

A Many-Core Software Framework for Embedded Space Computing

by

Eugene Yu-Ting Sun

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

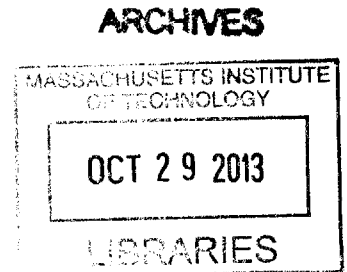
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2013

©2013 Eugene Yu-Ting Sun, All rights reserved

The author hereby grants to MIT and The Charles Stark Draper Laboratory, Inc. permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in any part medium now known or hereafter created.



Author
Department of Electrical Engineering and Computer Science
May 21, 2013

Certified by
Joseph A. Kochocki
Group Leader, Avionics Architecture, Draper Laboratory, Inc.
Thesis Supervisor

Certified by
Christopher J. Terman
Senior Lecturer, Department of EECS, MIT
Thesis Co-Supervisor

Accepted by
Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

A Many-Core Software Framework for Embedded Space Computing

by

Eugene Yu-Ting Sun

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2013, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Space computing has long called for powerful yet power-efficient hardware for on-board computation. The emergence of many-core CPUs on a single die provides one potential solution. The development of processors like Maestro strives for the balance between computational power and energy efficiency. However, development in software has not kept up. Not a single dominant programming framework has emerged to allow developers to easily write applications to take advantage of the new multi-core paradigm. As a result, in NASA's technology roadmap, fault management, programmability, and energy management under the new multi-core paradigm have been listed as top challenges. The goal of this thesis is to develop a framework that streamlines programming for multi-core processors, in the form of a programming model and a C++ programming library. A 49-core Maestro Development Board (MDB) serves as the development and testing hardware platform. The framework's usability is tested through a software simulation of a vision-based crater recognition algorithm for a lunar lander. A parallel version of the algorithm is written under the framework and tested, and a performance gain of about 300%, using 21 Maestro cores, is observed over the RAD750. The uniqueness of this framework lies in the principle that task blocks, not CPU cores, are the fundamental abstraction for individual processes. Each task block is allocated an arbitrary number of CPUs, completes one task, and communicates with other task blocks through message passing. Fault tolerance, power management, and communication among task blocks are abstracted out so that programmers can concentrate on the implementation of the application. The resulting programming library provides developers with the right tools to design and test parallel programs, port serial versions of applications to their parallelized counterparts, and develop new parallel programs with ease.

Thesis Supervisor: Joseph A. Kochocki

Title: Group Leader, Avionics Architecture, Draper Laboratory, Inc.

Thesis Co-Supervisor: Christopher J. Terman

Title: Senior Lecturer, Department of EECS, MIT

Acknowledgments

This thesis has been funded by The Charles Stark Draper Laboratory, Inc. under the project code 27606-001.

I'd like to first thank my supervisor and thesis advisor at Draper, Joe Kochocki. He has spent a great amount of time and energy guiding me through the writing of my thesis. I could have never been able to finish it without him. I would also like to thank Gail DiDonato for being my point of contact at Draper's Education Office. Her help throughout my time at Draper has made this thesis possible.

I'd like to thank my thesis co-advisor, Chris Terman, for reading my thesis and giving me valuable feedback. I'd also like to thank my academic advisor, John Guttag, for the years of mentorship at MIT.

Thanks go to my family and friends for their love and support over all these years.

And lastly, I want to thank my girlfriend, Junyao Song, for bringing all the love and joy into my life. Cheers to all our years ahead.

