# A STUDY OF THE HYBRID PROGRAMMING PARADIGM ON MULTICORE ARCHITECTURES

by

Ashay Rane

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

ARIZONA STATE UNIVERSITY

December 2009

A STUDY OF THE HYBRID PROGRAMMING PARADIGM

ON MULTICORE ARCHITECTURES

by

Ashay Rane

has been approved

November 2009

Graduate Supervisory Committee:

Daniel Stanzione Jr., Chair
Donald Miller
Partha Dasgupta

ACCEPTED BY THE GRADUATE COLLEGE

ABSTRACT

With multicore processors making their presence felt in all forms of computing, there is an increased focus towards using parallelization for getting higher performance. Using the Message Passing Interface (MPI) standard and the Open Multiprocessing (OpenMP) specification, this work relates the design of modern-day clusters to the hybrid MPI/OpenMP parallel programming model. This programming model combines message passing (using MPI) and threading (using OpenMP) at the right levels in the design hierarchy. Through distinct but cohesive application kernels, called dwarfs, it is demonstrated that using the hybrid programming paradigm can prove to be an advantage for applications that exhibit certain characteristics. These characteristics revolve around the pattern of communication exhibited by the application. It is observed that the impact of hybrid optimizations gets more prominently visible when using higher core counts. This means even though hybrid optimizations are mostly concerned with intra-node optimizations, their effect is visible across node boundaries.

*This work is dedicated to:*

*my parents, Suhas and Meena Rane*

*and to "friends" turned "family":*

*Gayatri, Navaneet, Rads,*

*Shilpette and Swaroop*

ACKNOWLEDGEMENTS

I would like to acknowledge the enthusiastic supervision of Dr. Daniel Stanzione during this work. I thank Dr. Donald Miller and Dr. Partha Dasgupta for being on the committee and helping with guidance and relevant discussions. Members of the High Performance Computing Initiative (HPCI) research group are thanked for the numerous stimulating discussions, help with the setup and general advice; in particular I would like to acknowledge the help of Gil Speyer, Scott Menor and Doug Fuller for their support. Nathan Kerr and Anthony DiGirolamo are thanked for their assistance with all types of technical problems & discussions and Marisa Brazil for making administrative tasks easy.

I am grateful to my friends for being the surrogate family during the years I stayed here at Arizona State University and for their continued moral support.

Finally, I am forever indebted to my parents for their understanding, endless patience and encouragement when it was most required.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

# Chapter 1: Introduction

## 1.1 The Multicore Paradigm

It is a well-accepted notion that the design of future processors will be dominated by multicore (possibly extended to many-core) technology[32, 22, 9]. Application development can thus no longer depend on new single-core processors to automatically make applications run faster with each generation[31]. Until recently, once an application was coded, it's improved performance was assured. Newer processors with faster clock rates would automatically make the applications run faster. However, manufacturers have moved from producing new chips with higher clock rates, and instead now produce multicore chips. The prime reasons for switching to multicore processors are limits in exploiting Instruction-Level Parallelism (ILP), memory bandwidth limitations and high power consumption in single-core processors.

The concept of multicore processors is to continue Moore's law style performance improvement at the socket level. However, this does not necessarily mean there are performance improvements at the core level. In fact, in most cases, the per-core performance is lower than the single core processors of the previous generation, due to lower clock rates. This is attractive to chip makers, and some consumers, as you get an improved performance per dollar of operating cost. For instance, two cores at 80% of the single core clock speed deliver a peak performance of 1.6 times the previous chip, perhaps using 50% less electrical power. While this is an improvement in peak performance, if you care about the performance of a single application thread, performance is lower. Thus if an application is programmed for making use of a single core only, then the application is very likely to run slower than on a previous generation single-core processor. The continued focus on multi-core, and the corresponding cessation of Moore's law performance increases for single threaded code, make the use of parallelization imperative to deliver application performance improvement.

Hence in order to continue to deliver the expected increase in performance in each new generation of multicore processors, codes must increasingly use parallel programming. To properly exploit multicore architectures, programming techniques must make use of the unique architectural characteristics of multicore design.

However, writing programs and optimizing them for multicore processors is complicated by the presence of a memory hierarchy and the need to program while taking this hierarchy into account, limited memory bandwidth, correctness of the program being harder to guarantee because of implicit communication among threads, non-uniform access speeds between processors and memories and the presence of multiple threading libraries each differing in their implementation and supported features. One often finds that these threading libraries need to be complemented with additional tools for optimization of application programs. This work is a study of a specific parallel programming paradigm called hybrid programming. It also looks into tools and techniques for optimizing hybrid parallel programs.

*Design Of Multicore Processors*

Multicore processors are seen in desktop and laptop computers, servers and also in High Performance Computing Systems[12]. High end servers have employed multicore technologies for a number of years. High Performance Computing (HPC) is at the extreme end of this spectrum where multiple "servers" are used. Hence parallelism is not just exploited within a multi-core processor but also among the "nodes" or the computers. Many modern-day clusters in High Performance Computing are built using more than one multi-core processor inside each such node.

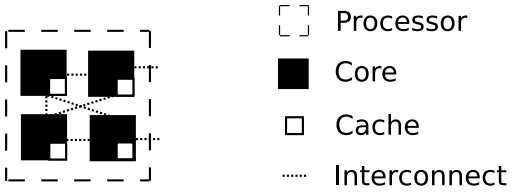



Figure 1.1. A Quad-Core Processor

As the name suggests, multicore processors are those that contain more than one processing core in a single package, as illustrated in figure 1.1. Each core is an independent execution unit, in that it contains an Arithmetic & Logic Unit (ALU), registers, cache and I/O capability. To balance cost and performance, higher levels of cache (Level 2, Level 3) are shared amongst all or a subset of the cores[11]. In the following text, a 'processor' is assumed to be multicore. Today's server class processors have 2, 6, or 8 cores with more cores per processor expected in the next year. Certain specialized processors have even greater number of cores (e.g. GPGPUs contain 128 cores or more).

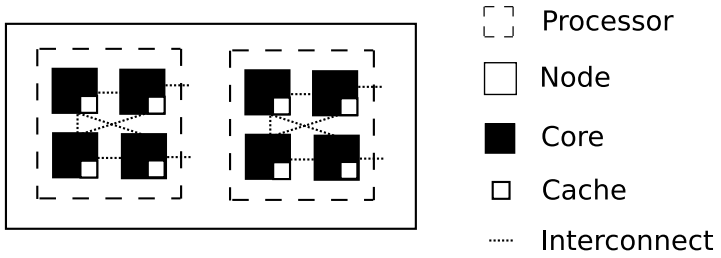



Figure 1.2. A Node Containing Dual Quad-Core Processors

The design of such multicore processors is influenced by manufacturing costs. Often HPC computers are built using more than one multicore processor into one "node" or computer, with a common memory accessible from each core on the node. An 8-core node formed using quad-core processors is shown in figure 1.2. Such a design allows making use of shared memory parallelism.

For effective use of such a hierarchy, the programming paradigm used should also be able to leverage the benefits/advantages of each level in the hardware hierarchy. In the world of HPC, the de-facto standard for distributed memory programming is MPI (the Message Passing Interface)[8] while OpenMP[25] (specification for shared memory parallel programming) is the standard way of programming shared memory systems. MPI provides an explicit messaging model with no assumption of shared memory, and as such is the standard for use on distributed memory systems. OpenMP provides a threading model, with implicit communication and the

assumption of shared memory, and therefore is often used on shared memory systems. Note that each of these models (pure-MPI or pure-OpenMP) assumes a flat hardware organization instead of a heirarchy.

## 1.2 Programming Multicore Processors

As can be seen from the previous section, HPC clusters are built using a hierarchy of processors and cores as shown in figure 1.2. The following are the differences in the execution units of code running at each level in the hardware hierarchy, from the point of view of communication:

(a) Between nodes: Communication occurs over the network link (Ethernet, Infiniband or Myrinet)

(b) Between processors within a node: Communication occurs via Random Access Memory (RAM)

(c) Between cores within a processor: Communication can occur via cache or via RAM

Note that `(a)` is a problem that is best solved using Distributed Memory, while `(c)` can be solved only using shared memory. Thus MPI would be best suited to `(a)` and OpenMP to `(c)`. There have been efforts[19][15] to make a single solution (either MPI or OpenMP) sufficient for exploiting both distributed and shared memory. For ease of programming and debugging, often an MPI-only approach (also referred to as a pure-MPI solution) is used, as shown in Figure 1.3. However from our experiences and as illustrated later in this document, this approach often suffers significantly lower performance than a hybrid implementation. The hybrid programming paradigm attempts to solve this problem by making use of the combination of MPI and OpenMP. An example of using hybrid MPI and OpenMP is shown in Figure 1.4.

## 1.3 Hypothesis

The hypothesis of this thesis work is:

*The hybrid MPI/OpenMP programming paradigm is a natural fit for the design of current day*

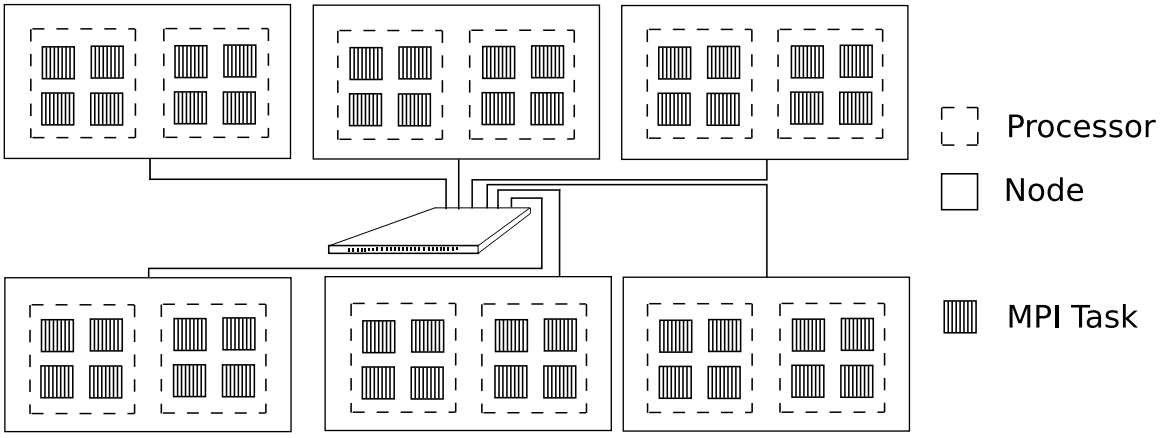



Figure 1.3. Using Only MPI For Running A Program On 6 Nodes, With 8 Cores In Each Node

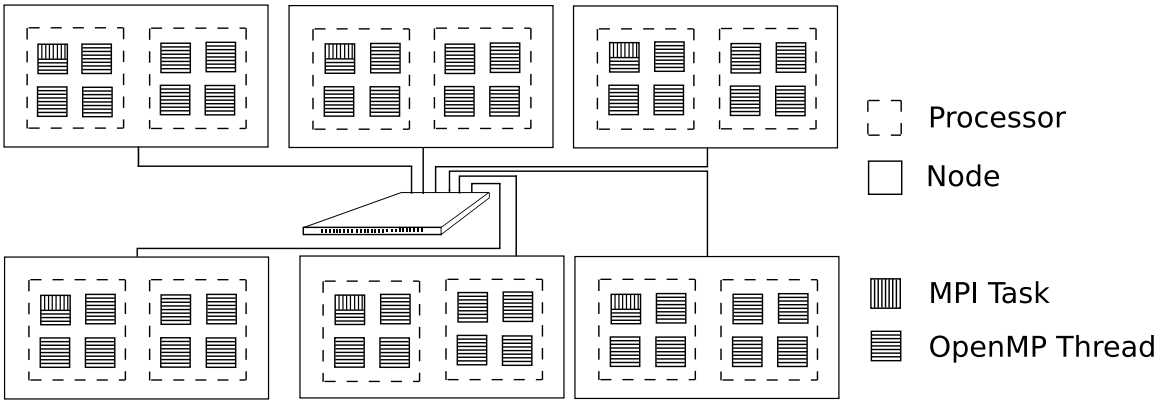



Figure 1.4. Using MPI In Combination With OpenMP To Leverage Hardware Hierarchy

*HPC clusters. Hence it is better than the pure-MPI paradigm in terms of execution speed, scalability and resource consumption.*

## 1.4 Thesis Goals

The following are the objectives of this thesis work:

- To examine tools for tuning hybrid programs along with optimization techniques specific to hybrid programs
- To demonstrate significant performance improvement using hybrid programming paradigm over pure-MPI paradigm
- To understand how hybrid MPI/OpenMP program performance differs from performance of pure-MPI program
- To write a guide for cluster users on optimizing parallel programs using hybrid programming paradigm

Tools found useful for program optimization and the comparison between pure-MPI and hybrid programs with the results of tuning them are covered in subsequent chapters. During this work, a guide was written for tuning hybrid parallel programs for HPC cluster users.

## 1.5 Methodology

A set of sample programs was chosen to optimize and demonstrate the performance improvement. Because of the importance of having a diverse set of programs that resembled real-world problems, one solution was to use the standard benchmark applications such as the Weather Research and Forecasting Model (WRF). However, these programs are too big and complex and thus would have required significant time to understand, to convert to hybrid MPI/OpenMP and then to optimize. The solution adopted was to write programs that replicate the computation/communication patterns that are found among programs commonly used in HPC. These patterns or "dwarfs" are described in[1]. These dwarfs, originally the work of Phil Colella[7] and later extended by David Patterson and others at the University of California at Berkeley, are a list of kernels which capture the typical programming patterns in scientific

Table 1.1. Dwarfs And Their Discriminating Characteristics

| **Dwarf** | **Example code** | **Characteristics** |
|---|---|---|
| Monte-Carlo / MapReduce | Computation of PI | • Strictly computation-bound kernel |
| Dense linear algebra | Dense matrix/matrix multiplication | • Point-to-point sends<br>• Can leverage shared memory for communication |
| Sparse linear algebra | Sparse matrix/vector multiplication | • Uses collective communication between master and slave |
| Structured grids | Jacobi solver | • Computation-bound kernel with "regular" access patterns |
| N-body methods | 3 dimensional gravitation problem | • "Cyclical" data transfer among processes |
| Spectral methods | 2 dimensional Fast Fourier Transform | • Communication-bound kernel, all-to-all communication |

applications. The dwarfs are widely accepted as the standard set of programs used to test parallel programming paradigms.

The dwarfs considered in this work, with their discriminating characteristics are described in Table 1.1.

The rest of this document is organized as follows. Chapter 2 explains background topics such as the Operating System support for threads, Message Passing Interface, OpenMP, hybrid programming and tools used to tune MPI/OpenMP code in detail. Chapter 3 compares the hybrid programming paradigm with the pure-MPI paradigm considering various factors. To realize the differences, different run-time characteristics like memory, inter-process messages, programming hierarchy, cache effects, etc. were observed. It also talks about the tools and techniques used in this work for tuning hybrid MPI/OpenMP programs. TAU, PAPI and `libnuma` were the tools explored for discovering optimization techniques. Chapter 4 presents the results of applying the optimizations to hybrid programs, while chapter 5 presents the concluding remarks. Appendix A talks about the implementation of multidimensional data

decomposition in OpenMP and code that was written to accomplish it. Appendix B presents the values received from the performance counters for each dwarf for varying problem sizes, on which most of the analysis was based on.

