing element, known as the Central Processing Unit (CPU), supported the execution of one process at a time, with one memory address space running on one physical machine. Over time, two basic strategies have been used to make this architecture run faster. The first is to improve the performance and utilization of a machine's existing components, including its main memory, secondary storage subsystems, and PEs. The second method is to increase the number of components, in the hope that software can exploit the increases.

Adding more PEs to computing systems has become attractive for economic reasons. In the past, architects relied primarily on increases in single PE speed to increase performance. This strategy was safe because the effective speed of a single PE typically doubled every 18 months. In an article published in 2005, Asanovic et al. [1] argued that single PE speeds will increase more slowly in the future, due to limits to power consumption, memory speed, and performance increases obtainable through instruction pipelining. Because of this, adding more PEs to a system has become the most cost-effective path to performance improvements.

Multiprocessor systems support multiple, complete PEs that access a common hardware address space [7]. Multiprocessors simplify the sharing of work among PEs because all PEs have direct access to this common memory. A multiprocessor system can be difficult to program, because access to shared data must be synchronized to prevent one PE from changing values that

other threads are actively using. Multiprocessors also do not scale well, due to contention for system resources such as disk and memory.

Simultaneous multithreading (SMT), also referred to as hyperthreading, augments PEs with multiple instances of registers that hold program state and data [7]. These extra registers, which typically include general purpose and status registers, allow processes that wait for non-PE-based operations like data transfers and system calls to switch to a new process quickly, without the need for spilling and reloading registers. While the speed up from hyperthreading is not as great as for multiprocessors, hyperthreading is less expensive than adding another full PE.

Multicomputers support multiple PEs by placing each PE in its own hardware address space. Multicomputers typically consist of multiple physically separated machines with a common interconnect [2]. While this use of separate address spaces eliminates the need to synchronize PE access to memory, processes must exchange data by copying it between address spaces—a process that slows computation, even with a fast interconnect between the PEs.

A cluster is a set of multiple computers, any of which could be run as a stand-alone system, that use a common networking interconnect to cooperate on a common task [7]. Often these machines are identical, though this is not necessary. Data sharing is much slower, because the data has to be routed through the interconnect.

2.1.1.2 MPI. MPI is the de facto standard for implementing parallel code for distributed computing [9]. MPI defines a programming and runtime environment that supports the launching of applications that consist of multiple processes in interconnected computer networks. The environment also provides an Application Programming Interface (API) that allows for communication between the application's processes.

An MPI-based application is invoked using MPI's launcher, which starts processes locally and on remote machines and assigns to each process a unique number, called its rank. Ranks allow each of an application's processes to identify itself for the purpose of deciding what logic to run, and to direct messages to and receive messages from the application's other processes.

The most basic MPI functions for network communication are MPI_Send and MPI_Recv, which support point-to-point message transmission, and MPI_Bcast, which broadcasts a value from a process with a specific rank to all other processes in the network. The MPI_alltoall function synchronizes the contents of a specified array across a specified set of processes by sending some data from each node and collecting all data from each other node.

2.1.1.3 Software Architectures. VASP does not appear to have a formal architecture. The code is highly coupled with little modularity and follows no particular design philosophy. The only major exception to this appear to be the MPI functions, where higher level communication functions were built on top of MPI.

2.1.2 Performance

The primary metric used here for evaluating application performance is wall clock time: the total time for which a program runs. This includes the time spent in MPI initialization, network communication, disk IO, and PE execution, and excludes time spent waiting in the cluster scheduler. Wall clock time is cited by authorities such as Hennessy and Patterson as the only reliable metric for performance [7].

Program speedup and performance are defined as in Hennessy and Patterson [7], where the phrase "X is n times faster than Y" and n is defined as follows:

$$n = \frac{\text{Wall Time}_Y}{\text{Wall Time}_X} \qquad (1)$$

A second metric for program performance, parallel efficiency, identifies the extent to which additional CPUs contribute to gains in overall run time. Parallel efficiency measured as the ratio of a program's best runtime on a single CPU, relative to a given workload, to that program's average runtime on a given number of CPUs, multiplied by the number of CPUs:

$$\text{efficiency} = \frac{\text{Single CPU runtime}}{\text{Wall time} * \text{Number of cpus}} \qquad (2)$$

Program wall clock time was measured by using the program's execution command as a parameter to the standard Unix command 'time'. For parallel programs, the 'time' command reports the time on the head node rather than time on worker processes. Because this approach also measures how long it takes to launch and tear down the MPI network, runs with more nodes will incur additional sequential overhead, due to the extra time spent by the MPI launcher script to start and manage the larger launch. This additional overhead, however, is still of interest in gauging the efficiency of using more CPUs.

2.1.3 Performance Optimization.

2.1.3.1 Overview. This study employed four key strategies to maximize performance. The first, replacing slow code with faster code, improved VASP's sequential performance. The second, data prefetching, was also intended to improve sequential performance. Prefetching was applied at the level of the CPU cache in an attempt to increase the efficiency of tight loops where VASP spends most of its time. The third, reducing message latency, was intended to improve VASP's parallel performance by reducing the time that processes waited for data before computing with it. The fourth, improving network throughput by reducing network load, was also intended to improve parallel performance by reducing latency.

2.1.3.2 Replacing slow code. VASP's performance, like that of other applications, is sensitive to the hardware on which it is run. The initial migration of VASP from PQS to

Blackpearl included the AMD math libraries, which did not perform well on Intel hardware. Replacing the AMD libraries with Intel libraries produced a quick and easy performance gain.

2.1.3.3 Prefetching. Prefetching involves the transfer of data from one memory to another (here, from main memory to a PE's cache) before a code accesses it [7]. Prefetching is intended to reduce or eliminate the need for computations to wait on data transfers. Most compilers provide built-in functions (intrinsics) that generate prefetch code. Certain compilers also offer options that direct the compiler to produce prefetch code in areas where the user believes that performance gains are possible.

2.1.3.4 Communication Efficiency. The speed at which a parallel application's processes communicate can affect that application's ability to run effectively. The strength of this effect depends on the frequency of process communication and message size. Latency has the greatest effect on performance when applications communicate frequently using small messages.

Communications between a process's threads typically exhibit low latency because of shared address spaces. Communication with a process on the same PE with a separate address can still be quick, though this may involve copying data from one part of the machine's address space to another. Communication between processes on separate PEs is slower, due to the need to send data across a network.

When a node's PEs run threads rather than separate MPI processes, network communication can be made more efficient. When an MPI process is copying data from one process to another on the same node, MPI will still use the network to transfer data. Using threads eliminates MPI networking overhead.

2.2 VASP

VASP is a widely used materials science software package [3] that employs density functional theory (DFT) to model many-body systems. In materials science, the DFT method is used to obtain stable structures of molecular and condensed matter systems, and, on the basis of these structures, to derive their physical and chemical properties. A material's structure determines many of its chemical and physical properties, among them its stability, reactivity, magnetism, and response to light. Identifying structure is thus among the primary tasks of computational materials science. A stable structure is characterized by the nuclear positions that correspond to a minimum of the system's total energy. This minimum can be determined by varying these positions with respect to each other in order to minimize the unit's energy landscape, defined as its total energy relative to the nuclear position.

DFT is rooted in the observation that all properties of a compound in its most stable state may be expressed in terms of this state's electron density characteristic [8]. Problems in materials science are commonly framed in terms of complex systems involving many electrons. DFT simplifies these problems by mapping a system of N electrons and M nuclei onto an auxiliary model that requires far less computational effort to solve. While the modeled system's N electrons are mutually repelling, the auxiliary model uses N non-interacting electrons, each moving under the influence of the average repulsion due to all other electrons as well as the nuclei's attractive force. It has been shown that the fictitious system has the same electron density and the same energy as the real one [8]. It is not known, however, how to exactly compute the average electronic force. The VASP program operates with certain well-tested approximations to this quantity.

A search algorithm implemented in the VASP program, the Conjugate Gradient Optimization technique[5], efficiently computes a system's minimum total energy. The execution time of this algorithm scales as a third-order polynomial in the number of electrons, i.e. as $aN + bN^2 + cN^3$. While this scaling behavior is dictated by the nature of DFT, the authors of VASP kept the prefactor c of the cubic term low, such that it impacts the program performance only for relatively extensive systems that contain several thousands of electrons. This feature makes VASP very suitable for studies of condensed matter systems from first principles.