Contents

List of Figures	6
List of Tables	8
1 Introduction	9
1.1 Motivation	10
1.2 Background	12
1.2.1 Many-Core Programming	12
1.2.2 RAD750 TM : The Current Standard	13
1.2.3 Requirements of Space Computing	14
1.3 Solution Overview	16
1.4 Thesis Structure	18
2 MDB: The Hardware Platform	20
2.1 Processor Specification	21
2.1.1 Tile Specification	22
2.1.2 iMesh Network	23
2.1.3 Memory Architecture	25
2.1.4 Power Usage	26
2.2 MDE	27
2.2.1 tile-eclipse: the IDE	27
2.2.2 tile-monitor: the Runtime Monitor	29

3	MAPLES: The Framework	32
3.1	Programming Model	32
3.1.1	Types of Targeted Problems	33
3.1.2	Planned Solutions to the Problems	35
3.1.3	Task-Block Abstraction Design	37
3.1.4	Communication Scheme	39
3.1.5	Program Run-Time Behavior	39
3.1.6	Space Computing Considerations	42
3.2	Intended Platforms	42
4	TRN Crater Recognition	45
4.1	Algorithm Overview	46
4.2	Single-Core Benchmark	48
4.3	Pre-MAPLES Efforts	49
4.4	MAPLES Implementation	53
4.5	Parallelization Result	56
4.6	Discussion	58
5	Conclusion	60
5.1	Contributions	60
5.2	Future Work	61
	Acronyms	64
	References	66

List of Figures

2-1	A Diagram of the Maestro Processor	21
2-2	Detailed View of a Single Maestro Tile	22
2-3	A Screenshot of MDE	28
2-4	A Closer Look at the Grid View Tab	29
3-1	Multiple Applications Running on the Same Board	33
3-2	Illustration of How an Embarrassingly Parallel Problem is Solved	35
3-3	Illustration of How an Parallel Problem is Solved in a MapReduce Way [14]	35
3-4	Class Diagram for MAPLES	38
3-5	Sequence Diagram for MAPLES	41
4-1	An Overview of the TRN Algorithm	46
4-2	An Overview of the TRN Crater Recognition Algorithm	47
4-3	Benchmark Comparison of Unparallelized TRN Crater Recognition	48
4-4	Execution Time Ratios of the TRN Crater Recognition Algorithm	49
4-5	Crude Crater Recognition Parallelization, Design 1	50
4-6	Crude Crater Recognition Parallelization, Attempt 2	50
4-7	Crude Crater Recognition Parallelization on MDB Diagram	51
4-8	Sample Lunar Images used for Benchmarking	52
4-9	Benchmark Results with Varied Number of Cores	52
4-10	Benchmark Comparison between Crude Parallel Run on Multi-Core Maestro and Sequential Run on RAD750 TM	53
4-11	Core Diagram of Task Block Allocation for Crater Recognition	54

4-12 Benchmark Comparison of the Un-Pipelined MAPLES Performance	56
4-13 Benchmark Comparison of the Pipelined MAPLES Performance	57

List of Tables

1.1	RAD750 TM Feature Comparison by Generation	14
2.1	TILE64 TM /Maestro Cache Specifications	26
4.1	Unparallelized Benchmark Comparison Table	48
4.2	Benchmark Comparison Table	57

Chapter 1

Introduction

Since the 1950s, the National Aeronautics and Space Administration (NASA) and other organizations have been launching missions into space. Dozens of Low Earth Orbit (LEO) and Geosynchronous Orbit (GEO) missions have been conceived and launched to support a broad range of missions including Earth-based communications, Global Positioning System (GPS), deep space observation and communications, Earth observation and national security. NASA missions typically go well beyond Earth local missions such as the International Space Station and Hubble Space Telescope to perform missions within the solar system of general interest to humanity. Much attention has been focused recently on missions to Mars to study that planet's surface and atmospheric chemistry and to look for signs of water, and hence signs of life. Missions are already being envisioned to study the asteroid belt and other planets including Jupiter's moons. Though each mission has its unique requirements, all of them require spaceborne computing platforms that can meet the mission goals and deliver the necessary science value back to Earth for investigation.

This thesis inspects the challenges faced by the community of space computing in its efforts to meet next-generation mission requirements, specifically in the domain of computational performance. The challenges in the field of embedded space computing are surveyed. A solution is offered, and the implementation of the solution is attempted, benchmarked, and analyzed.