# Chapter 2: Background And Related Work

This chapter explains the standards, tools and libraries that are mentioned in the later chapters of this document. A brief section illustrates how to run hybrid MPI/OpenMP applications.

## 2.1 Operating System Support For Parallelism

Most HPC installations use Linux as their operating system. Of the TOP500[21] list for November 2008, more than 87% installations run Linux. Linux provides a highly configurable kernel that can be modified to suit individual purposes. Since HPC programs are supposed to run as fast as possible, it is important to ensure that the Operating System (OS) does not add any overhead. Hence the Linux kernel is often hand-configured to include only the required support from the OS. This not only implies removing support for unnecessary drivers for hardware devices that are not used but also eliminating certain features like virtual memory which is usually considered indispensable.

The Linux kernel tries to incorporate certain features that might be useful for running parallel programs, e.g. multicore aware scheduler, first touch policy for memory placement, etc.[29]. The multicore aware scheduler is aware of multiple cores sharing last level caches and hence is capable of making decisions concerning performance and power utilization. The first-touch policy ensures that memory pages are placed close to the socket containing the core that first accessed the page. This is done with the expectation that further references to this page would be made mostly by the same core, or another core from the same socket, thus resulting in faster access. However due to the wide variety of programs that might be executed, it is difficult for the kernel to provide features that accelerate the performance of all applications. Instead of favoring any particular class of applications, the kernel exposes the underlying variables which can be then used by higher-level applications. As an example, when configured with the `CONFIG_NUMA` option, the Linux kernel allows applications to access system-level information such as number and organization of cores, processors and sockets on the node, “distances” between sockets (in terms of time to communicate), CPU operating frequency, etc. It also allows applications to set CPU affinity and memory placement for

threads or processes. Another example is the `CONFIG_MIGRATION` that allows physical pages to be migrated from memory belonging to one socket to another socket's memory.

In other cases, even the above explained level of support proves to be incomplete. For example, modern microprocessors include special purpose registers called hardware performance counters that can be used to store hardware related events such as L1/L2/L3 cache misses, branch mis-predictions, etc. These can be very important in tuning programs to find inefficiencies in application codes. However, these counters are not directly accessible from the user space. For such information to be accessible, various patches to the kernel and supporting user-space utilities are available. One of these, the `perfctr`[26] patch is discussed later.

## 2.2 Message Passing Interface

As mentioned briefly in the previous chapter, Message Passing Interface (MPI) is a standard designed for programming distributed memory computers. Thus MPI tasks run on separate computers and communicate via messages. MPI tasks can also be run on a multiple processors on a single node. In the latter case, the tasks still communicate using messages, maintaining the programming model in either case.

There are two parts of the MPI standard, MPI-1 and MPI-2. MPI-1 covers point-to-point communication (sending messages from one task to another) and most collective operations (one task sending messages to a group of other tasks). Different vendor implementations of MPI exist, some of which are: MPICH from Argonne National Lab, MVAPICH from Ohio State University and OpenMPI developed and maintained by multiple contributors. Although all of these conform to the standard, their implementations vary and hence they have different effects on performance.

MPI programs have a typical structure as follows:

```
int main (int argc, char* argv []){
        int id, numTasks;
```

```
        MPI_Init (&argc, &argv);

        MPI_Comm_rank (MPI_COMM_WORLD, &id);
        MPI_Comm_size (MPI_COMM_WORLD, &numTasks);

        // Process data

        // Optionally exchange messages using MPI_Send(), MPI_Recv() or other
           functions

        MPI_Finalize ();
}
```

The `MPI_Init()` function initializes the MPI environment. Similarly, `MPI_Finalize()` shuts down the MPI environment. `MPI_Comm_rank()` and `MPI_Comm_size ()` retrieve the rank (ID) of the MPI process and the total number of tasks respectively. As the names suggest, `MPI_Send()` and `MPI_Recv()` routines establish point-to-point communication.

Obviously, hybrid MPI/OpenMP programming involves using threads. These threads can be used not only for processing data but also for making calls to the MPI runtime. This implies that the MPI runtime should be thread-safe. It is possible to get around this requirement by ensuring that only one thread invokes the MPI routines (via the `#pragma omp master` directive). However, this requires placing an extra barrier synchronization directive which may increase the runtime overhead. The MPI-2 standard makes it mandatory for MPI implementations to support thread safety. The MPI runtime can be initialized in a thread-safe manner using `MPI_Init_thread()` function[16] (which is the counterpart of `MPI_Init()` for the non-threaded case). The desired level of thread support can be configured via parameters to `MPI_Init_thread`.

```
int MPI_Init_thread(int *argc, char *((*argv)[]), int required, int
                                    *provided)
```

The value of `required` for the above mentioned case can be:

- `MPI_THREAD_FUNNELED`: Only the master thread in a process will make calls to MPI routines
- `MPI_THREAD_SERIALIZED`: Multiple threads may make calls to MPI routines, but only one at a time
- `MPI_THREAD_MULTIPLE`: Multiple threads may call MPI routines with no restrictions

`provided` returns the highest level of thread support.

MPI programs are executed using a special wrapper script called `mpiexec`. For example,

```
mpiexec ./a.out
```

Optional arguments to mpiexec make it possible to speicify the total number of MPI tasks and also the number of MPI tasks per node as shown below:

```
# Run hello-world program using 16 MPI tasks
mpiexec -np 16 ./hello-world
# Run hello-world program using just 1 MPI task per node
mpiexec -npernode 1 ./hello-world
```

## 2.3 OpenMP

OpenMP is a standard for programming shared memory systems. OpenMP allows creation, synchronization and destruction of threads along with sharing work among them. OpenMP does this by means of compiler directives (`#pragma omp ...`). Like MPI, OpenMP is a standard for which different implementations exist. However, unlike MPI, OpenMP is not distributed mainly by means of libraries. It is instead included in the compiler and hence the implementation of OpenMP depends on the compiler that was used. All major compilers like GNU gcc, Intel icc, IBM xlc, Sun Studio compilers and Portland Group compilers include support for OpenMP. The most recent version of the OpenMP standard is 3.0. Full support for OpenMP 3.0 is available in the recent versions of all the above-mentioned compilers.

As mentioned earlier, OpenMP uses work-sharing compiler directives to parallelize application code among threads. OpenMP supports data parallelism using `#pragma omp parallel for` and task parallelism using `#pragma omp parallel sections`. Additional options to these directives allow customizing the way the work is divided among threads. OpenMP also provides APIs for tasks like querying/setting the number of threads, stack size, etc. Some of the runtime options can also be set using shell variables.

## 2.4 Hybrid MPI/OpenMP Programming Paradigm

MPI (the Message Passing Interface) and OpenMP are the de-facto standards when writing parallel programs. MPI provides an explicit messaging model with no assumption of shared memory, and as such is the standard for use on distributed memory systems. OpenMP

provides a threading model, with implicit communication and the assumption of shared memory, and therefore is often used on shared memory systems. Hybrid programs are those that use both MPI and OpenMP – MPI for communication between nodes and OpenMP for communication within a single node. For the last several years, most clusters have been composed of a collection of multi-core nodes, with shared memory at the node level, and distributed memory between nodes. This appears to fit well with the hybrid model. The other, more compelling reason for using both MPI and OpenMP is that communication by means of shared memory is known to support much greater bandwidth as opposed to communication using messages[6]. There have been efforts[19] to make MPI leverage the shared memory architecture on a node. However, from our experience hand-tuning using MPI and OpenMP gives greater benefits than relying on the techniques in the MPI distribution alone. Similarly, efforts like those extending OpenMP to clusters[15] failed to give performance equivalent to that of hand-tuned MPI+OpenMP code. Thus, to get optimum performance from the cluster of computers, one needs to take into accout it's organization.

## 2.5 Challenges In Hybrid MPI/OpenMP Programming

The hybrid programming model combines MPI and OpenMP to get the best performance from the underlying architecture. However, a straight-forward integration of OpenMP constructs into the MPI program often does not give good speedup results[27]. Using OpenMP to get good speedup values can be difficult due to the inadequate transparency provided by the language constructs. With the programmer specifying just a single statement like "`#pragma omp parallel for`", the compiler and the runtime system together have to find the optimum way of parallelizing the given loop. While this simplicity in specifying the OpenMP construct is good for the programmer, it is difficult to optimize. This simple syntax is complemented by additional library routines to pass hints to the compiler and the runtime system. These include means to set the thread stack size, allowing or disallowing nested parallelism, setting the manner in which workload is distributed among threads, etc.[25]. As an aside, one may

note that the growing number of such supporting library routines (10 in OpenMP version 1.0, 22 in OpenMP version 2.0 and 31 in OpenMP version 3.0) is an indication of the growing complexity and detail that needs to be added to the OpenMP specification.

Also, for hybrid programs the characteristics of the MPI distribution (MPICH, MVAPICH, OpenMPI, etc.) also have an effect on the performance of the program. Due to these difficulties, there is a high degree of uncertainty when it comes to programming large applications for multi-core processors.

Hybrid programming paradigm is still quite new and its benefits are still being discovered. There has been only limited work into specific techniques for optimizing applications for the hybrid programming paradigm. The work in this thesis is directed towards finding how hybrid parallel programs can be further improved using optimizations specific to that programming paradigm.

## 2.6 Motivation For Using Hybrid Paradigm

Clearly, using the hybrid MPI/OpenMP paradigm requires more effort in programming and debugging application codes. However we noticed that adopting the hybrid programming paradigm provided some significant benefits as listed below.

- Hybrid programs seem to fit well with most current HPC cluster architectures
- Hybrid codes appear to scale well
- Hybrid applications should make better use of the high shared memory bandwidth as opposed to message driven applications
- They contribute lesser towards message transfer overhead when operating over a large number of nodes

Contrary to what one might believe, hybrid optimizations tended to have increasing impact as the number of nodes used in a given program increased. That is, although optimizations are applied within a node, improvement in running time is clearly visible as the number

of nodes used is increased. However, the performance of hybrid codes depends on the arrangement of cores and sockets and also on the ability of the algorithm to make use of shared memory. Hence writing and tuning hybrid applications can be difficult.

The results demonstrating the scalability of hybrid codes and their memory utilization are shown later.

## 2.7 Running Hybrid Programs

Since hybrid MPI/OpenMP applications have a two-level hierarchy (processes and threads), invocation of hybrid programs follows a slightly different approach as compared to invoking pure-MPI programs. To understand this, consider the architecture of today's clusters. These clusters are built using multiple nodes (or computers) connected to each other using a high-speed network such a Gigabit Ethernet, Myrinet or Infiniband. Inside each node is a hierarchy of sockets and cores. A 16-core node might be typically organized as 4 sockets, each of which holds 4 cores, as shown in figure 1.2. Such a hierarchy exists to balance cost and performance (with the two extremes being a single socket holding 16 cores and 16 sockets having a single core each). Furthermore, each core is not connected to memory directly. Instead, all cores within a socket are connected to each other and only a subset of these cores are connected to memory banks directly. Needless to say, the memory latency for each core varies.

Considering the typical design of clusters as mentioned above, the best way to run hybrid MPI/OpenMP programs would be to have 1 MPI task per socket and an equal number of OpenMP threads per task as the number of cores per socket. So for a cluster built using 4-socket, 4-core nodes, one might say:

```
OMP_NUM_THREADS=4 mpiexec -npernode 4 ./hybrid-a.out
```

This starts four MPI tasks for every node. Each MPI task will contain four OpenMP threads.

## 2.8 Optimization Tools

As mentioned earlier, the features provided by the threading mechanisms/libraries like OpenMP are often insufficient for tuning and optimizing application codes. In such cases, third-party tools help to complement the limited support provided by OpenMP. The tools that were used in this work are explained below.

*PAPI*

The Performance API (`PAPI`)[24] is an API framework for measuring hardware related events such as cache hits / misses, cycles spent in memory access, floating point operations per second, branch predictions / mis-predictions, etc. `PAPI` essentially is a layer of abstraction over the `perfctr`[26] performance counter software. The `perfctr` interface provides a means to access hardware performance computers. However `PAPI` ensures that the way this performance counter information is obtained from the hardware is consistent across different hardware platforms. The disadvantage is that the hardware may support only a finite (and often severely limited) amount of hardware counters. However this can be overcome in certain cases by making `PAPI` multiplex counting of two or more events.

The use of `PAPI` becomes especially important when using threading. This is because communication between threads is implicitly handled by the processing cores, caches, memory subsystem and the bus. Due to the complex architecture of modern day CPUs, one cannot easily predict the chain of actions that will be triggered in each of the above-mentioned subsystems at run time. These under-the-hood activities are not easily identifiable but measuring hardware events like cache misses, TLB misses, branch mis-predictions using `PAPI` can give a good insight into their cause.

*TAU*

`TAU` stands for Tuning and Analysis Utilities[28]. It is a profiler that is well suited for measuring performance of parallel applications that use MPI. `TAU` can also profile OpenMP

based applications by making use of the Opari[23] toolkit (which instruments the source code of the OpenMP program). TAU displays results using pprof (console-based utility) or using paraprof (graphical interface program). In the most generic form, TAU displays the amount of time spent by each process and thread in every function. It also displays the mean and standard deviation values, that can help to find if there are load imbalance issues. TAU allows simple arithmetic operations on two metrics to form a single derived metric. It also allows measuring system events reported by PAPI, thus making it possible to measure events related to cache, TLB, memory and branch prediction.


Figure 2.1. TAU Profiler

There exist other thread profilers too like OProfile, Intel VTune analyzer, gprof, etc. However, many of them suffer from drawbacks when it comes to tuning hybrid MPI/OpenMP applications. OProfile and Intel VTune analyzer required super-user access to the compute nodes. gprof is not inherently suited towards profiling MPI or OpenMP applications. TAU is a thread profiler that is specifically suited for tuning MPI, OpenMP and hybrid applications.

*libnuma*

`libnuma`[20] is a collection of APIs and a command line program that allows setting the Non-Uniform Memory Access (NUMA) policy. Through this interface it is possible to set the CPU affinity and memory placement for any program. The `numactl` interface is a user-space utility and hence requires appropriate support from the kernel. The kernel can be configured to support NUMA policies by setting the `CONFIG_NUMA` flag prior to compilation of the kernel. This flag is currently available only for 64-bit processors.

The command line program is called `numactl` and it supports various policies for setting the CPU affinity and memory placement such as Round Robin, Interleaved, Local-only or custom allocation. `numactl` also allows setting the NUMA policy for System V shared memory segments or files. However, these do not apply to programs using MPI or OpenMP. The command line utility sets the NUMA policy for the entire program and hence the exact policy cannot be modified during program execution. As opposed to this, the `libnuma` API allows greater flexibility.