CHAPTER 3

METHODS

3.1 Computing Platforms

3.1.1 PQS

VASP was originally run on the Parallel Quantum Solutions (PQS) Quantum Cube system, a 4-node cluster. Each node has two dual core 2.2 GHz AMD Opteron CPUs with 16 GB of memory. Nodes are configured as pairs of non-uniform memory access (NUMA) processors, with each processor having 8 GB of directly attached memory.

PQS's nodes are connected with GigE Ethernet using a single switch and single interface on each node. This connection carries all MPI data transmissions, all MPI control messages, and the traffic from the shared file system, NFS (Network File System). The cluster is small enough to provide acceptable file system performance, even with the use of NFS.

Each node also has about 800GB of disk space. Some of this space, which is local to the cluster's head node, is dedicated to cluster management software. Most of this space is provided by the nodes' local hard disks, which are not shared across the cluster. This makes it difficult to effectively manage the disk space.

Each node runs a modified version of SUSE Linux Enterprise Desktop 10 (SLED 10), which includes a set of cluster-management utilities like those provided by the Platform Rocks cluster toolkit provided on Blackpearl. These tools are exclusive to PQS systems and were not extensively studied in this research.

3.1.2 Blackpearl

Blackpearl, East Tennessee State University's (ETSU) principal computer cluster, is a Dell blade system with 60 nodes. Each node has a dual core Intel Xeon 5160 CPU, running at 3

GHz with hyperthreading enabled. The head node has 8 GB of memory and 8 GB of swap space. Each compute node has 4 GB of memory and 4 GB of swap space. All nodes are connected by a Gigabit Ethernet connection.

At the time when this work was done, Blackpearl ran Rocks 4.4, a modified version of Red Hat Enterprise Linux that supports tools for remote file and program installation, remote command execution, and cluster monitoring and management. One of these tools, Rolls, allows for Rocks to be installed and customized in a large scale, deploying and installing large packages of software on client nodes automatically and in a consistent way.

Blackpearl uses three file systems: EXT3 for intra-node file access, NFS with a shared 16 GB partition for sharing home directories, and ibrixfs, a 2.6 TB filesystem for cross-node file sharing. There is currently no formal policy regarding what data should be put on ibrixfs versus shared across NFS on the home directories.

3.1.3 PQS vs. Blackpearl

Overall, Blackpearl is more powerful than PQS. Blackpearl has 120 dual-threaded CPUs to PQS's 16 CPUs. Each of Blackpearl's CPUs is also faster. While PQS has more memory per node (16 GB vs. 4 GB), Blackpearl has 240 GB of memory overall to PQS's 64 GB of memory.

Blackpearl's Intel based nodes support a Uniform Memory Access (UMA) [7] architecture. PQS's Opteron-based nodes support a Non-Uniform Memory Architecture (NUMA) [7], with each dual core CPU having direct access to 8 GB of memory.

Both systems are interconnected with Gigabit Ethernet switches. Because PQS's switches support 4 nodes as opposed to Blackpearl's 60, PQS will have a greater peak throughput per node.

PQS uses NFS as its shared file system, a classic standard [11] that has shown major limitations in terms of scaling up. The primary bottleneck to NFS has been its single server architecture. NFS's performance degrades as a function of the number of clients and the amount of additional network traffic [6]. Blackpearl runs the IBRIX Fusion filesystem on a dedicated network attached storage system.

Blackpearl's performance also depends on how well tasks can be partitioned among its nodes. This dependence on partitioning also increases the extent to which performance can depend on fast communication.

3.2 Performance Measurement Tools

3.2.1 Gprof

The Gprof profiling tool is a standard Linux utility [4]. Support for Gprof-based profiling is built into the gcc compiler and enabled with the -p flag. When profiling is enabled, gcc inserts code into a compiled program that records where the program spends its time. When a program finishes running it writes these performance metrics to an external file. This file can then be processed using gprof to get statistics for amount of time spent in certain functions, as well as call chains that show which functions call other functions.

The biggest advantage of gprof is its availability: gprof is packaged with the gcc compiler as part of the standard Linux distribution. Gprof also has a minimal impact on program run time, allowing for fast profiling. Gprof, however, is limited in the information it collects. It cannot profile individual particular lines of code, or system calls, or program executions that fail to exit normally.

3.2.2 Valgrind

Valgrind produces results that are similar to Gprof's, but in significantly more detail [13]. Rather than profiling by instrumenting the code, Valgrind simulates a code's execution, enabling it to estimate cycle counts and cache behavior. Using Valgrind, one can measure how long individual lines of code take to run. This level of detail can be used to determine which of an execution's loops use the most CPU time and which function calls take the longest. This is detailed enough to identify individual calls to functions at specific locations that gprof does not provide. Valgrind also does not require a program to be modified, only that the program be compiled with debug symbols.

Using Valgrind, however, can slow a program's execution by as much as 100 times[13]. This slowdown limits the Valgrind's usefulness to smaller datasets that can be run quickly, Because longer runs would be impractical.

3.2.3 mpiP

Gprof and Valgrind are limited to measuring CPU run time: neither code measures the runtime of code in the application itself, including time spent on tasks like disk IO and network communication. To measure the time spent in MPI calls the library mpiP was used [10]. MpiP uses special hooks built into MPI libraries to record how long individual MPI functions run and how long MPI waits for network activity. Gprof and Valgrind cannot provide this information because their architecture limits them to computational tasks. MpiP, however, only measures the time spent in MPI functions—not the time spent on computation. This means that mpiP's output must be augmented with data from other profiling tools to obtain a comprehensive view of the program's runtime.

3.3 Optimization Strategies

VASP was analyzed several times to determine its most time consuming operations. After VASP was profiled, the performance of VASP's sequential codes was optimized, followed by that of its parallel codes.

The optimization of VASP's sequential performance began with VASP's linear algebra and Fast Fourier Transform (FFT) routines, which consumed much of VASP's time. The original linear algebra routines were replaced with faster libraries optimized for Blackpearl's CPUs. Work was also done to improve the performance of the FFT routines built into VASP using prefetch directives.

Other optimization strategies included optimizing VASP's inter-process communication. This included changing VASP so that it limited group communication to a fixed number of processes and replacing some VASP processes with threads.

3.3.1 Replacing Slow Code.

The version of VASP that was installed on PQS included AMD-specific math libraries. The initial VASP port to Blackpearl used these math libraries to ensure that VASP worked correctly. The initial step to improving the performance of VASP on Blackpearl was to replace these libraries with Intel's math libraries.

3.3.2 Prefetching.

Initially, VASP's FFT routines appeared to be a good candidate for prefetch optimizations. Profiling in Valgrind indicated that FFT functions spent most of their time in a few loops. Two attempts were made to use prefetching to improve VASP performance: a first, aggressive attempt that used prefetch intrinsics as often as possible, and a second, less aggressive attempt that only used the prefetch intrinsics when entering a loop.

3.3.3 Improved Network Communications.

The original, PQS version of VASP used a fairly simple call to MPI_alltoall function call to distribute arrays to the nodes. MPI_alltoall accepts as input two N-element arrays, where N is the number of active MPI processes. MPI_alltoall distributes these values over the network, sending element i in the first array to the MPI process with rank i, and receiving element j in the second array from the process of rank j. This communication strategy scaled well for up to 16 processes. When it was moved to Blackpearl, the use of more than 17 CPUs degraded wall clock performance.

VASP was recompiled to use a different algorithm for its communication. This alternative algorithm subdivides the global communicator into different subgroups. Specifically, the algorithm divides an x-process network into m groups of n processes, where n is configured at compile time and m is

$$\text{ceil}(x/n)$$

Communication is managed as a sequence of m+1 rounds: m rounds involving communication amongst individual groups of processes, one group at a time, followed by a final round where one process in each group communicates with one process in each of the other groups. This strategy reduces the number of processes that communicate, keeping the program from overwhelming the network.

3.3.4 Converting Processes to Threads.

The initial build of VASP allocated one process per PE. This is the simplest way of deploying a code in a cluster environment. Each process had its own memory space and all communication, even between PEs on the same node, used the network. While this is effective

for a few nodes, increasing the number of nodes and MPI processes increases overhead from MPI communication.