Chapter 3: Hybrid Paradigm Observations

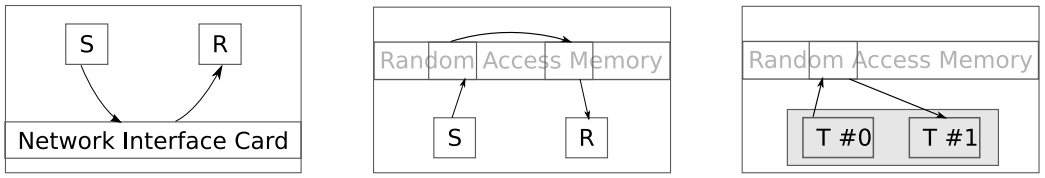
This chapter talks about the factors that we investigated about the hybrid programming paradigm. It starts with explaining the differences between hybrid MPI/OpenMP applications and pure-MPI applications. It not only considers the differences in the programming paradigm and the architecture but also the differences that were noted considering runtime factors. Based on these observations, techniques used for optimizing hybrid applications are explained next.

## 3.1 Hybrid MPI/OpenMP Versus Pure-MPI Applications

The biggest difference between pure-MPI programs and hybrid programs is that hybrid programs use threads in conjunction with processes. Apart from this, there are few other differences between these two programming paradigms that have an effect on the consumed CPU time, memory utilization and time to transfer messages.

*Shared Memory*

Since there are threads at the very base of any hybrid program, we are dealing with shared memory whenever communicating between threads. As opposed to this inherent design of shared memory, pure-MPI programs have traditionally implemented sharing of data (even on the same node) via messages. On a single node, this is made possible by using the local loopback interface, as shown in Figure 3.1(a).




(a) Using the loopback interface

(b) Using memory instead of I/O

(c) Using true shared memory among threads

Figure 3.1. Communication Between Tasks And Threads

This scenario has changed with recent MPI distributions starting to implement the data sharing via shared memory as shown in Figure 3.1(b). However there are subtle differences in the way data are shared using shared memory in the pure-MPI case and in the hybrid case. The difference arises from the fact that MPI implementations have a buffer-copy step to share

the data from the data structure in one MPI program to that in the other MPI program[3]. On the other hand, in the hybrid case explained in Figure 3.1(c), the programmer is aware of the shared memory and hence allocates only a single large data structure and reads from different portions of it (portions that each thread operates on). Thus the buffer copy step is avoided in the hybrid case.

*Larger Data Sizes*

Typically there are less MPI tasks being run in a hybrid program as compared to a pure-MPI program utilizing the same hardware and operating on the same size of problem. However, due to lesser MPI tasks, the per-task data size is greater for the hybrid program than for the pure-MPI progam. This usually has negative effects for hybrid code because, unless proper care is taken, the entire tasks data may reside in memory close to a single socket instead of being spread out appropriately among all sockets. This problem may happen due to the first-touch policy used by operating systems. This implies that a memory segment is allocated on the node/socket that first accesses it.

As an example, consider a hybrid code in which the parallel region is entered after the data is received from MPI task #0. In this case, all the data is likely to be allocated near the socket that ran the MPI task #0. Thus there would be an access penalty for every thread that accesses the memory from the other socket. To get around this problem, memory placement techniques have to be employed using the `libnuma` interface described earlier.

*Startup And Shutdown Overhead*

Starting up an MPI program means that the scheduler (such as PBS or LSF among others) has to invoke a task on the compute node (a remote node). Previously, this was done with the scheduler logging into the remote nodes via `rsh` or `ssh` and starting a task on the remote node[18]. This is not just slow but it also does not allow the scheduler to control the (remote) MPI process or manage its accounting information. More recent schedulers like PBS export interfaces that allow asynchronous spawning of MPI tasks causing the MPI startup delay to

be reduced drastically[13],[30]. Accordingly recent MPI distributions take advantage of these features to load MPI programs faster[18],[4].

However, the size of current clusters growing to thousands of cores has implications on the startup and shutdown delays for MPI programs. Using pure-MPI programs is hence a concern for high core counts. On the other hand, with hybrid MPI/OpenMP applications, due to lesser MPI tasks running on the same number of cores, the startup and shutdown overhead is lesser. The graph in Figure 3.2 shows the measured values for pure-MPI and hybrid MPI/OpenMP programs. These values were measured by finding the difference between the time reported by the UNIX `time` command and the time taken inside the `main()` function.

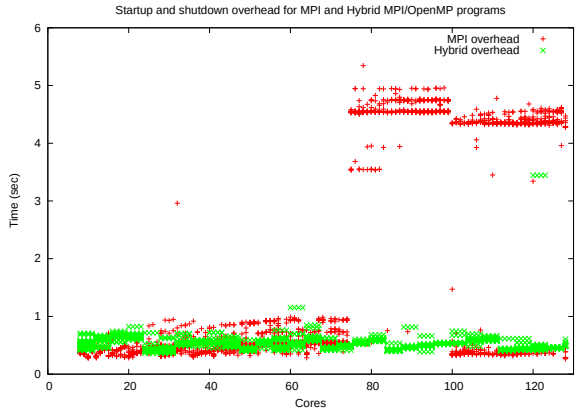



Figure 3.2. Overhead Of Starting And Shutting Down MPI Tasks

*Hierarchical Organization (Processes And Threads)*

As opposed to the flat model of pure-MPI, hybrid code has a two-level hierarchy of processes and threads. Also, as described earlier, hybrid code has lesser number of MPI tasks than in pure-MPI code. This helps mitigate some of the negative points about MPI. For example, there are concerns whether MPI would scale well at a very high number of cores (1M+) as it does for relatively smaller core counts[2]. Another problem with MPI that hybrid programming mitigates is the time required for performing collective operations.

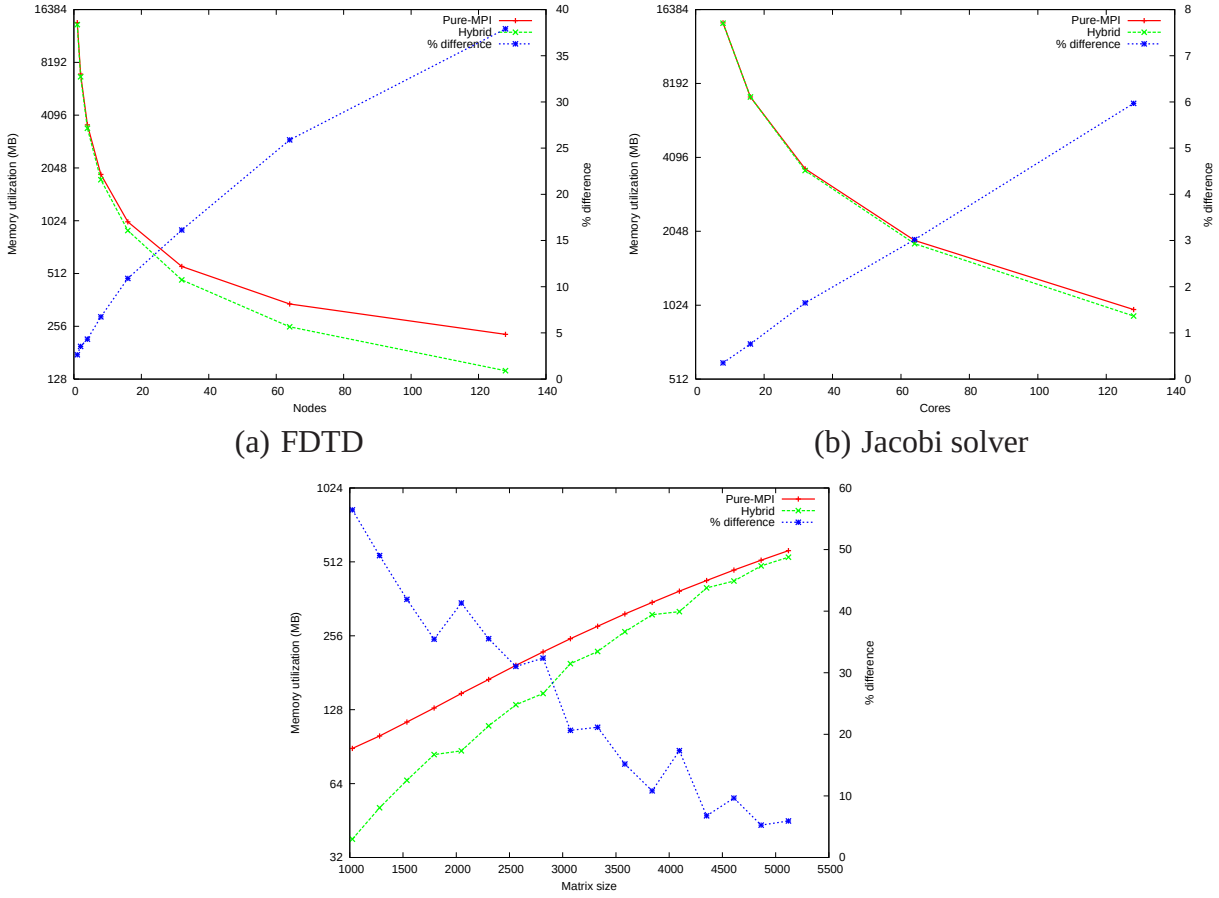
Also, due to the two-level hierarchy, the steps that need to be taken to optimize a pure-MPI program are not entirely the same as the steps to be taken for optimizing hybrid programs. Due to the presence of threads, finding ways of optimizing hybrid code becomes slightly more difficult. This is especially true because certain things such as location of memory segments, cache behavior, communication between cores (both intra-socket and inter-socket), etc. is very hard to verify. These issues either do not exist or are less complex for MPI-only programs because the MPI task (process) is the only executing entity.

*Memory Usage*

Another difference that is seen between pure-MPI and hybrid applications is in the memory that is consumed. The memory consumed by programs includes both code as well as data memory. Pseudo-files like `/proc/meminfo`, `/proc/pid/status` and tools like `ps` report the amount of Virtual Memory (VM) occupied by a process. In the most simplistic case, the VM size of a process would equal its physical memory consumption. However, two or more programs may be referring to a shared segment of code or data and hence looking at the VM size would be an incorrect way to infer the physical memory allocated to the process. The `pmap` utility (which formats the data available from `/proc/pid/maps`) shows the address map of all pages allocated to a process along with their type (private or shared). The `/proc/pid/smaps` file presents greater detail by listing the memory consumption of each entry in the address map. Both `maps` and `smaps` files provide similar information about threads in a process in files under `/proc/pid/task/task-id/`. Since it is the `smaps` file that provides the most accurate information, it was used to measure the memory utilization.

The observation made was that the hybrid program consistently used lesser memory than the pure-MPI program. Furthermore we noticed that the memory utilization goes on decreasing drastically as we increase the node count. As demonstrated by the plots in figures 3.3(a) and 3.3(b) and in figure 3.3(c), for a program that runs on a fixed number of cores, it can be seen that using hybrid codes dramatically reduces the memory footprint of the program. This also

implies that the hybrid paradigm makes it possible to solve problems of a large size that might not have been possible when using the pure-MPI paradigm.




(a) FDTD (b) Jacobi solver

(c) Matrix multiplication

Figure 3.3. Per-Node Memory Usage Comparison For Pure-MPI And Hybrid Applications

*Effect Of Message Aggregation*

The overhead of transferring messages is lesser when the messages are aggregated as opposed to when they are sent separately. This comes into play when strong scaling. In strong scaling, the dataset size is kept constant and the number of cores in gradually increased. Hence the per-task data sizes goes on decreasing. However the time to transfer messages does not decrease till zero. Instead it reaches from minimum and then either stays constant or starts

increasing. However since aggregated messages are always larger than individual messages, they contribute lesser to message overhead than individual messages. The plot showing the communication time for one of the codes is shown in figure 3.4.

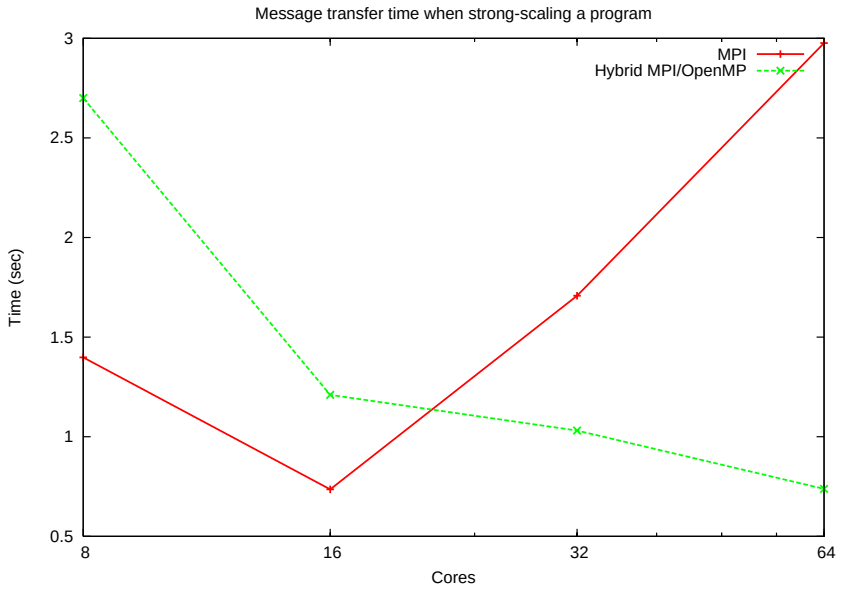



Figure 3.4. Communication Time For Constant Problem Size And Increasing Cores

*Effect Of Message Transfer On Hybrid Programs*

Measuring values from performance counters for MPI and hybrid codes (included in Appendix B) showed greater insight into the execution of the programs. We measured the number of instructions executed, cycles spent and the number resource stalls arising in both the computation and communication portions of the code. However, most of the codes use non-blocking communication and hence, strictly speaking, the communication portion in which the performance counters were measured is just the `MPI_Waitall()` function. It was observed that, in general, the number of cycles spent in communication is just a fraction of those spent in computation, i.e. message transfer was being almost overlapped with the computation. The only program that was an exception was the spectral code (FFT). For the FFT program, it was observed that as the number of cores were increased, there was more time being spent in the

communication portion of the code than in the computation portion, i.e. the ratio of cycles spent in communication to those in computation was greater than 1.

To understand the effect of non-overlapped communication, we studied the comparison of time spent in different MPI communication operations (basic send/receive, broadcast, reduce, gather, scatter, allreduce, allgather, alltoall). For each operation, the MPI code performed 1000 iterations with a message of size 128 KB while the hybrid code performed the same number of iterations with a message of size 512 KB (since the data would aggregated among 4 threads). In one case the "problem size" showed strong-scaling and in the other case it exhibited weak-scaling. The results are shown in Figures 3.5(a) to 3.5(b).

From the results, one can infer that non-overlapped communcations in hybrid code (when using `MPI_THREAD_FUNNELED`) can be detrimental to performance, unless it is an `MPI_Alltoall()` communcation. However, since non-blocking collective operations are not yet available[14], these cannot be hidden under computation sections.

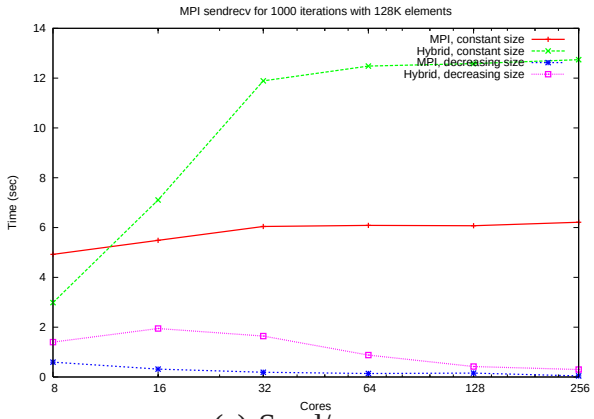
## 3.2 Tuning

In this section, we consider different techniques for tuning hybrid programs based on the differences and observations noted in the previous section. Specifically, there are three main optimization techniques that we talk about:

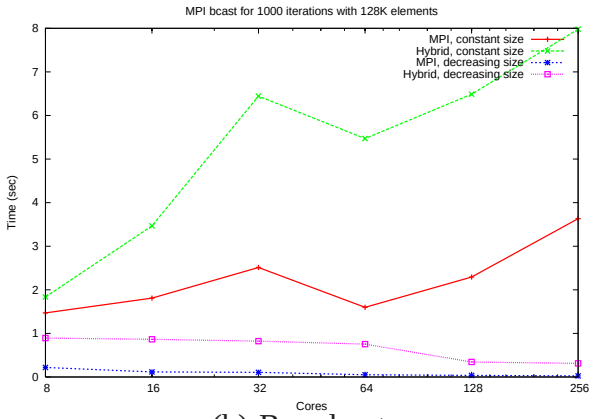
- Minimizing OpenMP `parallel` overhead
- Using multidimensional decomposition
- Specifying thread and memory placement

*Minimizing OpenMP Parallel Overhead*

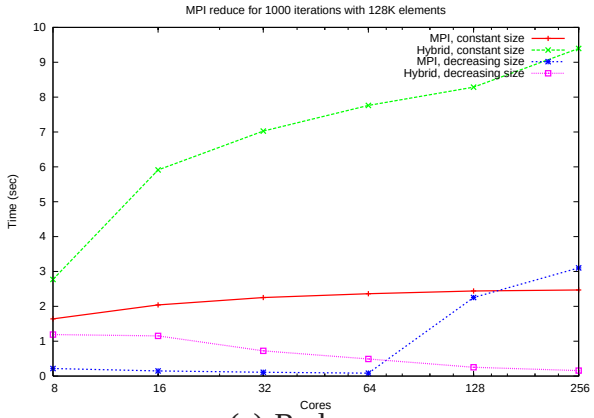
Scientific codes usually exhibit a "big" outer loop that repeats individual steps of the computation. This loop may represent the time steps of a simulation or continued execution of the body of the loop till the results of the computation converge, etc. In such cases, inserting



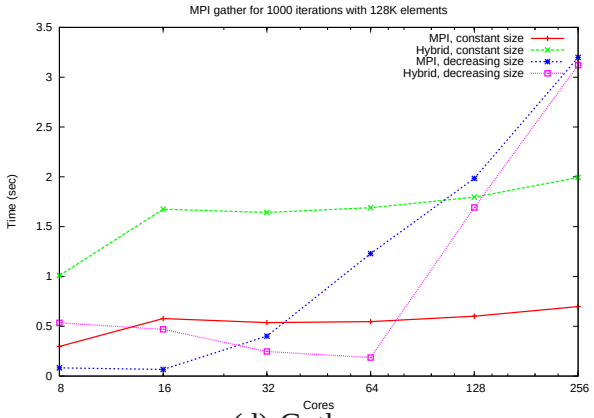

(a) Send/recv



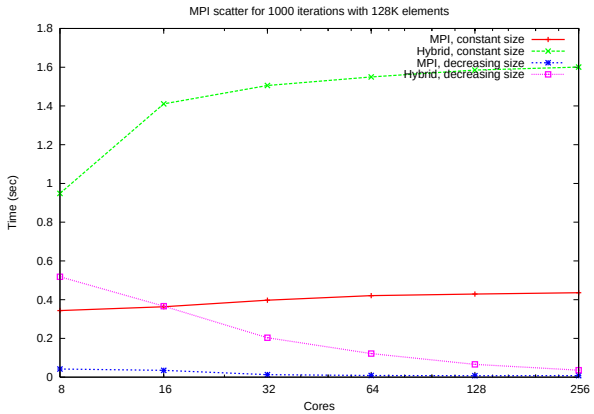

(b) Broadcast



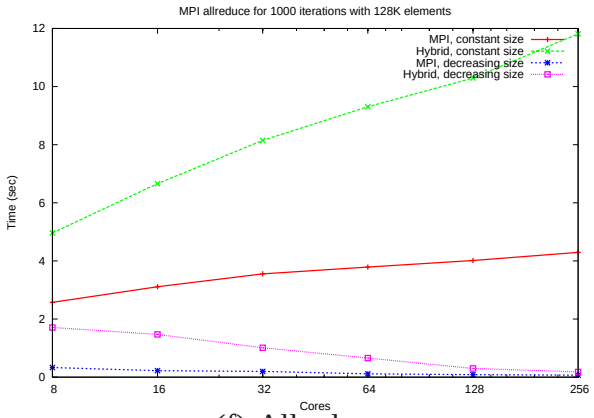

(c) Reduce




(d) Gather

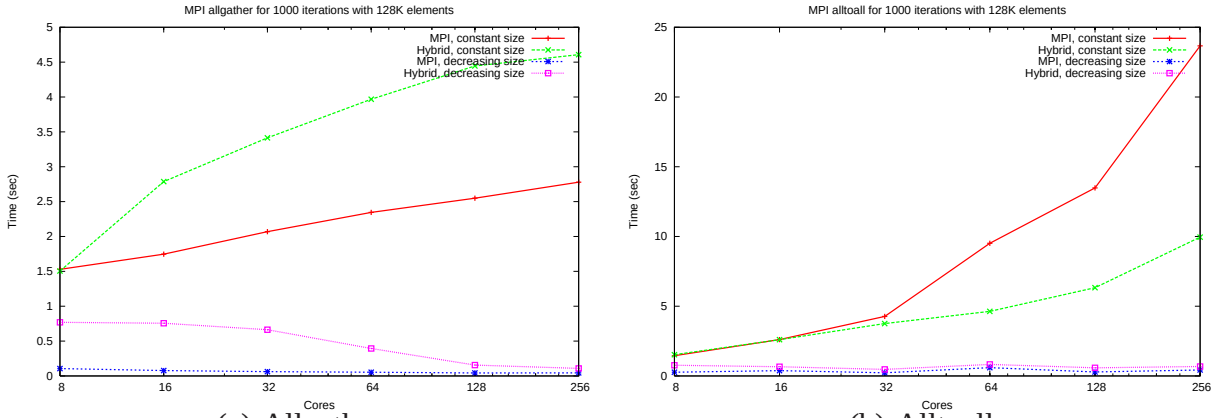



(e) Scatter




(f) Allreduce

Figure 3.5. Comparing MPI and Hybrid Communication Operations




(a) Allgather (b) Alltoall

Figure 3.5. Comparing MPI And Hybrid Communication Operations

"`#pragma omp parallel for`" directives for the inner loops doing the computation of a single step is probably the most natural way of parallelizing the code. However, although creation of the threads is performed only once in the program's lifetime, each `parallel for` directive introduces the overhead of waking the threads at the beginning of the `parallel` block. When this is repeated over multiple iterations of the outer loop, this overhead increases too. The Intel OpenMP runtime[17] uses thread pooling to eliminate repeated creation and destruction of threads and thus tries to reduce the overhead. However, even with this provision, we observed that we could reduce the running time of the program by hoisting the `parallel` pragma outside of the outer loop.

To eliminate the repeated overhead mentioned above, we hoisted the `parallel` directive outside the loop. Thus the outer loop is executed by all threads independently. Synchronization between the threads can be done using the OpenMP `single` and `master` directives as illustrated in Figure 3.6. Also, care needs to be taken to explicitly mark variables as either private to each thread or shared among the threads and what should be their initial value, if any.

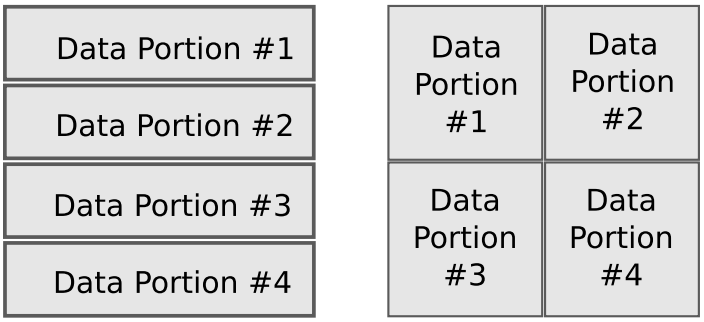
```
for (...)
{
        // Initialization
        {
                ...
        }

        // Computation
        #pragma omp parallel for
        for (...)
        {
                ...
        }

        // Communication
        {
                MPI_Send (...);
                MPI_Recv (...);
        }
}
```

```
#pragma omp parallel private
   (...)
{
    for (...)
    {
        #pragma omp single
        {
            // Initializations
            ...
        }

        // Computation
        #pragma omp for
        for (...)
        {
            ...
        }

        #pragma omp master
        {
            // Communication
            MPI_Send (...);
            MPI_Recv (...);
        }

        #pragma omp barrier
    }
}
```

Figure 3.6. Minimizing Parallel Overhead

However, in such cases one needs to ensure that the MPI runtime is initialized in a thread-safe manner.

The code shown in Figure 3.6 is a representative of the equivalent hybrid code when `MPI_THREAD_FUNNELED` is the maximum thread support available. Since it is only the master thread that can make calls to MPI routines for the `MPI_THREAD_FUNNELED` case, all MPI sends and receives have to be wrapped into the `omp master` directive. However the initialization code could be executed by any thread and hence is written as part of the `omp single` directive.




(a) One dimensional data decomposition

(b) Two dimensional data decomposition

Figure 3.7. Tiling Or Data Decomposition

If the data, and the initialization operations on the data, can be split equally among all the cores, each core could initialize its own data instead.

*Using Multi-Dimensional Decomposition*

Many programs in the area of high performance computing process data arranged in 2-dimensional or 3-dimensional data structures. A few examples of these are Jacobi solvers, finite difference methods for electro-magnetic simulations and matrix multiplication. For such codes, parallelization is usually performed by splitting the data among processors and each processor processing its local data. There are various ways in which this splitting can be performed depending on the number of dimensions chosen and the exact dimensions. As an example, consider a 2D grid of size L x B elements that needs to be split among a given number of processors. The grid can be split either along the L dimension or the B dimension (as shown in figure 3.7(a)) or along both L and B dimensions (shown in figure 3.7(b)).

The data access pattern of the program along with the selected number of dimensions for decomposition and the size of the local data that each processor operates on, together influence the running time of the program. As an example consider the Jacobi solver program, which, for each write, reads the four neighboring elements. When using a row-wise single-dimensional decomposition as shown in figure 3.7(a), each processor would process a complete row of data before moving to the next. Thus the number of memory references between accesses to the

same data element (once as a read and the next time as a write) depend on the size of data grid in each dimension (either L or B). When L or B is very high, the number of intermediate references made is also high. Hence the likelihood of the read/write element being retained in the cache is very less. As opposed to this in a 2D decomposition, the number of memory references between access to the same element will be less because of the smaller per-processor data size in each dimension. As a general rule, the higher the number of dimensions in the decomposition, the better the performance. To get the best performance, it is also important to ensure that the per-processor data size is sufficiently large and that it fits well into cache lines to ensure that no false-sharing of data occurs among processors.

MPI supports multi-dimensional decomposition by providing routines to arrange processors into a grid and to distribute data among them. However, OpenMP does not have routines to implement multi-dimensional decomposition (also called tiling in the context of threads) in user programs. OpenMP provides only one way to implement data distribution (by means of `#pragma omp for`). This causes only the outer one of the nested loops to be parallelized and thus is equivalent to single-dimensional decomposition. A few macros were implemented to allow for such multi-dimensional decomposition in OpenMP or hybrid programs. The implementation is discussed in detail in Appendix A. The comparison between multi-dimensional enabled hybrid and pure-MPI programs showed that the hybrid code benefited greater from multidimensional decomposition.

*Thread And Memory Placement*

Thread affinity allows us to specify which CPU each thread should run on. Setting the affinity mask is applicable to both pure-MPI as well as in hybrid MPI/OpenMP situations but it gains special importance in hybrid programs due to conflicting settings between the MPI distribution and the OpenMP runtime. MPI distributions often set the CPU affinity mask for the MPI processes to ensure that the processes are confined to individual cores. When using hybrid MPI/OpenMP strategy, the OpenMP threads are created as part of the MPI process. If

the affinity for the threads is not set explicitly, they all inherit the affinity mask of the process. This implies that by default, all OpenMP threads are made to run on the same socket as the MPI process. This would clearly be detrimental to performance when you would want the OpenMP threads to occupy the available cores. Hence it becomes important to set the affinity mask explicitly when using hybrid strategy.

On similar lines, if the given program uses memory in an intensive manner, specifying the placement of memory can help improve performance. If the memory placement is not specified, the default placement policy used is the "First touch policy". This means that the memory segment is allocated on the node that first access it. This may not be desirable in situations when the master thread initializes the data and the remaining threads then operate on specific portions of the data.

As of today, there are no direct means in the OpenMP standard to specify thread affinity or memory placement. Two methods are known, by which, the desired thread affinity can be specified viz., the `KMP_AFFINITY`[17] environment variable for the Intel C/C++ compiler and the `numa_run_on_node()` function in `libnuma`[20] (or the equivalent `numactl` command line utility). It should be noted that the `KMP_AFFINITY` variable is specific to the Intel compiler, however other similarly named variables exist for other compilers, e.g. `GOMP_AFFINITY` for GNU GCC compiler. This variable essentially permits specifying three things:

- whether threads should be tied down to individual cores or should they float among the available cores
- whether the default affinity mask (of the process) should be respected and
- should consecutive threads be scheduled on neighboring (or close-by) cores (and thus control cache effects)

Another means of setting the thread affinity is by use of the `numa_run_on_node()` func-

tion. As seen from the source code[20] of the containing library `libnuma`, this function uses the `sched_set_affinity()` system call to set the affinity mask. The function takes the number of the core that the thread is to be associated with.

`numactl` also allows setting the memory affinity by specifying the node from which the memory is to be allocated. This can also be accomplished using the `numa_set_membind()` function.

These optimizations were used while evaluating the hybrid paradigm against the pure-MPI paradigm. The results of the comparison are presented in the next chapter.

# Chapter 4: Results

This chapter combines the knowledge of using the performance analysis tools described in chapter 2 along with the differences noted and the tuning techniques devised in chapter 3. The tuned hybrid programs are compared to the pure-MPI programs. These results are then analysed to find key characteristics of programs that benefit from the hybrid paradigm. This is done with the objective that it would then be possible to infer whether any given algorithm or program can benefit from hybrid programming.

## 4.1 Comparing Pure-MPI And Hybrid MPI/OpenMP Programs

This section presents the results comparing pure-MPI and hybrid MPI/OpenMP programs with respect to running time and speedup.