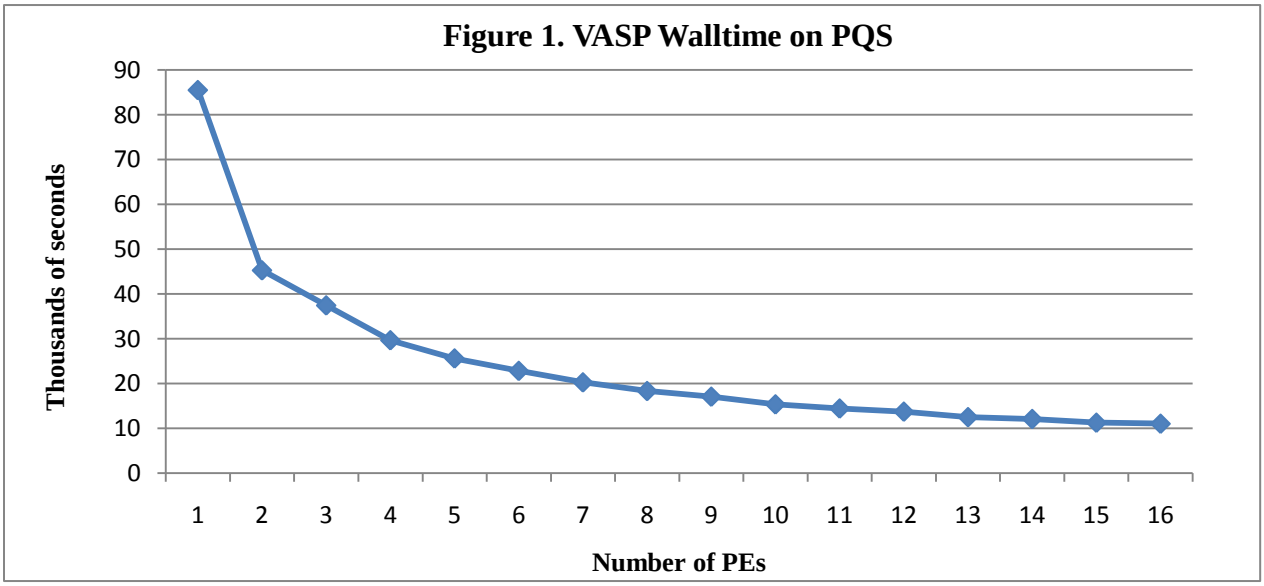
One approach to reducing MPI overhead is to alter VASP to spawn threads to manage local CPUs rather than doing so through MPI. The first approach for threading VASP used here involved threading concurrent iterations of VASP loops. The second used parallel libraries instead of sequential libraries. The Intel BLAS libraries were the first target, because a substantial amount of compute time was done in the BLAS routines and making them parallel only involved a new link procedure.

CHAPTER 4

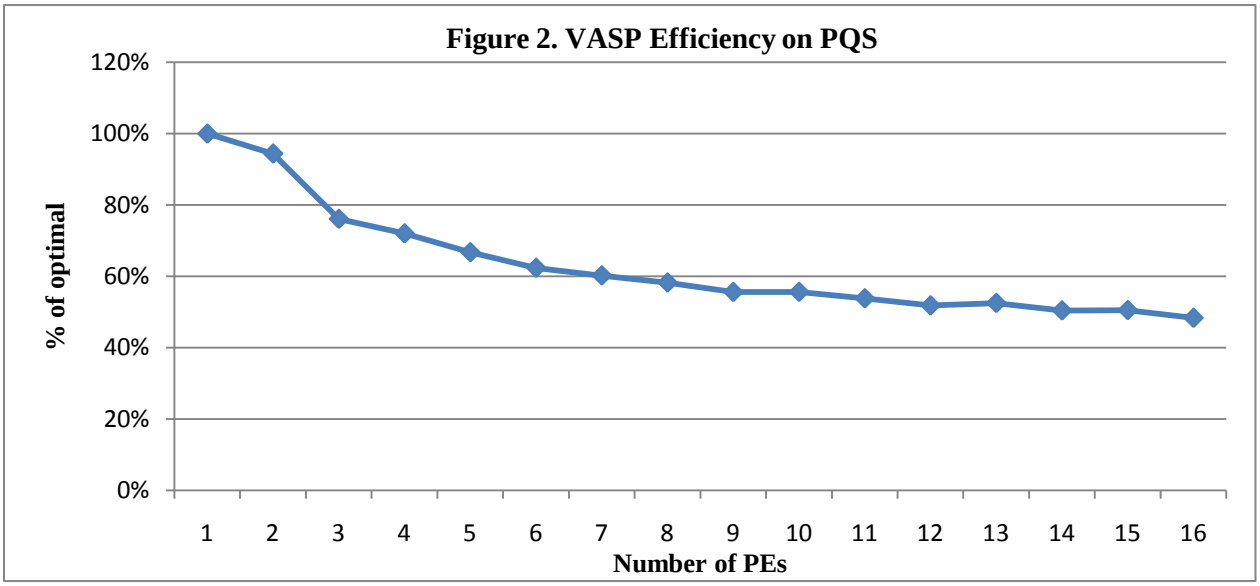
DATA

4.1 Initial Benchmark on PQS

VASP was benchmarked on PQS using Hg bench, an input included with VASP. The chart below shows wall clock time as a function of the number of CPUs.



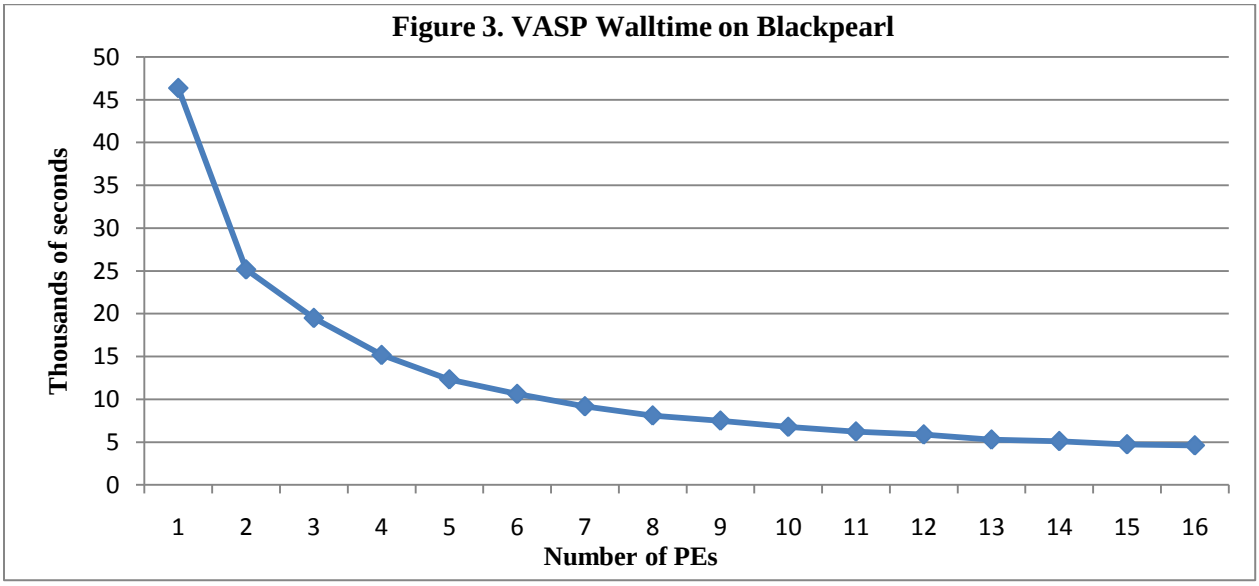
VASP's efficiency relative to the number of processors in use was then computed, using the formula (2) in section 2.1.2.



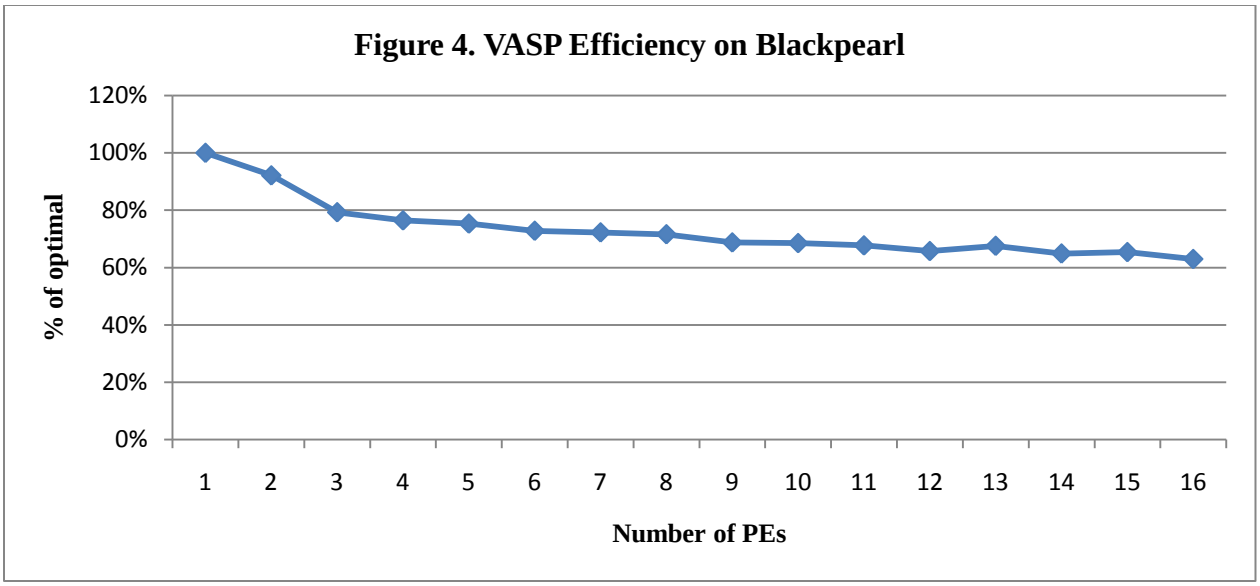
The marked drop in efficiency from 2 to 3 processors is likely due to PQS's architecture. PQS, a NUMA architecture, consists of 4 nodes each with 2 dual core Opteron CPUs each, where each dual core CPU has access to its own separate sets of memory for a total of 16 PEs.

4.2 Initial Blackpearl Benchmarks

VASP was initially benchmarked on Blackpearl with Hg bench, without hyperthreading technology. The nodes were restricted to using the two real cores available per node.



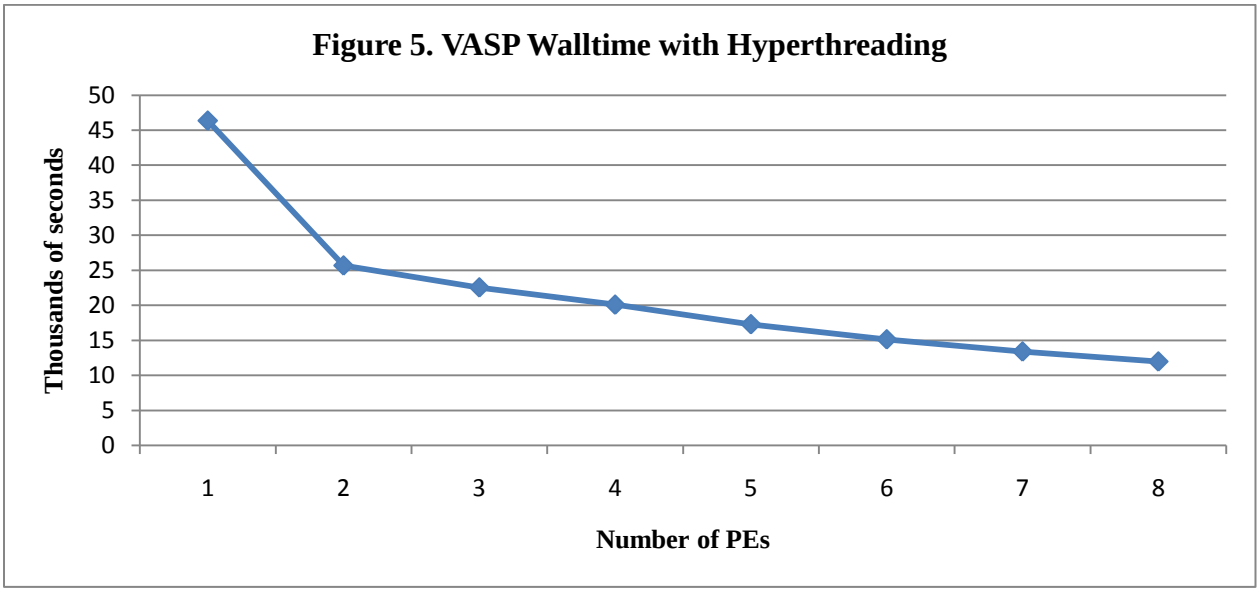
In general Hg bench ran twice as fast on Blackpearl as on PQS. This speedup can be attributed primarily to Blackpearl's faster CPUs. The efficiency numbers also show improvement.



Blackpearl also has a higher efficiency: at 16 nodes it runs at 60% of the optimal speed compared to PQS's 48%.

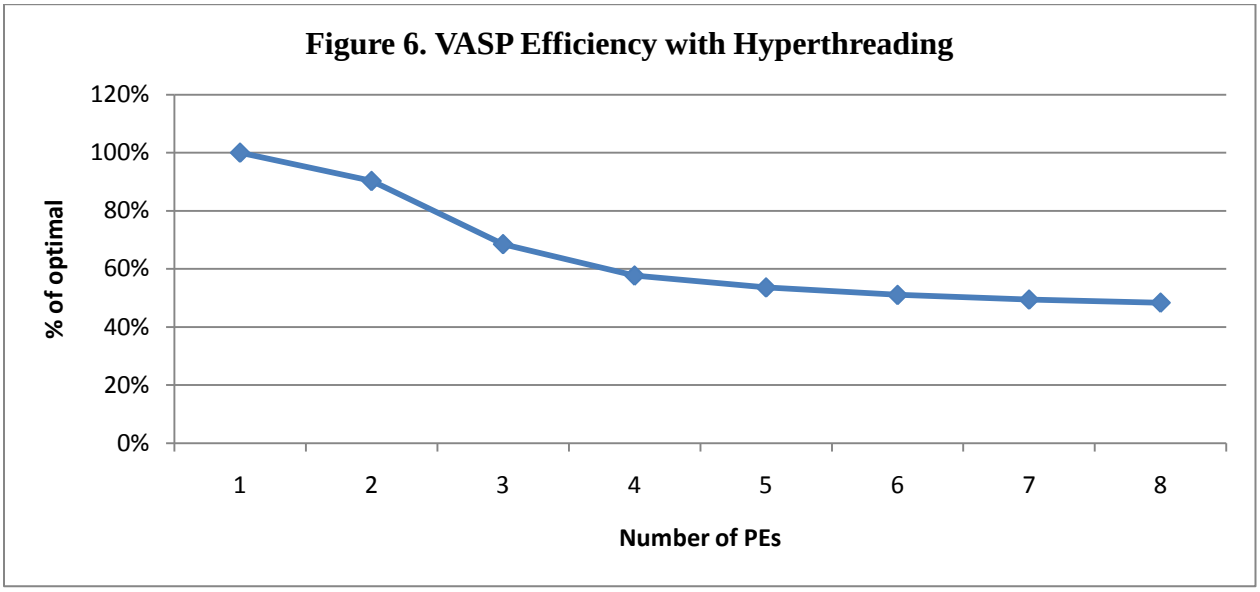
4.3 Blackpearl with Hyperthreading

Running Blackpearl with hyperthreading produced the following execution times:



Blackpearl's CPU scheduler assigned tasks to real PEs first before scheduling tasks on the hyperthreading PEs. This policy resulted in a huge drop in run time as is seen in the diagrams

shown in 4.2 after the second PE was added, but little after the third and fourth. The efficiency numbers reflect this.



It is interesting to note the lack of major drops in efficiency following the addition of the 5th PE. The addition of this 5th PE required the addition of a second node, thereby incurring networking costs.

4.4 Profiling VASP with Gprof

After the initial benchmarking VASP was rebuilt with the gprof option enabled. This provided more detailed information about where VASP was spending its execution time. VASP was run on four PEs and the results were combined into a single dataset.