*Dense Linear Algebra - Dense Matrix/Matrix Multiplication*

For the dense linear algebra program, we optimized the hybrid MPI/OpenMP version of Cannon's algorithm[5] for Matrix multiplication. The algorithm operates by dividing the problem space into a square number of grids, with each grid being processed by one core. During processing, each core operates on its local grid and then does pair-wise exchanges of the grid with select neighbors. Thus, it involves communication along both axes. For the hybrid version of this dwarf, we implemented communication along one axis (or dimension) entirely using shared memory. Thus, hybrid program has to exchange messages only along one dimension instead of two. This makes it particularly beneficial in terms of running time for large problem sizes because a significant amount of network communication can be avoided by leveraging shared memory.

One can observe that the runtime for the MPI code does not increase in a continuous manner. We believe this is due to the heavy reliance of the MPI code on sends and receives. Since the hybrid code involves less communication in terms of messages, it shows a more consistent improvement in the running time across cores.

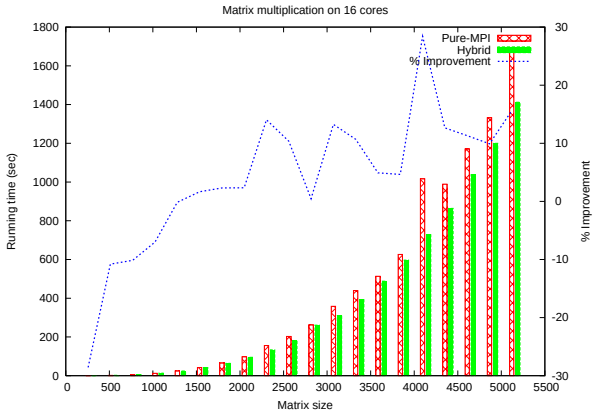



Figure 4.1. Matrix Multiplication Using Cannon's Algorithm

*Structured Grids - Jacobi Solver*

Next, we converted the Jacobi solver code from pure-MPI to hybrid. The hybrid code runs in almost the same amount of time that the pure-MPI code takes. To understand the reason, we compared the number of cycles spent in executing computation code to those spent in communication code, as shown in Appendix B. It was noticed that the communication cycles were only a fraction of the computation cycles, i.e. the problem is compute bound and trivially parallel. Since hybrid MPI/OpenMP results in savings in communication, there was not much scope for improvement in the total running time of the program. The plot comparing pure-MPI and hybrid version of the Jacobi solver code is shown in figure 4.2.

*Spectral Methods - Fast Fourier Transform*

The Fast Fourier Transform application is a communication intensive application using spectral methods to transform data. The application under test calculates a 2D FFT for varying problem sizes. It thus calculates 1D FFTs in parallel, performs an All-to-all communication and does another set of 1D FFTs. To ensure that the program runs for a significant amount of time, the entire process is repeated 10 times. The design of the problem does not allow for much manual tweaking to be applied to the code to optimize it. However, from the plot in

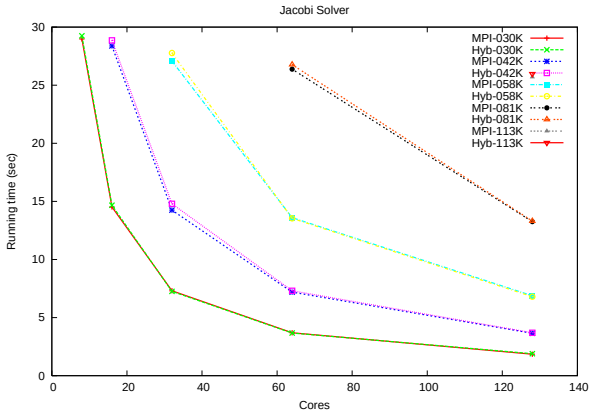



Figure 4.2. Jacobi Solver

figure 4.3 it can still be seen that the hybrid MPI/OpenMP program performs far better than the pure-MPI program. One interesting notion is the wide difference in the running time at 128 cores. To understand the cause, we studied the time spent by the code in the computation and communication phases. We noticed that the communication time increases sharply for the MPI code as the processor count is increased. The graph for communication times for the FFT code for both pure-MPI and hybrid programs is shown in figure 4.4.

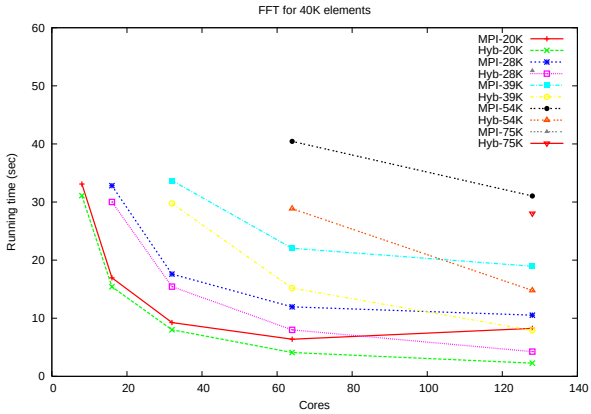



Figure 4.3. 2 Dimensional Fast Fourier Transform Calculation

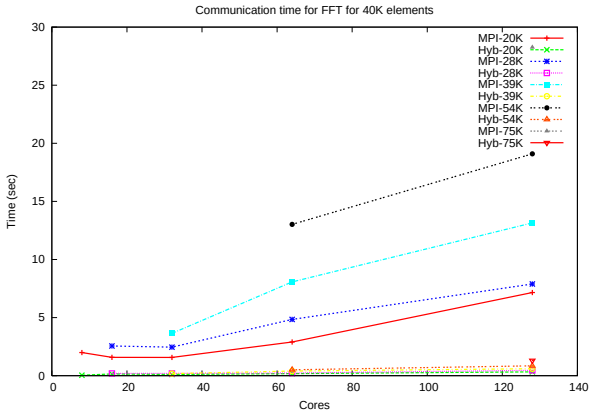



Figure 4.4. Time Spent In MPI_Alltoall() In Pure-MPI And Hybrid FFT Programs

*Monte-Carlo / MapReduce - Computation Of Pi*

The Monte-carlo Pi computation program is a purely computation intensive program. Each task/thread generates a specific number of random numbers between 0 and 1. The code iterates over these random numbers and performs simple calculations to calculate an estimate of the value of Pi. The only communication involved is when all tasks inform the master of their locally computed value of Pi. Like the FFT application, this code does not provide many opportunities for manually tweaking the code. Comparison results for running times for pure-MPI and hybrid MPI/OpenMP programs are shown in Figure 4.5. Being heavily compute-intensive, the exact opposite of the communication intensive FFT application, the hybrid code does not show a very high improvement in running time over the pure-MPI code.

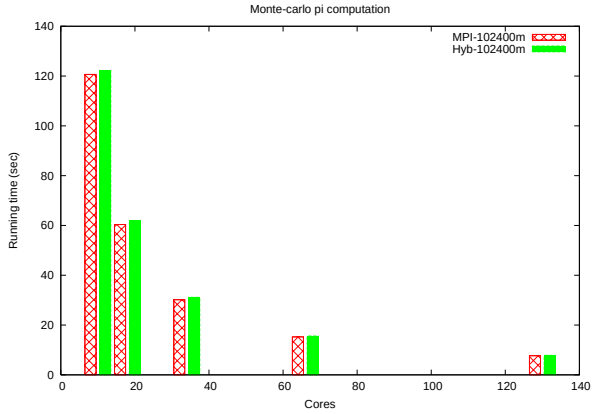



Figure 4.5. Computation Of Pi Using Monte-Carlo Method

*N-Body Methods - 3 Dimensional Gravitation Problem*

The results for the N-Body code are shown in figure 4.6. This code performs an $O(n^2)$ computation to determine the position and velocity of objects in a 3D space. During each timestep, the current position and velocities of the objects is passed "cyclically" among each task. One can see that after an approximately constant 16% improvement over the pure-MPI code up to 64 cores, the hybrid percentage improvement rises to about 30% at 128 cores. This follows the general pattern exhibited by communication intensive hybrid codes at a high core count.

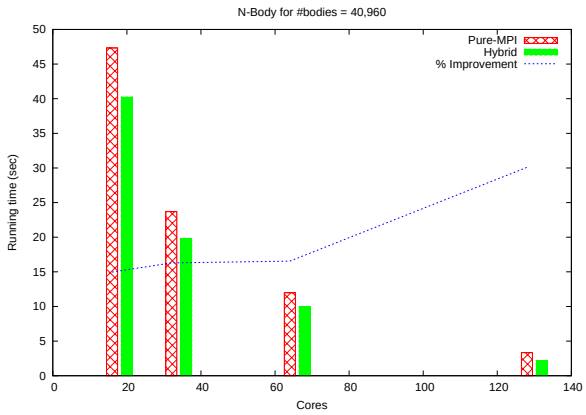



Figure 4.6. N-Body Simulation

*Sparse Linear Algebra - Sparse Matrix/Vector Multiplication*

The Sparse linear algebra code that we tested uses MPI collective operations (specifically, `MPI_Scatterv()`) to distribute the data among slave nodes. Since these collectives are non-blocking, there was little scope for hybrid optimization. The plot comparing pure-MPI and hybrid run times is shown in figure 4.7. Probing further into the characteristics of the plot revealed that the running time was heavily dominated by communication (the computation cycles were often under 10% of communication cycles). Furthermore, it was observed that the computation portion of the code was scaling well across cores. Thus to understand why

the communication portion was not scaling, we plotted the scaling characteristics of a code that uses only `MPI_Scatterv()`. This plot is shown in figure 4.8. Further analysis of this plot remains to be performed. However, one can note brief similarities in the plots in figures 4.7 and 4.8.

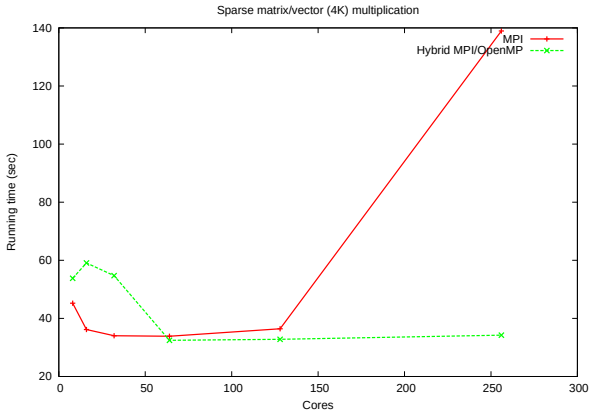



Figure 4.7. Sparse Linear Algebra

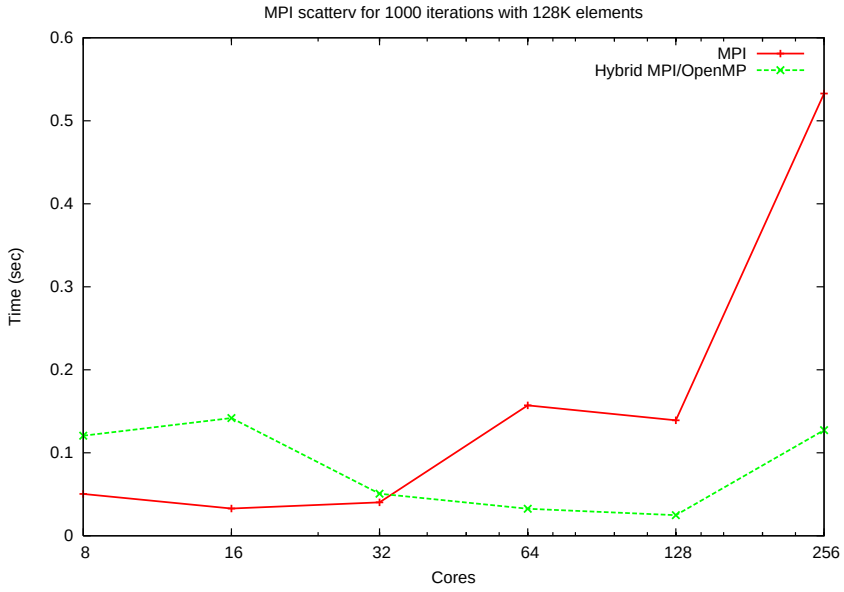



Figure 4.8. Scaling Characteristics Of MPI_Scatterv()

*Speedup Comparison*

The graph in figure 4.9 shows the comparison of speedup values using increasing number of cores for both pure-MPI and hybrid programs for a real-world application, 3-dimensional photonics simulation. Overall, the hybrid code displays better scalability than the pure-MPI code. Moreover, it can be seen that the difference in speedup values grows larger when using greater number of cores.

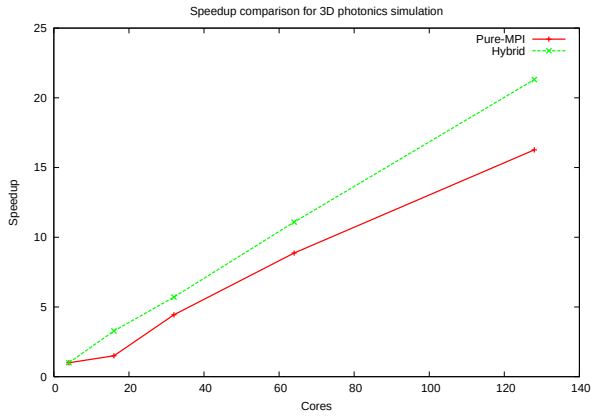



Figure 4.9. Comparing Speedup Values For Increasing Number Of Cores For Pure-MPI And Hybrid 3D Photonics Simulation Code

## 4.2 Tiling Results For Jacobi And Dense Matrix/Matrix Multiply Codes

Figures 4.10 and 4.11 shows the effect of tiling on two hybrid codes: Jacobi solver code and Matrix Multiplication code. The Matrix multiplication code was run for a fixed number of processors (16). One can see that the effect of tiling the code is more pronounced when operating over a large datasize and at high core counts.