Table 1. Gprof Data

% time	cumulative seconds	self seconds	Calls	Self Ksec/Call	Total Ksec/Call	Function name
20.99	8548.17	8548.17	835,081,188	0	0	fpassm_
15.66	14922.9	6374.72				zmmkern32pack_
10.12	19044.72	4121.82	342,546,198	0	0	ipassm_
6.95	21872.86	2828.14				dmmkernt_
5.77	24221.34	2348.48	2,015,051	0	0	fftwav_
5.63	26515.64	2294.3	424	0.01	0.03	rmm_diis_mp_eddrmm_
3.91	28107.04	1591.4	4,036,406	0	0	map_forward_

Table 1. (continued)

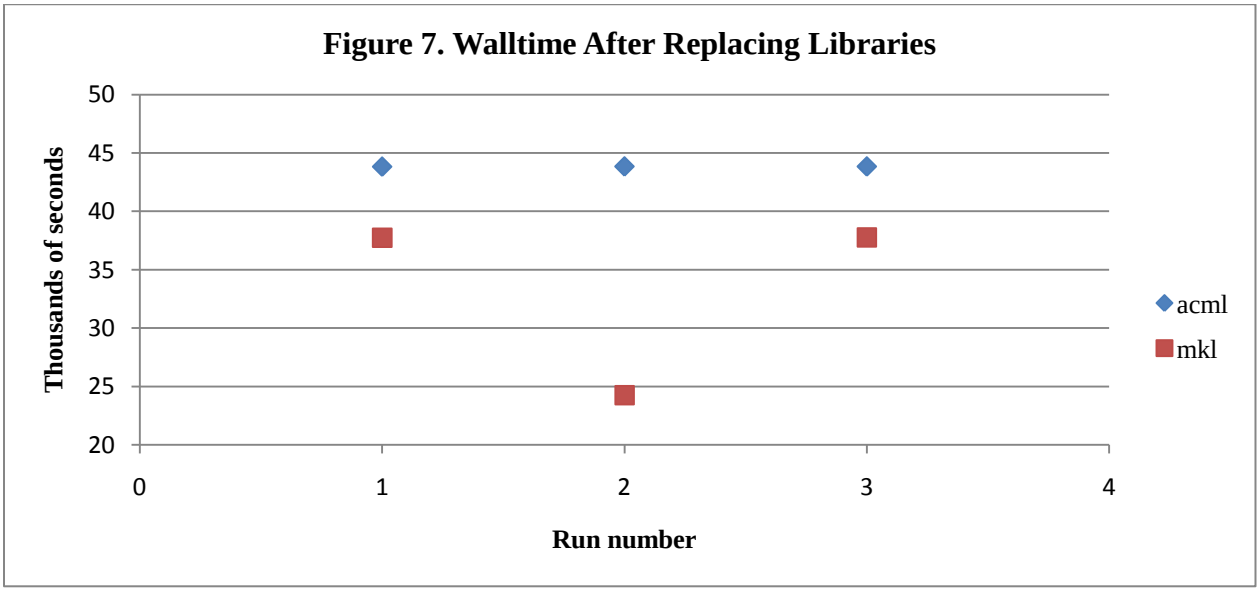
2.7	29204.49	1097.44	1,062,622	0	0	hamil_mp_eccp_
2.65	30282.62	1078.13	123,043	0	0	hamiltmu_
2.27	31205.24	922.62				zgexblkmatn_
2.26	32123.41	918.17				dmmkern_
2.01	32939.82	816.41	2,022,423	0	0	map_scatter_
1.54	33566.14	626.32	347,680	0	0	vhamil_
1.12	34023.86	457.72	286,843	0	0	nonlr_mp_rpromu_
1.1	34470.89	447.03				zgeblk2matn_

The profile data omits functions that account for less than 1% of VASP's runtime, along with call data for library functions, including call counts.

4.5 Replacing Libraries

The initial profiling work used a version of VASP that was compiled with the PQS math libraries. Replacing the AMD math libraries (ACML) with Intel math libraries (MKL) yielded speed gains of 16%, 80%, and 16%, as shown below.

Figure 7. Walltime After Replacing Libraries



The cause of the second run's huge gain is unknown. Most likely, this is the result of VASP making an exceptionally lucky guess.

4.6 Profiling with Valgrind

VASP was then profiled with Valgrind, which was about 10 times slower than Gprof. This provided a detailed characterization of function operation, including counts of individual instruction execution and cache usage data.

Table 2. Valgrind Instruction Count and Cache Behavior

Ir (millions)	Dr (millions)	Dw (millions)	I1mr	D1mr	D1mw	I2mr	D2mr	D2mw	Function
163,974	107,158	5,288	74,494	381,397,167	397,573,573	74,494	180,127	367,846	fpassm_
81,158	54,416	5,116	136	77,564,201	38,827,298	136	19,407,582	38,792,439	wave_mp_wfinit_
6,930	5,157	484	144	27	20,145,550	144	27	20,145,550	wave_mp_allocw_
6,189	1,620	775	15	152	13	15	152	13	rane_
5,875	2,744	309	2,305,347	85,361,450	3,306,151	486,604	44,750,392	11,815	0x000000000087E710
4,700	2,970	473	844	293	19,726,906	823	92	19,726,045	MAIN__
1,894	411	167	54,735	53,194,765	20,991,342	10,740	30,801,380	793,969	rpro_
1,067	69	380	732	7,216,566	48,450,969	732	7,214,877	41,014,654	fftwav_
772	444	43	242	4,472	119,361	137	3	119,272	gen_index_
756	315	164	78	6,651,754	13,144	78	0	4,728	mgrid_mp_gen_rc_sub_grid_

Fields in this table that start with 'I' and ‘D’ characterize operations involving instructions and data, respectively. The 'r', 'w', ‘mr’, and ‘mw’ designations refer to reads, writes, misreads, and miswrites. Misreads are reads where the cache lacks the requested data, with 1 meaning Level 1 cache and 2 meaning Level 2 cache. Miswrites are writes where cache does not have the requested information to update.

4.7 Prefetch

Manual prefetch directives were added in an attempt to reduce cache misses for the function fpassm. To measure prefetch run times a base time was compared to subsequent runs where groups of prefetch instructions were inserted. Valgrind was used to locate a specific loop that saw the most cache misses. This loop looked to be two loops for performing FFTs in VASP.

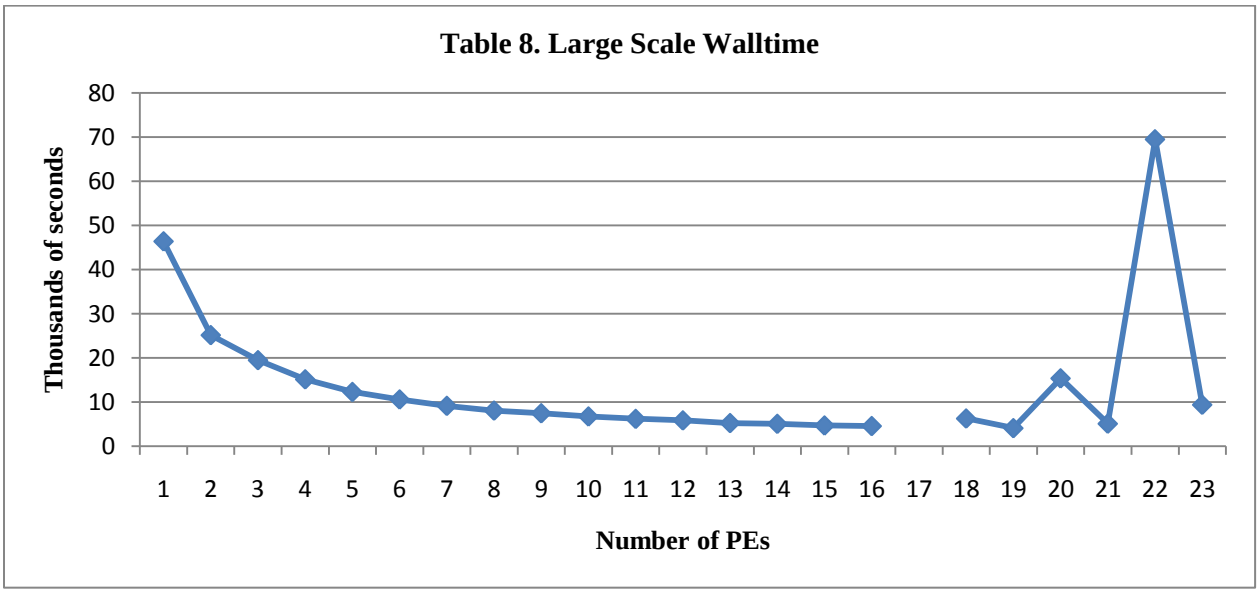
Initially the prefetching was done for both loops in fpassm for the FFT algorithm. Subsequent runs were made with prefetching done either for the innermost loop or only for the outermost loop.