## 4.3 Analysis Of Performance Counter Values

The performance counter values collected via the Performance API (PAPI) described in Section 2.8 for the dwarfs are shown in Appendix B. As expected, we observed that using hybrid MPI/OpenMP did not alter the time taken by the program in the computation sections of the code by a large extent. The only exception was in the case of the Fast Fourier Transform

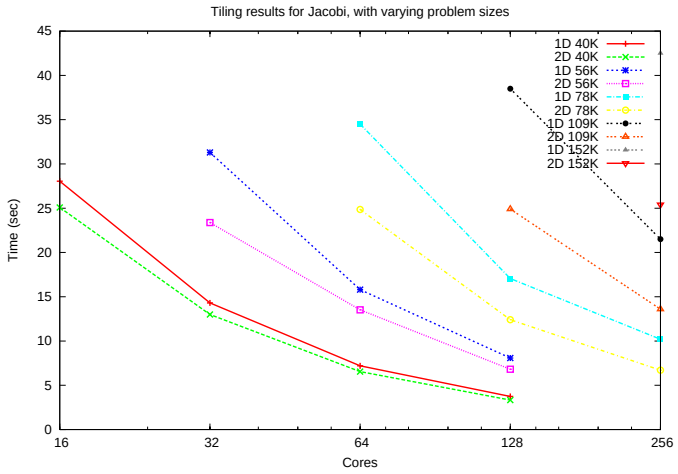



Figure 4.10. Comparing Tiled And Non-Tiled Hybrid Jacobi Codes

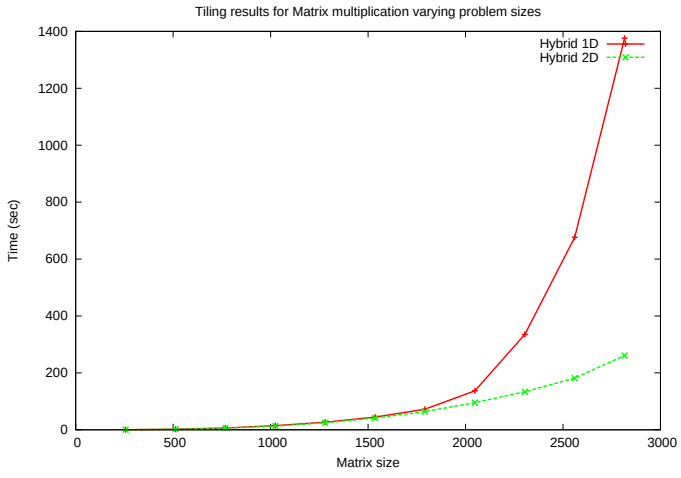



Figure 4.11. Comparing Tiled And Non-Tiled Hybrid Matrix Multiply Codes

(FFT) code. The FFT code uses the `fftw`[10] library for one-dimensional FFTs. The plot in figure 4.12 shows the ratio of cycles spent in computation portions of the hybrid code to those of the MPI code, which is almost always equal to 1.

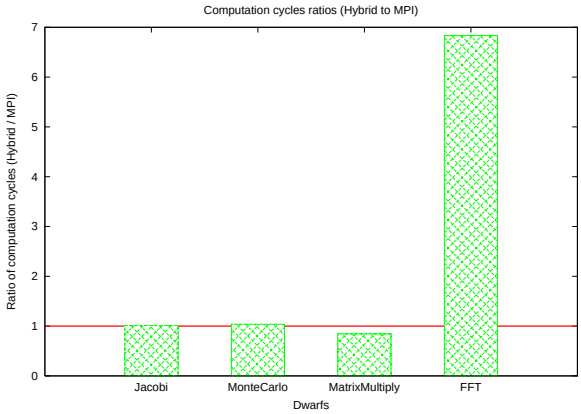



Figure 4.12. Comparing Ratios Of Computation Cycles Of Hybrid Codes To MPI Codes

Furthermore, we observed that the ratio of cycles spent in communication for hybrid and MPI codes varied significantly from dwarf to dwarf. The comparing plot is in figure 4.13. Since lesser number of cycles are spent in communication, the ratio of cycles spent in communication portions of hybrid code to that of MPI code is always less than 1. Moreover, the ratio varied depending on the "compute-intensiveness" of the code. The Monte Carlo code, which hardly involves any communication, has a ratio close to 1. This indicates that there was little benefit of hybridizing the code. On the other hand, the FFT code, which is communication-heavy, has a ratio much less, equal to 0.05.

Observe that the ratio in figure 4.13 for the Jacobi solver code is still relatively low. This may cause one to ask that if communication in Jacobi is optimized by approximately 40% then why does the total running time not improve by a comparable amount? To answer such questions, we compared the number of cycles spent in compute and communicate sections of both, MPI codes and hybrid codes. The results are in figure 4.14. Note that the vertical axis uses a logarithmic scale to accomodate the wide range of values. One can see that the ratio

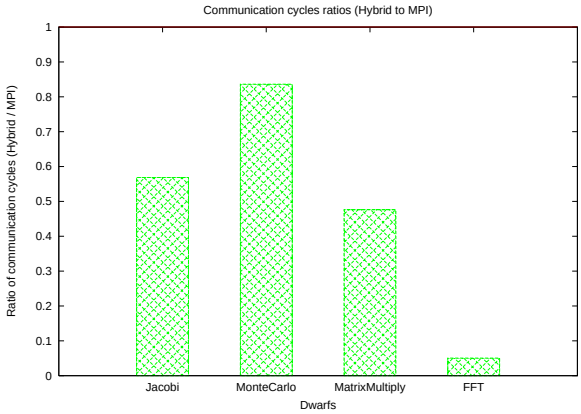



Figure 4.13. Comparing Ratios Of Communication Cycles Of Hybrid Codes To MPI Codes

for hybrid codes is always higher than for MPI codes. This is because of the small fraction of cycles spent in communication for the hybrid code.

The value of the ratio points towards the compute-intensiveness of the code. Considering the values for pure-MPI applications, for the Jacobi solver code this ratio is of the order of 100s, for Monte Carlo and Matrix Multiply codes in the order of 10s and for FFT, less than 1. Due to the relatively high number of cycles spent in computation to those in communication for the Jacobi solver code, hybrid optimization results in almost no change in the total running time of the program.

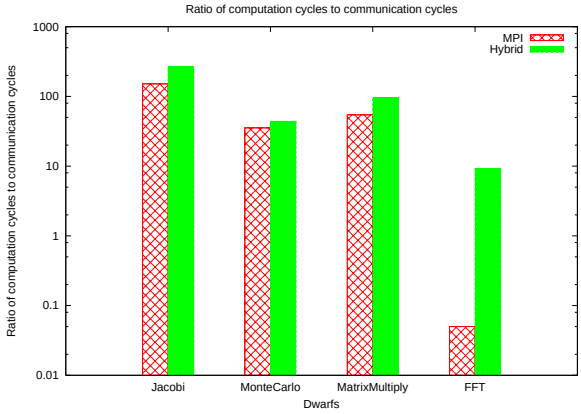



Figure 4.14. Comparing Ratios Of Computation To Communication Cycles

# Chapter 5: Conclusions And Recommendations

The multicore paradigm has forced us to rethink the way we program for solving large problems. However, it is not only important to program "in parallel" but also to consider the programming techniques used. As Chapter 1 shows, the design of current-day clusters is better suited to the hybrid MPI/OpenMP programming technique, with nodes connected using high-speed networks and each node containing multiple multi-core processors. Thus MPI can used at the node-level and OpenMP can be used at the processor-level. This makes it possible to leverage shared memory and message passing at the right levels.

This work described the hybrid MPI/OpenMP programming paradigm and compared it with the pure-MPI technique. The main goals of this work were to illustrate the differences between the two parallel programming techniques, mention the tools and their usage for programming and tuning hybrid programs and to find out what "type" of applications benefit from the hybrid paradigm over the pure-MPI paradigm.

However, built-in support in OpenMP for tuning and optimizing application programs is limited. This forces using external tools and utilities, some of which are described in section 2.2.8. These help to find bottlenecks and override default OS scheduling decisions that may be inefficient for certain programs.

For proper comparison between pure-MPI and hybrid programs, the dwarfs (described in Table 1.1) were chosen. These are a set of programs that are diverse in the computation and communication characteristics but they still mimic the characteristics of parallel programs used today.

Differences between pure-MPI and hybrid MPI/OpenMP programs were described in section 3.3.1. These differences not only spanned the theoretical aspects of combining processes and threads but also runtime characteristics such as memory consumption, effects on message transfer and startup and shutdown costs.

Tuning hybrid programs is important as it was observed that often non-optimized hybrid code ran in almost as much time as the pure-MPI code. However, tuning the hybrid code using simple techniques described in section 3.3.2 yielded substantially positive results.

The following were the major findings of this work:

**Tuning hybrid programs is important** We observed that often non-tuned hybrid codes took almost as much time to execute as the pure-MPI programs. However, the tuning techniques listed previously, although not complete, caused substantially better performance than the pure-MPI program in the average case. In the worst case, hybrid program performance was almost as good as that of the pure-MPI program.

**Hybrid paradigm has the potential to optimize communication:** There was little difference that we observed in the computation portions of the code when comparing the two programming paradigms. We observed that if the ratio of number of cycles spent by the MPI program in computation to those spent in communication is very high (meaning, the code is compute-intensive) then hybrid optimization did not help much. Hence when hybridizing programs, one should target those that involve a "significant" (although not necessarily high) amount of communication.

**Leverage shared memory:** Since hybrid MPI/OpenMP essentially optimizes the communication portion of the program, it is very important to make maximum use of the shared memory. This often involves restructuring the algorithm to a certain extent. We observed that inserting '`#pragma omp parallel for`' statements hardly ever improved performance with respect to the pure-MPI run time.

**Use non-blocking communications:** Assuming the use of `MPI_THREAD_FUNNELED`, messages from threads need to be aggregated causing the message size in hybrid programs to be larger. Depending on the exact form of communication (send/recv, broadcast, alltoall, etc.) message transfer may take a significantly higher amount of time in hybrid codes than in pure-MPI codes. Hence non-blocking communication (for basic sends and recvs) should be used whenever possible.

**Collective communications usually hurt hybrid performance:** Since collective communications are blocking, there is often little that one can do to have a reduced effect of these on the total run time. Other than alltoall and scatterv communication, if a program makes intensive use of collective communications then it is likely to be not a good candidate for hybrid programming.

**Impact of hybrid optimizations is visible at larger core counts:** For the codes for which we saw that using the hybrid technique improved performance, the difference in running time between programs of the two paradigms increased as we increased the number of cores/nodes *or* the problem size. Thus, although hybrid optimizations are majorly concerned with intra-node optimizations, their effect is prominently visible across nodes, as the number of processing nodes are increased.

Overall, the hybrid codes were found to exhibit better scalability, made better use of shared memory and depending on the communication characteristics contributed much less to message transfer overhead. Using hybrid MPI/OpenMP paradigm appeared to be more beneficial for communication-intensive programs due to aggregated messages and lesser number of participating senders and receivers.

# BIBLIOGRAPHY


[1] Asanovic, K., R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick (2006, Dec). The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

[2] Balaji, P., D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. L. Traeff (2009). Mpi on a million processors. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 20–30.

[3] Barney, B. (2003). Posix threads programming tutorial.

[4] Barrett, B., R. Castain, and G. Shipman (2007). Open mpi: A high-performance, fault-tolerant message-passing interface. Technical report.

[5] Cannon, L. E. (1969). *A cellular computer to implement the kalman filter algorithm*. Ph. D. thesis, Bozeman, MT, USA.

[6] Cappello, F. and D. Etiemble (2000). Mpi versus mpi+openmp on ibm sp for the nas benchmarks. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Washington, DC, USA, pp. 12. IEEE Computer Society.

[7] Colella, P. (2004). Defining software requirements for scientific computing.

[8] Dongarra, J. (1994, May). Mpi: A message-passing interface standard. Technical Report UT-CS-94-230, University of Tennessee, Knoxville, TN, USA.

[9] Feldman, M. (2007). Our manycore future.

[10] Frigo, M. and S. G. Johnson (2005). The design and implementation of FFTW3. *Proceedings of the IEEE 93*(2), 216–231. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[11] Fruehe, J. (2005). Multicore processor technology. *Dell Power Solutions*.

[12] Geer, D. (2005). Industry trends: Chip makers turn to multicore processors. *Computer 38*(5), 11–13.

[13] Group, P. W. (1996, November). PSCHED: An API for parallel job/resource management version 0.1. http://www.pbspro.com/docs/psched-api-report.ps.

[14] Hoefler, T. and A. Lumsdaine (2008). Non-blocking collective operations for mpi-3. Technical report.

[15] Hoeflinger, J. (2006). Extending openmp to clusters. Technical report, Intel Corporation.

[16] Huss-Lederman, S. (1997, July). MPI-2: Extensions to the message passing interface. Technical report, MPI Forum.

[17] Intel Corporation. *Intel C++ Compiler 11.0 Professional Edition User and Reference Guides*. Intel Corporation.

[18] Jeff, B. B., J. Squyres, and A. Lumsdaine (2003). Integration of the lam/mpi environment and the pbs scheduling system.

[19] Jin, H.-W. and D. K. Panda (2005). Limic: Support for high-performance mpi intra-node communication on linux cluster. In *ICPP '05: Proceedings of the 2005 International Conference on Parallel Processing*, Washington, DC, USA, pp. 184–191. IEEE Computer Society.

[20] Kleen, A. (2005, April). A numa api for linux. Technical report, SUSE Labs.

[21] Meuer, H. (2002). *TOP500 Supercomputer Sites; electronic version*.

[22] Microsoft (2007). The manycore shift. Technical report, Microsoft Corporation.

[23] Mohr, B., A. D. Malony, S. Shende, and F. Wolf (2002). Design and prototype of a performance tool interface for openmp. *The Journal of Supercomputing 23*, 105–128.

[24] Moore, S., P. Mucci, J. Dongarra, S. Shende, and A. Malony (Jaunary 2003). Performance instrumentation and measurement for terascale systems. *Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, Volume 2723, pp. 53-62*.

[25] OpenMP Architecture Review Board (2008). *OpenMP Application Program Interface*. OpenMP Architecture Review Board.

[26] Pettersson, M. The `perfctr` interface. http://user.it.uu.se/~mikpe/linux/perfctr.

[27] Rane, A. and D. Stanzione (2009, 3). Experiences in tuning performance of hybrid MPI/OpenMP applications on quad-core systems. In *10th LCI International Conference on High-Performance Clustered Computing*, Boulder, CO, USA, USA.

[28] Shende, S. and A. D. Malony (Summer 2006). The tau parallel performance system. *International Journal of High Performance Computing Applications, SAGE Publications, 20(2):287-331*.

[29] Siddha, S. (2008). Multicore and the linux kernel.

[30] Sistare, S. (2002). A new open resource management architecture in the sun hpc cluster-tools environment. Technical report, Sun Microsystems.

[31] Sutter, H. (2005). The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal 30*(3).

[32] Tulloch, P. (2007). Discussing the many-core future.