Table 3. Modified Prefetch Behavior Walltime

	Seconds	% run time (compared to base)
Base	45,678.547	100%
All inner and outer	56,877.301	125%
First time in outer loop	45,793.243	100.3%
First time in inner loop	46,058.19	100.8%

4.8 Scaling Up to More PEs

The next step to improving VASP's performance was to measure how VASP scaled up beyond the original 16 nodes for which it was originally optimized for.



Data point 17 is missing from the results because VASP terminated for unknown reasons. Data for PEs above 23 were not measured because runs were taking over 24 hours (86,400 seconds).

4.9 mpiP

The use of additional PEs was causing VASP to run considerably slower. To determine the effect of additional nodes on performance, VASP was compiled and executed with mpiP, a library that generates MPI performance statistics. The interesting data is excerpted below.

Table 4. mpiP Task Data

Task	AppTime	MPITime	MPI%
0	4.42E+03	3.04E+03	68.69
1	4.42E+03	3.06E+03	69.05
2	4.42E+03	3.10E+03	70.14
3	4.42E+03	3.09E+03	69.86
4	4.42E+03	3.03E+03	68.46
5	4.42E+03	3.03E+03	68.57
6	4.42E+03	3.07E+03	69.37
7	4.42E+03	3.15E+03	71.3
8	4.42E+03	3.04E+03	68.6

In this table 'Task' is the MPI process's rank, 'AppTime' is the wall time, MPITime is the total time the program spent in MPI functions and MPI% is the percentage of the time the program spent in MPI functions compared to the walltime. MpiP also notes where MPI functions were called from and how long each takes.

Table 5. mpiP Function Data

Call	Site	Time	App%	MPI%	COV
Waitall	33	1.37E+08	51.62	74.58	0.01
Waitall	38	1.50E+07	5.67	8.19	0.01
Waitall	10	1.22E+07	4.61	6.66	0
Isend	33	3.52E+06	1.33	1.92	0.01
Waitall	9	3.13E+06	1.18	1.7	0.17
Waitall	40	3.01E+06	1.13	1.64	0.31
Allreduce	50	2.04E+06	0.77	1.11	0.26

This table, 'Call', shows the mpiP profile for an MPI function. Site numbers are mpiP-assigned identifiers for instances of MPI functions. 'Time' is the aggregate time spent in the function. 'App%' is the percentage of the total application time spent in the function while 'MPI%' is the percent of the MPI time in the function. 'COV' is the coefficient of variation

between the processes, indicating how much each process varied from each other in that function call.

Table 6. mpiP File and Function Data

ID	Lev	File/Address	Line	Parent_Funct	MPI_Call
33	0	mpi.f90	1176	M_alltoall_z	Irecv
38	0	mpi.f90	1316	M_sumf_d	Irecv
10	0	mpi.f90	1328	M_sumf_d	Irecv
9	0	fftmpi_map.f90	415	map_backward	Irecv
40	0	fftmpi_map.f90	361	map_forward	Irecv
50	0	mpi.f90	1410	M_sum_d	Allreduce

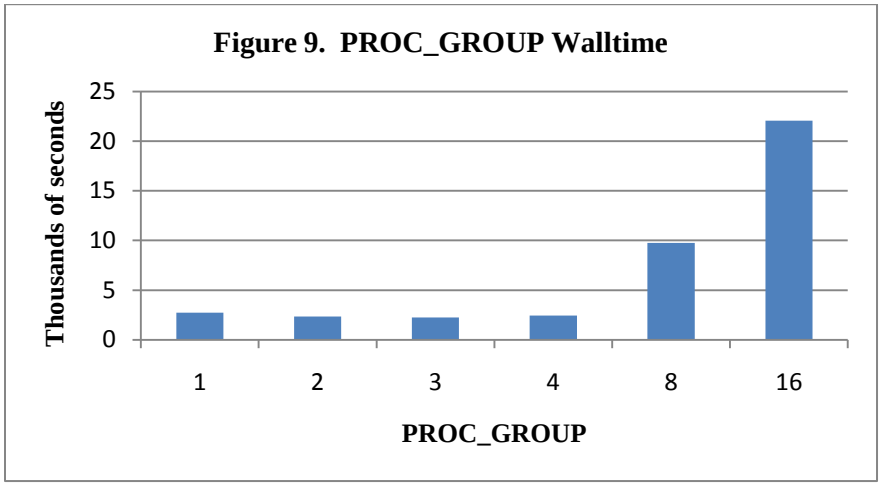
This table relates mpiP function IDs to source codes. 'Lev' is a field identifying Pcontrol Level, an advanced function of mpiP that was not used. 'File/Address' has either the file name or the function address if the file name is unavailable. 'Line' is the line number in the source code. 'Parent Funct' is the name of the function containing the MPI call. 'MPI Call' is the name of the MPI function call.

Unfortunately some of this information is not reliable. For instance, function ID 33 says that the parent function is M_alltoall_z and that it is on line 1176 of the file mpi.f90. However, line 1176 in mpi.f90 is in function M_alltoall_d. This is probably due to compiler optimizations, because the real M_alltoall_z is only a call to M_alltoall_d with some minor input parameter alterations.

4.10 Limiting Communication

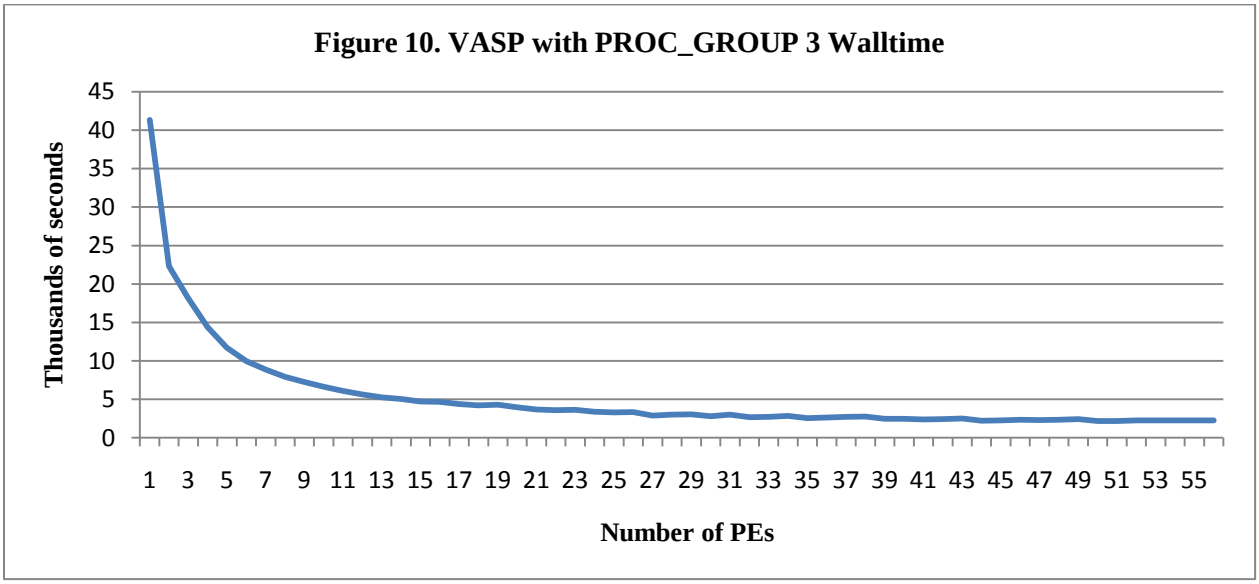
After establishing that the system's performance bottleneck was the MPI_alltoallv function, MPI_alltoallv was replaced with another function that restricts communication. The function, M_alltoall_d, exchanges the rows and columns of a square matrix that is distributed across all nodes. The M_alltoall_d function has two different built-in implementations: one that uses the MPI collective operation, MPI_alltoall and one that uses MPI_isend and MPI_irecv

pairs across groups of CPUs. Switching from MPI_alltoall to using proc groups produced dramatic improvements. The following chart was produced using 56 PEs.



As noted in section 4.8, using more than 23 PEs caused VASP to run for more than a day. With PROC_GROUP set to 3, VASP completed in about 37 minutes (2,237 seconds). The most likely cause of this dramatic increase seemed to be the reduction of collisions on the network.

With these dramatic improvements VASP was benchmarked again, going up to 56 PEs. This was done with 2 PEs on each node, eschewing the use of the hyperthreading option. Two runs were done and the chart below is the average of the two runs.

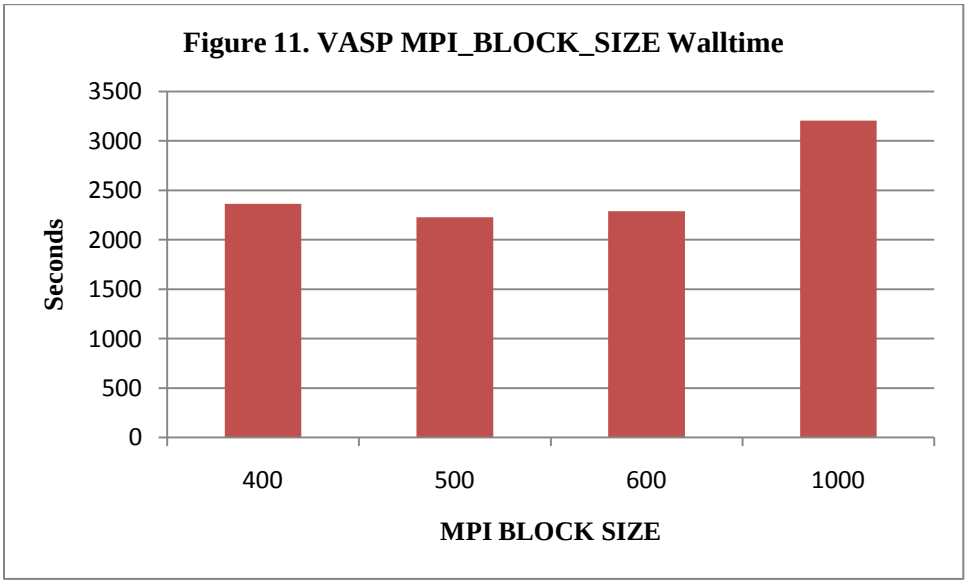


While the speed gained from adding additional PEs diminishes quickly after about 30 PEs, at no time does the speed get notably worse. After these benchmarks were complete VASP was benchmarked using hyperthreading for 112 PEs. This benchmark run took 7,145 seconds, compared to the non hyperthreading 56 PE run which took 2,272 seconds. This indicated that PROC_GROUP greatly reduced the networking inefficiency, but not to the degree where hyperthreading can be used without losing all performance gain to the network cost.

4.11 Adjusting MPI Block Size

VASP allows the maximum size of MPI communications to be set when VASP is compiled. By default, VASP uses the number of double precision floating point elements as the measure of MPI communication size. VASP was benchmarked at the default 500 elements and with 400 element, 600 element and 1,000 element sizes.

Figure 11. VASP MPI_BLOCK_SIZE Walltime

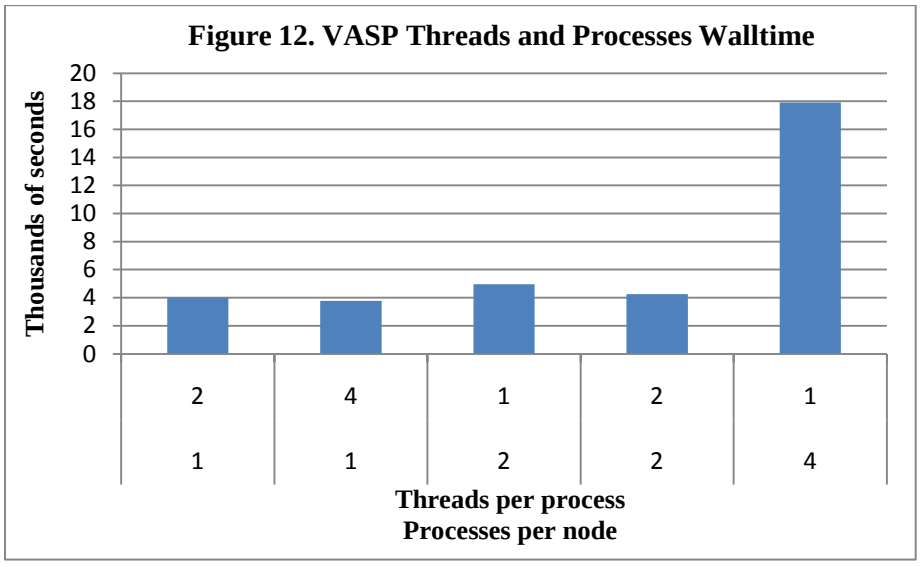


The default size proved to be the best size. Changing the size of MPI messages only seemed to hurt performance.

4.12 Threading

The original strategy for threading VASP was to alter VASP's FFT algorithms (ipassm and fpassm) to use OpenMP directives in the main loops computing the FFT. This approach produced versions of VASP that crashed due to memory access errors attributable to an incomplete knowledge of the algorithms being used. The second approach was to replace the FFT algorithms in VASP with the FFT functions found in the Intel math library (Intel MKL), which supports the use of threads to compute FFTs.

With VASP rebuilt to use threaded routines for the most computationally intensive tasks VASP was rebenchmarked on 29 nodes. These benchmarks use a more complex molecular model than previous benchmarks to make any speed increases more pronounced. Because an additional node was available at the time these benchmarks were done, they were run on 29 nodes instead of the 28 nodes used previously.

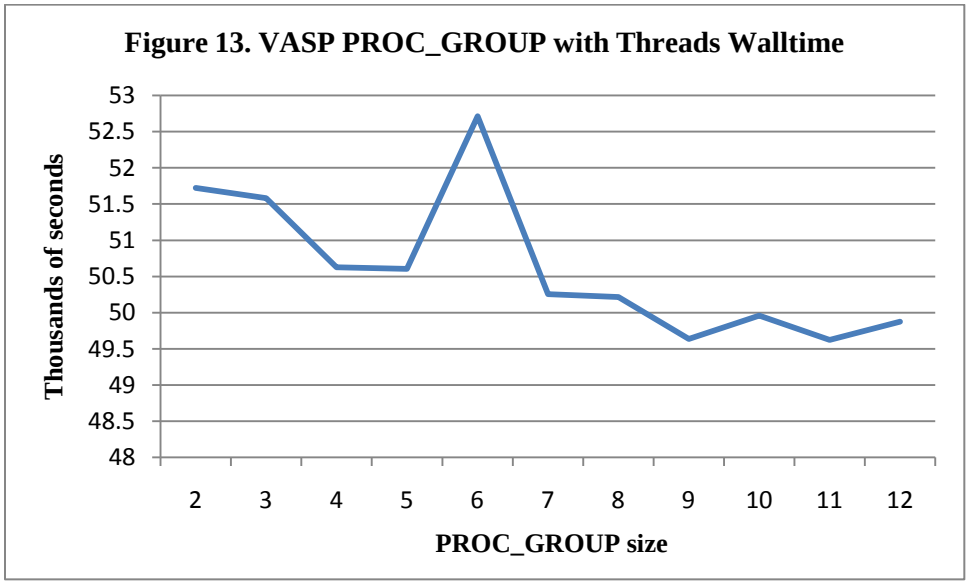


The X axis shows the number of threads a VASP process spawns above the number of processes MPI assigns to each node. The runs not using hyperthreading used 58 PEs, such as in

the first and second bars on the chart. The runs using hyperthreading ran on 116 nodes, as shown in the chart's other three bars.

4.13 Readjusting Communication Group Size

After reducing the number of processes communicating in VASP the PROC_GROUP setting was retuned for the new environment.



This chart shows that a larger PROC_GROUP, after VASP was modified to use threads to exploit additional PEs, were beneficial.

CHAPTER 5

ANALYSIS

The initial transfer of VASP source code from PQS to Blackpearl VASP made VASP 84% faster running on a single PE, using the same libraries and compilers. The figures in section 4.1 and 4.2 show that this trend continues up to 16 PEs, the maximum on PQS. The chart in section 4.3 indicates that hyperthreading was detrimental, causing VASP to execute 25% slower when using the PEs in two nodes.

Improving VASP at the source code level proved more difficult than anticipated. Attempting to make substantial changes to the program's source code or its algorithms usually resulted in runtime errors. For example, the attempts to use OpenMP to convert the FFT functions built into VASP never succeeded. The prefetch code was at best negligibly slower.