# Appendix A

## MULTI-DIMENSIONAL DATA DECOMPOSITION IN OPENMP

As mentioned in section 3.2, using multi-dimensional decomposition helps when dealing with large datasets. However, OpenMP, by itself, does not provide any means to implement multi-dimensional decomposition. Using `#pragma omp for` causes the following loop to be parallelized among threads. Inserting another `#pragma omp for` is an example of nested parallelism and not implementing multi-dimensional decomposition. Hence, a series of macros were written to be able to implement 2D and 3D decomposition when using OpenMP or hybrid MPI/OpenMP.

```
#include <math.h>

typedef struct
{
        long x, y, z;
        long *highestDim, *lowestDim;
} threeDim;

typedef struct
{
        threeDim id;
        threeDim size;          /* Actual size of current block */
        threeDim fullsize;      /* Desired size of block */
        threeDim numBlocks;
} blockInfo;

typedef struct
{
        threeDim size;
        threeDim ghostrows;
} gridInfo;

enum { X_CONSECUTIVE, Z_CONSECUTIVE };

#define MIN(a,b)        (a) < (b) ? (a) : (b)

#define INIT_NUM_BLOCKS(block, grid, threadId, consecDim)
                                        \
        do {
                                                \
                block.numBlocks.x = ceil ((( double) grid.size.x-2*grid.ghostrows.y)/block.
                    fullsize.x);                \
                block.numBlocks.y = ceil ((( double) grid.size.y-2*grid.ghostrows.x)/block.
                    fullsize.y);                \
```

```
                block.numBlocks.z = ceil  (((double) grid.size.z-2*grid.ghostrows.z)/block.
                    fullsize.z);                    \
                block.id.x = 0;
                                                    \
                block.id.y = 0;
                                                    \
                block.id.z = 0;
                                                    \
                if (consecDim == X_CONSECUTIVE)
                                                    \
                        block.id.lowestDim=&block.id.x, block.id.highestDim=&block.id.z,
                                                        \
                        block.size.lowestDim=&block.size.x, block.size.highestDim=&block.
                            size.z,                     \
                        block.fullsize.lowestDim=&block.fullsize.x, block.fullsize.
                            highestDim=&block.fullsize.z,           \
                        block.numBlocks.lowestDim=&block.numBlocks.x, block.numBlocks.
                            highestDim=&block.numBlocks.z; \
                else
                                                    \
                        block.id.lowestDim=&block.id.z, block.id.highestDim=&block.id.x,
                                                        \
                        block.size.lowestDim=&block.size.z, block.size.highestDim=&block.
                            size.x,                     \
                        block.fullsize.lowestDim=&block.fullsize.z, block.fullsize.
                            highestDim=&block.fullsize.x,           \
                        block.numBlocks.lowestDim=&block.numBlocks.z, block.numBlocks.
                            highestDim=&block.numBlocks.x; \
                NEXT_BLOCK_INFO (block, threadId)
                                                    \
        } while(0);

#define NEXT_BLOCK_INFO(block, numThreads)
                                    \
        do {
                                                \
                *block.id.lowestDim = *block.id.lowestDim + numThreads;
                                                    \
                block.id.y = block.id.y + *block.id.lowestDim / *block.numBlocks.
                    lowestDim;                      \
                *block.id.highestDim = *block.id.highestDim + block.id.y/block.numBlocks.y
                    ;                               \
                if (*block.id.highestDim >= *block.numBlocks.highestDim)
                                                    \
                        *block.id.highestDim = -1;
                                                        \
```

```
                else                                                    \
                        *block.id.highestDim = *block.id.highestDim % *block.numBlocks.
                            highestDim;                                 \
                    *block.id.lowestDim %= *block.numBlocks.lowestDim;  \
                    block.id.y %= block.numBlocks.y;                    \
                    block.size.x = MIN(grid.size.x-2*grid.ghostrows.y-block.id.x*block. fullsize
                        .x, block. fullsize .x);                        \
                    block.size.y = MIN(grid.size.y-2*grid.ghostrows.x-block.id.y*block. fullsize
                        .y, block. fullsize .y);                        \
                    block.size.z = MIN(grid.size.z-2*grid.ghostrows.z-block.id.z*block. fullsize
                        .z, block. fullsize .z);                        \
            } while(0);

#define GLOBAL_X(block, grid)  block. fullsize .x*block.id.x + grid.ghostrows.y
#define GLOBAL_Y(block, grid)  block. fullsize .y*block.id.y + grid.ghostrows.x
#define GLOBAL_Z(block, grid)  block. fullsize .z*block.id.z + grid.ghostrows.z

#define ALL_BLOCKS_PROCESSED(block) (*block.id.highestDim == -1)

// Get pointers, find max according to value and finally return the pointer
#define MAX_OF_THREE(lp01, lp02, lp03) (*lp01 > *lp02 ? (*lp01 > *lp03 ? lp01 : lp03) : (*
    lp02 > *lp03 ? lp02 : lp03))

/* The following is a function instead of a macro because it is to be called only once for
    every grid, while initializing blocks */
void gridToBlockSize (gridInfo grid, blockInfo* lpBlock)
{
        // Assumes 4 threads under an MPI task, but thats the case with Saguaro and Ranger
            anyway

        if (!lpBlock)
                return;

        lpBlock->fullsize.x = grid.size.x-2*grid.ghostrows.y, lpBlock->fullsize.y = grid.
            size.y-2*grid.ghostrows.x, lpBlock->fullsize.z = grid.size.z-2*grid.ghostrows.z
            ;
        long* max = MAX_OF_THREE (&lpBlock->fullsize.x, &lpBlock->fullsize.y, &lpBlock
            ->fullsize.z);
        *max = *max / 2;            // Dividing the dimension that has the largest size

        long* newMax = MAX_OF_THREE (&lpBlock->fullsize.x, &lpBlock->fullsize.y, &
            lpBlock->fullsize.z);
        if (newMax != max)      // Common case, no jumps: X, Y, Z are not all equal
```

```
        {
                *newMax = *newMax / 2;
                return;
        }
        else
        {
                // Choose a different dimension
                if (newMax == &lpBlock->fullsize.x)     { lpBlock->fullsize.y = lpBlock->
                    fullsize.y / 2; return; }
                if (newMax == &lpBlock->fullsize.y)     { lpBlock->fullsize.z = lpBlock->
                    fullsize.z / 2; return; }
                if (newMax == &lpBlock->fullsize.z)     { lpBlock->fullsize.x = lpBlock->
                    fullsize.x / 2; return; }
        }

        // returns the block size that each thread should operate on
        // passed through parameters themselves
}
```

The above macros cause the entire data (called as “grid”) to be split into “blocks” of desired size. These blocks are then distributed among the threads in a round-robin fashion. As an example, for a 2D data grid of size 300x300 and a desired block size of 90x90, the grid would be split into blocks as shown in figure A.1.

The numbers in figure A.1 are the identification numbers assigned to blocks. These are distributed among threads in a round-robin fashion, i.e. assuming 4 threads, block #1 is assigned to thread 0, block #2 to thread 1, block #4 to thread 3, block #5 to thread 0 and so on.

The following code snippet shows how the macros are to be used.

```
/* globals */
gridInfo grid;
blockInfo block;

int threadId, numThreads;
#pragma omp threadprivate(threadId, block)

void processData ()
{
        /* Specify dimensions of the grid to be processed */
        /* Here, we have a 2D grid, so size in Z-dimension is 1 */
        grid.size.z = 1;
```

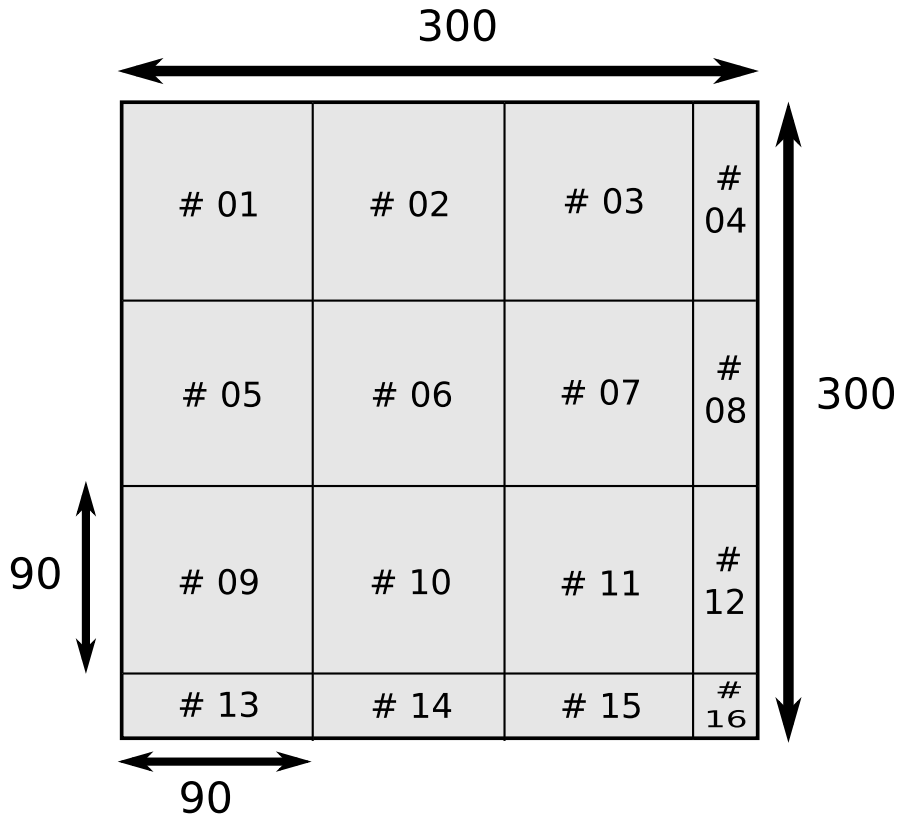



Figure A.1. Using Multi-Dimensional Decomposition To Split A Data Grid Into Smaller Blocks

```
grid.size.y = localSize [DIM_Y];
grid.size.x = localSize [DIM_X];

/* Should we skip over bordering rows and columns of the grid? */
grid.ghostrows.z = 0;
grid.ghostrows.y = 1;
grid.ghostrows.x = 1;

#pragma omp parallel
{
        threadId = omp_get_thread_num();
        numThreads = omp_get_num_threads();

        gridToBlockSize (grid, &block);
}

/* Finished initializing, now process the data (grid), a block at a time */
#pragma omp parallel
```

```
        {
                int startX, startY;

                /* The grid is continuous in X-dimension */
                INIT_NUM_BLOCKS (block, grid, threadId, X_CONSECUTIVE);
                while (!ALL_BLOCKS_PROCESSED (block))
                {
                        startX = GLOBAL_X (block, grid);
                        startY = GLOBAL_Y (block, grid);

                        endX = startX + block.size.x;
                        endY = startY + block.size.y;

                        for (i=startY; i<endY; i++)
                                for (j=startX; j<endX; j++)
                                        localGrid [workingCopy][i][j] = 1;

                        NEXT_BLOCK_INFO (block, numThreads);
                }
        }
}
```

# Appendix B

## PERFORMANCE COUNTER INFORMATION FOR DWARFS

This appendix lists the information obtained from performance counters while running the MPI and hybrid versions of the dwarfs for different problem sizes. For each problem size, the tables show:

- `TOT_INS`: Total number of instructions executed
- `TOT_CYC`: Total number of cycles spent in execution
- `RES_STL`: Total number of resource stalls (due to cache misses, pipeline stalls, etc.)

This information is presented for both communication and computation cycles.

Table B.1. MPI Jacobi Solver With Problem Size As 30K x 30K

| | Communication | | | Computation | | |
|---|---|---|---|---|---|---|
| #cores | TOT_INS | TOT_CYC | RES_STL | TOT_INS | TOT_CYC | RES_STL |
| 008 | 55589200 | 37986500 | 11687900 | 4012530000 | 8096950000 | 6256358226 |
| 016 | 28917500 | 22533500 | 9221090 | 2006320000 | 4048044774 | 3128664985 |
| 032 | 12690800 | 10940114 | 4958860 | 1003027947 | 2025850000 | 1566224402 |
| 064 | 70106292 | 46887000 | 15924200 | 501482000 | 1016070000 | 786233000 |
| 128 | 23932213 | 17107500 | 6986370 | 250740927 | 505153000 | 390242000 |

Table B.2. MPI Jacobi Solver With Problem Size As 42K x 42K

| | Communication | | | Computation | | |
|---|---|---|---|---|---|---|
| #cores | TOT_INS | TOT_CYC | RES_STL | TOT_INS | TOT_CYC | RES_STL |
| 016 | 64424600 | 45372000 | 15671300 | 3932783517 | 7934860000 | 6131340000 |
| 032 | 20161300 | 15140400 | 5704447 | 1966391549 | 3967327837 | 3066394816 |
| 064 | 12670744 | 10811206 | 4905868 | 983287000 | 1982988269 | 1532470000 |
| 128 | 67308400 | 46616400 | 18441200 | 491552414 | 991433432 | 766118000 |

Table B.3. MPI Jacobi Solver With Problem Size As 58K x 58K

| | Communication | | | Computation | | |
|---|---|---|---|---|---|---|
| #cores | TOT_INS | TOT_CYC | RES_STL | TOT_INS | TOT_CYC | RES_STL |
| 032 | 34734472 | 25085500 | 8860850 | 3749389242 | 7562520000 | 5845063617 |
| 064 | 18931900 | 16432700 | 7756680 | 1874820000 | 3780902023 | 2922030000 |
| 128 | 13358700 | 11470600 | 5359190 | 937284000 | 1889392674 | 1459940000 |

Table B.4. MPI Jacobi Solver With Problem Size As 81K x 81K

| | Communication | | | Computation | | |
|---|---|---|---|---|---|---|
| #cores | TOT_INS | TOT_CYC | RES_STL | TOT_INS | TOT_CYC | RES_STL |
| 064 | 42920788 | 36693000 | 17091800 | 3656960000 | 7371540000 | 5696560000 |
| 128 | 19575592 | 16589400 | 7662180 | 1828480000 | 3684050000 | 2847650000 |

Table B.5. MPI Jacobi Solver With Problem Size As 113K x 113K

| | Communication | | | Computation | | |
|---|---|---|---|---|---|---|
| #cores | TOT_INS | TOT_CYC | RES_STL | TOT_INS | TOT_CYC | RES_STL |
| 128 | 57459400 | 47421232 | 21349628 | 3558000000 | 7175620000 | 5548048855 |

Table B.6. Hybrid Jacobi Solver With Problem Size As 30K x 30K

| | Communication | | | Computation | | |
|---|---|---|---|---|---|---|
| #cores | TOT_INS | TOT_CYC | RES_STL | TOT_INS | TOT_CYC | RES_STL |
| 008 | 52777700 | 31540700 | 9151710 | 1979400000 | 8202840000 | 7127990000 |
| 016 | 26164600 | 15656861 | 4613353 | 1179810000 | 4107230000 | 3473520000 |
| 032 | 14504700 | 9925357 | 3776110 | 589623000 | 2024340000 | 1707220000 |
| 064 | 17403018 | 13207577 | 6071001 | 294711092 | 1011530000 | 852605951 |
| 128 | 2954080 | 3086010 | 1788810 | 147973000 | 508122241 | 427413000 |

Table B.7. Hybrid Jacobi Solver With Problem Size As 42K x 42K

| | Communication | | | Computation | | |
|---|---|---|---|---|---|---|
| #cores | TOT_INS | TOT_CYC | RES_STL | TOT_INS | TOT_CYC | RES_STL |
| 016 | 53254915 | 31022400 | 8752270 | 1808666317 | 8070700000 | 7096534500 |
| 032 | 27568199 | 18321900 | 6671060 | 1730226962 | 4137510000 | 3209910223 |
| 064 | 16506200 | 12022400 | 5112068 | 495130000 | 2039400000 | 1772514644 |
| 128 | 6112782 | 5478948 | 2915470 | 247443715 | 1021310000 | 887414000 |

Table B.8. Hybrid Jacobi Solver With Problem Size As 58K x 58K

| | Communication | | | Computation | | |
|---|---|---|---|---|---|---|
| #cores | TOT_INS | TOT_CYC | RES_STL | TOT_INS | TOT_CYC | RES_STL |
| 032 | 53752400 | 34849800 | 12203500 | 2359610000 | 7786092919 | 6518830000 |
| 064 | 31119000 | 22627044 | 9819520 | 854942000 | 3780030000 | 3318574050 |
| 128 | 13574362 | 11535558 | 5970450 | 427319088 | 1893700000 | 1662834262 |