When computing FFTs VASP uses a pair of loops, an inner loop and an outer loop. The first prefetch strategy that was an attempt to prefetch data for the entry into the outer loop and the entry into the inner loop. Because this first attempt yielded runtimes that were substantially worse than the original, a second attempt was made, using prefetch directives to only prefetch the first iteration of the loop. This was done on the assumption that the CPU cache may fetch the next loop iteration after accessing the first. This was done first for the outer loop and then the inner loop.

As seen in the chart in section 4.7, all prefetching strategies resulted in slower execution. The Valgrind data in section 4.6 suggested that some cache optimization in the functions fpassm and ipassm would improve performance. The exact cause of this slowdown is difficult to determine, as documentation on CPU cache behavior is difficult to find. A reasonable

assumption is that the manual prefetch instructions interfered with the normal cache behavior, introducing additional memory and cache contention.

Most gains were realized from making better use of the existing codes and supporting their operations with faster libraries. As seen in the chart in section 4.5, using the Intel MKL libraries for BLAS produced a 16% improvement in speed. The AMD libraries, most notably zmmkern32pack_ and dmmkernt, accounted for more than a quarter (28%) of the profiled run time as shown in the gprof data in section 4.4. About 31% of the run time was consumed by ipassm and fpassm, which are integer and floating-point versions of an algorithm to perform 1-D FFTs on multiple datasets.

The chart in section 4.8 showed that VASP was having trouble scaling up after 17 PEs. An analysis of the sequential execution data indicated that VASP's use of the network needed to be analyzed. The mpiP data in section 4.9 showed that the MPI network communication was taking up 70% of the VASP walltime. The same data also showed that almost 75% of this MPI time was spent in a single function: an MPI barrier meant to ensure that all processes complete an action before continuing. This meant that something in that function was stalling the rest of the program.

Moving away from the MPI collective operations and restricting the maximum number of processes with which each process ordinarily communicates proved vital to efforts to scale up. The data in section 4.10 shows that restricting VASP nodes to communicating with 3 nodes at a time was optimal. The resulting speed up was probably the result of reduced network contention. Limited experimentation with block sizes following this adjustment suggested that the default MPI message size of 500 bytes per block is probably optimal for VASP.

VASP, when launched as an MPI job, is normally launched as a set of processes. By default, one process was launched for each PE. For example, when running on 56 PEs, 2 VASP processes would be launched on 28 nodes. The default procedure for using hyperthreading on 28 nodes was to launch 4 VASP processes on 28 nodes. This strategy created jobs that used MPI communication primitives to exchange information between processes on the same node.

To improve communication efficiency, VASP was rebuilt to use threaded routines for the BLAS functions and ipassm and fpassm were replaced with the FFT routines in the Intel MKL. In the chart in section 4.12, the bar in the middle, which shows the effect of running 1 thread per process and 2 processes per node, is comparable to previous benchmarks, except that the run used 58 PEs instead of 56 PEs. This execution completed in 4,972 seconds. Using 2 threads per process and 1 process per node, VASP ran in 4,012 seconds: that is, in 80.7% of the time.

When using hyperthreading VASP ran with 116 PEs. This was done with three configurations, one where there was 1 process per node with 4 threads, one with 2 processes per node, and 2 threads per process, and 4 processes per node with a single thread per process. With 1 process per node VASP ran in 3,776 seconds. With 2 processes per node VASP ran in 4,255 seconds. With 4 processes per node VASP ran in 17,915 seconds.

Because the Intel MKL was used both BLAS and FFT routines, using the threaded version boosted the performance of both groups of routines. While it would be possible to isolate individual gains from threading each of these routines, this data is not relevant to an analysis of VASP as a whole.

CHAPTER 6

CONCLUSION

When VASP was first moved to Blackpearl the main focus for improving performance was to measure and refine the single threaded performance of each VASP process. Replacing the BLAS libraries improved performance. Inserting manual prefetch instructions reduced performance. The gains from improving the performance of individual VASP processes were not very large compared to the challenges of scaling up to use more PEs. Reducing the number of nodes that communicated simultaneously allowed the application to effectively exploit more PEs. Using threads to use all of a node's PEs resulted in the most substantial gains in performance.

In the future, more research into the most efficient communication patterns would seem to be the best course to achieve additional gains in performance. One idea is to implement self-tuning collective operations that can adjust their method of communication to optimize performance. The fact that the MPI standard has an operation that is precisely what is needed but unusable due to poor performance seems counterintuitive.

REFERENCES

[1] Krste. Asanovic, Rastislav Bodik, Bryan Catanzaro, Joseph Gebis, Parry Husbands, Kurt Keutzer, David Patterson, William Plishker, John Shalf, Samuel Williams, and Katherine Yelick. "The landscape of parallel computing research: a view from Berkeley." Technical report, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2006.

[2] C. Gordon Bell, "Multis: A New Class of Multiprocessor Computers", Science, Vol. 228, 1985, 462-467.

[3] J. Furthmueller, J. Hafner, G. Kresse, "*Ab initio* calculation of the structural and electronic properties of carbon and boron nitride using ultrasoft pseudopotentials", Physical Review B 50, 1993, 15606.

[4] GNU Gprof, http://sourceware.org/binutils/docs-2.20/gprof/index.html (retrieved May 2010).

[5] G.H.Golub, C.F.Van Loan. Matrix Computation, 2nd ed. Baltimore, 1989.

[6] James Hall, Roberto Sabatino, Simon Crosby, Ian Leslie, Richard Black, "Counting the Cycles: a Comparative Study of NFS Performance over High Speed Networks," lcn, pp.8, 22nd Annual IEEE International Conference on Local Computer Networks (LCN'97), 1997.

[7] J. L. Hennessy and D. A. Patterson *Computer Architecture: A Quantitative Approach: 4th Ed.* Morgan-Kaufman Publishers, San Mateo, CA, 2006.

[8] P. Hohenberg, W. Kohn, "Inhomogeneous Electron Gas", Physical Review 136, 1964, B864-B871.

[9] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard Version 2.2", http://www.mpi-forum.org/docs/mpi22-report/mpi22-report.htm, September 2009 (retrieved May 2010).

[10] "mpiP: Lightweight, Scalable MPI Profiling", http://mpip.sourceforge.net/, April 2010 (retrieved May 2010).

[11] Bill Nowicki, "NFS: Network File System Protocol Specification," http://www.ietf.org/rfc/rfc1094.txt, March 1989 (retrieved July 2010).

[12] Gavin J. Pringle, "Optimisation of VASP on HPCx" http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0414.pdf, (retrieved July 2010).

[13] "Valgrind User Manual", http://valgrind.org/docs/manual/manual.html, August 2009 (retrieved May 2010).

APPENDIX

SELECTED TERMS

PE: A PE, often also called a CPU, is the individual computing element. In recent times individual processors are engineered with multiple 'cores'. Multicore technology allows a computer with a single socket to have multiple PEs and thus run multiple programs at the same time. This thesis treats a processor with two cores as equivalent to two processors with a single core: i.e., as two PEs. Intel hyper-threading will cause each PE to look like two PEs. For the purpose of this thesis, each virtual PE is counted as a PE, except when noted otherwise.

FFT: A **fast Fourier transform** (**FFT**) is an efficient algorithm to compute the *discrete Fourier transform,* a mathematical mapping that is used by the VASP program to evaluate some of the basic quantities involved in DFT simulations.

Hyperthreading: A feature of certain Intel CPUs, formally called hyperthreading technology. Special hardware allows an individual PE to present itself to the OS as two virtual PEs, theoretically allowing the OS to better use the single physical PE.

Node: A node is an individual computer, physically separated from the other nodes and connected by a network.

Compiler Intrinsic: Often shortened to just 'intrinsic' is a function that is part of the compiler, rather than part of a library. For example, the Intel compiler has an intrinsic for prefetching, where the programmer inserts what looks like a normal function call which the compiler will replace with assembly instructions for prefetching data.

VITA

MATTHEW BAKER

Personal Data:	Date of Birth: August 16, 1983
	Place of Birth: Memphis, Tennessee
	Marital Status: Single
Education:	High school, Franklin Road Academy
	B.S. Computer Science, East Tennessee State University, Johnson City, Tennessee 2007
	M.S. Computer Science with concentration in Applied Computer Science, East Tennessee State University, Johnson City, Tennessee 2010
Professional Experience:	Teaching Assistant, East Tennessee State University, College of Business and Technology, 2007-2009