Table B.9. Hybrid Jacobi Solver With Problem Size As 81K x 81K

| | Communication | | | Computation | | |
|---|---|---|---|---|---|---|
| #cores | TOT_INS | TOT_CYC | RES_STL | TOT_INS | TOT_CYC | RES_STL |
| 064 | 42467500 | 30047700 | 12599900 | 3224643423 | 7506151428 | 5777340000 |
| 128 | 26465732 | 20123300 | 9303450 | 835572000 | 3716180000 | 3265520000 |

Table B.10. Hybrid Jacobi Solver With Problem Size As 113K x 113K

| | Communication | | | Computation | | |
|---|---|---|---|---|---|---|
| #cores | TOT_INS | TOT_CYC | RES_STL | TOT_INS | TOT_CYC | RES_STL |
| 128 | 35245300 | 26967049 | 12370500 | 1665640000 | 7258773815 | 6360878375 |

Table B.11. MPI Computation Of Pi Using Monte-Carlo Method

| | Communication | | | Computation | | |
|---|---|---|---|---|---|---|
| #cores | TOT_INS | TOT_CYC | RES_STL | TOT_INS | TOT_CYC | RES_STL |
| 008 | 17008474078 | 9209370000 | 2465250470 | 291677572405 | 331455000000 | 120765773416 |
| 016 | 8433356947 | 4557988703 | 1189492336 | 145838789982 | 165756000000 | 60401833043 |
| 032 | 3794572063 | 2287860000 | 723174135 | 72919395649 | 82863700000 | 30190508043 |
| 064 | 2037533775 | 1268000000 | 477317039 | 36459699981 | 41433300000 | 15098543039 |
| 128 | 919860353 | 585379000 | 242620400 | 18229852217 | 20715600000 | 7548017797 |

Table B.12. Hybrid Computation Of Pi Using Monte-Carlo Method

| | Communication | | | Computation | | |
|---|---|---|---|---|---|---|
| #cores | TOT_INS | TOT_CYC | RES_STL | TOT_INS | TOT_CYC | RES_STL |
| 008 | 1602479229 | 927557000 | 270258159 | 355677581432 | 343576000000 | 99953828906 |
| 016 | 4559339924 | 2742622666 | 821560347 | 177838797129 | 171756696772 | 49940468108 |
| 032 | 2729145316 | 1912627493 | 778340994 | 88919402628 | 85874107301 | 24981261339 |
| 064 | 1248023650 | 988461000 | 387847796 | 44507596287 | 42942300000 | 12481624499 |
| 128 | 594199272 | 489569986 | 206318304 | 22295295382 | 21499600000 | 6242634707 |

Table B.13. MPI Computation Of Sparse Matrix/Vector Multiplication

| | Communication | | | Computation | | |
|---|---|---|---|---|---|---|
| #cores | TOT_INS | TOT_CYC | RES_STL | TOT_INS | TOT_CYC | RES_STL |
| 008 | 91209467 | 423464000 | 378264400 | 15308194 | 70493955 | 64185813 |
| 016 | 438118940 | 501977000 | 325029722 | 7654300 | 33970127 | 30886004 |
| 032 | 427182710 | 483674000 | 302254494 | 3827352 | 16182789 | 14601338 |
| 064 | 305579031 | 362581000 | 234663094 | 1913878 | 5885190 | 4983481 |
| 128 | 349628419 | 365515000 | 225505147 | 957144 | 1970100 | 1553214 |

Table B.14. Hybrid Computation Of Sparse Matrix/Vector Multiplication

| | Communication | | | Computation | | |
|---|---|---|---|---|---|---|
| #cores | TOT_INS | TOT_CYC | RES_STL | TOT_INS | TOT_CYC | RES_STL |
| 008 | 55667952 | 354877000 | 312399280 | 13612614 | 74051119 | 68450156 |
| 016 | 493396387 | 497245036 | 284549302 | 6896611 | 36658201 | 33766835 |
| 032 | 629271834 | 583679000 | 332032903 | 3496721 | 18042407 | 16486016 |
| 064 | 634290555 | 657912000 | 416496616 | 1836061 | 8831426 | 7915625 |
| 128 | 550936674 | 597164000 | 395043610 | 993372 | 4070860 | 3491597 |

Table B.15. MPI Fast Fourier Transform (2D) With Problem Size As 20K x 20K

| | Communication | | | Computation | | |
|---|---|---|---|---|---|---|
| #cores | TOT_INS | TOT_CYC | RES_STL | TOT_INS | TOT_CYC | RES_STL |
| 008 | 1023947115 | 587166251 | 191283285 | 382395419 | 314640624 | 56329194 |
| 016 | 564010678 | 354134789 | 126789175 | 201056404 | 160002000 | 26292538 |
| 032 | 403616770 | 339783189 | 115298409 | 101913480 | 80380146 | 13881889 |
| 064 | 927998823 | 559099663 | 262765739 | 52606508 | 41036739 | 6998440 |
| 128 | 1359783014 | 920720000 | 421443226 | 27764800 | 21489151 | 3829197 |

Table B.16. MPI Fast Fourier Transform (2D) With Problem Size As 28K x 28K

| | Communication | | | Computation | | |
|---|---|---|---|---|---|---|
| #cores | TOT_INS | TOT_CYC | RES_STL | TOT_INS | TOT_CYC | RES_STL |
| 016 | 1465984186 | 933785000 | 305404376 | 393633536 | 308936000 | 49554899 |
| 032 | 637666151 | 457452000 | 180036622 | 199365350 | 155633000 | 25132015 |
| 064 | 958943557 | 888770000 | 279263641 | 102571473 | 79266800 | 12531784 |
| 128 | 2099725636 | 1408880648 | 651558336 | 53373367 | 40752000 | 6707867 |

Table B.17. MPI Fast Fourier Transform (2D) With Problem Size As 39K x 39K

| | Communication | | | Computation | | |
|---|---|---|---|---|---|---|
| #cores | TOT_INS | TOT_CYC | RES_STL | TOT_INS | TOT_CYC | RES_STL |
| 032 | 1405669316 | 810630000 | 351941906 | 386277407 | 298713000 | 44392591 |
| 064 | 2923750188 | 1840775369 | 807643692 | 198320010 | 150794000 | 22986953 |
| 128 | 2918416167 | 2100407175 | 880553934 | 102255701 | 77325445 | 11756624 |

Table B.18. MPI Fast Fourier Transform (2D) With Problem Size As 54K x 54K

| | Communication | | | Computation | | |
|---|---|---|---|---|---|---|
| #cores | TOT_INS | TOT_CYC | RES_STL | TOT_INS | TOT_CYC | RES_STL |
| 064 | 4758169494 | 3202310000 | 1391832653 | 379354894 | 286778958 | 41267333 |
| 128 | 4761236968 | 3477150000 | 1412912676 | 194422700 | 146198460 | 21082409 |

Table B.19. MPI Fast Fourier Transform (2D) With Problem Size As 75K x 75K

| | Communication | | | Computation | | |
|---|---|---|---|---|---|---|
| #cores | TOT_INS | TOT_CYC | RES_STL | TOT_INS | TOT_CYC | RES_STL |
| 128 | 7961605127 | 5138510000 | 2307582745 | 372904649 | 278601000 | 40110080 |

Table B.20. Hybrid Fast Fourier Transform (2D) With Problem Size As 20K x 20K

| | Communication | | | Computation | | |
|---|---|---|---|---|---|---|
| #cores | TOT_INS | TOT_CYC | RES_STL | TOT_INS | TOT_CYC | RES_STL |
| 008 | 40470896 | 11175400 | 10277481 | 2727936876 | 2340514251 | 615239237 |
| 016 | 56374199 | 34392636 | 14943059 | 1363974950 | 1086208020 | 255785321 |
| 032 | 13078542 | 11801000 | 5149527 | 681994170 | 586044000 | 155599700 |
| 064 | 29714552 | 30115600 | 10972742 | 341003892 | 292554163 | 74910314 |
| 128 | 78971524 | 56207400 | 28795600 | 170508641 | 135822000 | 32054364 |

Table B.21. Hybrid Fast Fourier Transform (2D) With Problem Size As 28K x 28K

| | Communication | | | Computation | | |
|---|---|---|---|---|---|---|
| #cores | TOT_INS | TOT_CYC | RES_STL | TOT_INS | TOT_CYC | RES_STL |
| 016 | 74252150 | 56955700 | 19835830 | 2673152472 | 2126940000 | 502124715 |
| 032 | 44833985 | 41498900 | 15931269 | 1336582807 | 1143000000 | 296662024 |
| 064 | 45062090 | 38925400 | 16597863 | 668298069 | 574211590 | 150334520 |
| 128 | 107665642 | 80246001 | 39368328 | 334155810 | 265933000 | 62563233 |

Table B.22. Hybrid Fast Fourier Transform (2D) With Problem Size As 39K x 39K

| | Communication | | | Computation | | |
|---|---|---|---|---|---|---|
| #cores | TOT_INS | TOT_CYC | RES_STL | TOT_INS | TOT_CYC | RES_STL |
| 032 | 65485202 | 38480100 | 21627480 | 2593765523 | 2228830000 | 591648695 |
| 064 | 117070679 | 85847600 | 41481627 | 1296442760 | 1113640000 | 293584529 |
| 128 | 134140855 | 103434000 | 50296915 | 648228199 | 515181754 | 121095099 |

Table B.23. Hybrid Fast Fourier Transform (2D) With Problem Size As 54K x 54K

| | Communication | | | Computation | | |
|---|---|---|---|---|---|---|
| #cores | TOT_INS | TOT_CYC | RES_STL | TOT_INS | TOT_CYC | RES_STL |
| 064 | 158836294 | 104323000 | 57533929 | 2485658216 | 2139040000 | 565699597 |
| 128 | 225385180 | 166447000 | 82757679 | 1242692008 | 987375000 | 231962376 |

Table B.24. Hybrid Fast Fourier Transform (2D) With Problem Size As 75K x 75K

| | Communication | | | Computation | | |
|---|---|---|---|---|---|---|
| #cores | TOT_INS | TOT_CYC | RES_STL | TOT_INS | TOT_CYC | RES_STL |
| 128 | 348377197 | 259348000 | 127964848 | 2397082649 | 1903159660 | 446226424 |

Table B.25. MPI Matrix Multiplication

| | Communication | | | Computation | | |
|---|---|---|---|---|---|---|
| size | TOT_INS | TOT_CYC | RES_STL | TOT_INS | TOT_CYC | RES_STL |
| 256 | 156888 | 169353 | 97558 | 2492600 | 1179080 | 506235 |
| 512 | 565627 | 1201400 | 968546 | 19402681 | 9266400 | 3975150 |
| 768 | 1372170 | 3180918 | 2614095 | 64886500 | 30704761 | 13154547 |
| 1024 | 5221422 | 8448375 | 6235799 | 153100000 | 71166725 | 31154100 |
| 1280 | 8606910 | 17272200 | 13566018 | 298198000 | 139267564 | 61754800 |
| 1536 | 21217700 | 30765848 | 21743000 | 514338000 | 240790033 | 107892000 |
| 1792 | 4944320 | 31012700 | 28850700 | 815674000 | 381673000 | 171085939 |
| 2048 | 41918900 | 50401311 | 32748500 | 1216360000 | 585238032 | 273273137 |
| 2304 | 301792000 | 201956000 | 75937700 | 1741230000 | 844518000 | 393940000 |
| 2560 | 177174870 | 133619658 | 59463000 | 2385583169 | 1226867423 | 604565000 |
| 2816 | 179806000 | 149313923 | 74468200 | 3172020000 | 1636695546 | 811061000 |
| 3072 | 203526000 | 170575000 | 86489000 | 4114700000 | 2223200000 | 1157040000 |
| 3328 | 338238000 | 241226000 | 100215711 | 5227770000 | 2750575023 | 1395830000 |
| 3584 | 216435000 | 215091326 | 124703000 | 6525390000 | 3273839487 | 1589020000 |
| 3840 | 242394000 | 211042000 | 109694974 | 8021720000 | 4059580000 | 1991454860 |
| 4096 | 121216855 | 161000788 | 110517000 | 9730900000 | 6770910000 | 4263900000 |
| 4352 | 539845000 | 414479000 | 188734000 | 11705100000 | 6358814862 | 3330671132 |
| 4608 | 82046900 | 212430000 | 178216000 | 13887100000 | 7790770000 | 4182330067 |
| 4864 | 1009710000 | 731330000 | 318219297 | 16324700000 | 8523331767 | 4289340506 |
| 5120 | 71301136 | 205871519 | 177353957 | 19031980936 | 11211750240 | 6287821814 |

Table B.26. Hybrid Matrix Multiplication

| | Communication | | | Computation | | |
|---|---|---|---|---|---|---|
| size | TOT_INS | TOT_CYC | RES_STL | TOT_INS | TOT_CYC | RES_STL |
| 256 | 171333 | 198927 | 124723 | 2722625 | 1337060 | 500939 |
| 512 | 497982 | 790216 | 575703 | 21369795 | 9927700 | 3164701 |
| 768 | 1815220 | 2281610 | 1521660 | 71670900 | 32121400 | 9830800 |
| 1024 | 2403025 | 3280480 | 2270050 | 169354446 | 76463200 | 24250200 |
| 1280 | 4328400 | 7297990 | 5477928 | 332639875 | 148666524 | 46439400 |
| 1536 | 5280457 | 9514632 | 7268054 | 573362334 | 255284000 | 77063900 |
| 1792 | 10121163 | 15659300 | 11419200 | 908850000 | 403810000 | 122965099 |
| 2048 | 15989600 | 17174577 | 10135400 | 1354830000 | 603900000 | 186628528 |
| 2304 | 15352900 | 17485567 | 10652800 | 1935100000 | 866616000 | 271108913 |
| 2560 | 17561000 | 23858000 | 16013463 | 2651130000 | 1180420000 | 359708000 |
| 2816 | 279905531 | 184307401 | 67617600 | 3525040000 | 1577100000 | 488738000 |
| 3072 | 20295600 | 24728900 | 15487300 | 4572550000 | 2050325592 | 641270776 |
| 3328 | 12409128 | 28769000 | 22767600 | 5826210000 | 2604157433 | 811295000 |
| 3584 | 26771900 | 37797600 | 25716130 | 7270790000 | 3237910000 | 988703000 |
| 3840 | 30460500 | 53292958 | 39600100 | 8936360000 | 3982390000 | 1225810000 |
| 4096 | 40245920 | 48978324 | 30970764 | 10838643320 | 4840870000 | 1504148083 |
| 4352 | 41194441 | 48054400 | 29653600 | 13022100000 | 5828730000 | 1820340000 |
| 4608 | 56624700 | 57856800 | 32884359 | 15448500000 | 6888501470 | 2116600000 |
| 4864 | 56125589 | 69199700 | 44250800 | 18158900000 | 8095170000 | 2494875686 |
| 5120 | 113209000 | 98068249 | 49177300 | 21169200000 | 9461849338 | 2942170000 |