1.1 Motivation

Every ten years the National Research Council conducts the Decadal Survey for NASA. The latest one reported out in 2011 covers the period from 2013 - 2022. The survey identified a broad set of 25 planetary mission of interest to the scientific community that include: a Venus Lander to study that planet's surface and atmospheric composition; a Mars Sample Return to bring back specimens from the Mars surface to learn more about surface and atmospheric chemistry and the search for life there; a Trojan Tour of the Asteroid Belt to examine the potential for landing on an asteroid and future applications (e.g. mining) there. These missions will require the following capabilities:

- Entry, Descent and Landing
- Small Body Proximity Operations
- Autonomous Landing Hazard Avoidance
- Enhanced Guidance, Navigation & Control
- Autonomous/Goal-Based Mission Planning
- Fast Surface Mobility
- Surface Science during Traverse
- On-board Data Product Generation
- Science Event Detection and Response

Increased computational performance will greatly enhance mission capability and science value return, particularly in the case of un-manned autonomous missions which, by necessity, will require little to no ground-based intervention for mission success.

Many current NASA missions incorporate the 1990's era BAE RAD750TM processor into their technical solutions. The RAD750TM has been a rock-solid processing capability for near Earth missions as well as planetary missions for nearly two decades, delivering ~ 240 Dhrystone Millions of Instructions Per Second (MIPS) and 70-80 Mega Floating-point Operations Per Second (MFLOPS). Unfortunately, it cannot supply the computational performance envisioned for many if not most future space missions described above. NASA's

Human Exploration and Operation Missions Directorate (HEOMD) as well as the Space Missions Directorate (SMD), have established future High Performance Spaceflight Computing needs as documented in the HPSC Avionics Formulation Task Study Report [7]. Some of Key Performance Parameters (KPP) included in that study are dictated by very stringent system level mission requirements:

- Computational Performance
- Fault Tolerance
- Radiation Tolerance
- Power and Energy Management
- Programmability and Flight Software Applicability
- Flight Software Verification and Validation (V&V)
- Interoperability
- Extensibility and Evolvability

While some existing multi-core processing solutions meet some of these KPPs such as computational performance, they fall far short in other key critical areas important to NASA. For example, NASA missions often have both severe power restrictions as well as fault tolerance requirements needed to endure long space missions. Future NASA missions will go to new and uncertain environments where system level testing of all possible mission scenarios is not possible a priori. Deep space missions are too far away for time critical command and control from Earth, therefore more autonomy will be introduced into future spacecraft to support these missions. Enhanced spaceflight computing will clearly be a technology multiplier allowing for many more possibilities over today's existing space-based computing platforms by expanding on-board capabilities across a wide set of NASA missions.

This thesis addresses a limited set of the KPPs mentioned above including Computational Performance and Programmability. While some other KPPs were partially addressed such as Fault Tolerance and Power Management, time limited the scope of this thesis to parallel processing on the 49-core Maestro processor and the application of such to Vision-based Navigation (VBN).

1.2 Background

1.2.1 Many-Core Programming

Many-core programming is essentially a synonym with multi-core programming, concurrent programming, parallel programming, and maybe even distributed programming. These terms all refer to a method of software programming for the “form of computation in which many calculations are carried out simultaneously” [2]. The word “many-core programming” put emphasis on using a large number of CPU cores.

In the world of Graphics Processing Unit (GPU), many-core programming has been in developers’ back pockets for quite a long time. A modern graphics card generally has hundreds of GPU cores ready to crunch matrix manipulations. However, in the world of Central Processing Unit (CPU), the state of many-core programming, until recently, can only be described as stale. If Moore’s law didn’t force the CPU world into the paradigm of putting more and more cores onto the board, many-core programming would probably still be absent from our sight. According to a recent survey conducted by UBM® TechWeb on Dr.Dobbs® readers [18, p.2]:

81 percent of developers surveyed report having at least some parallel computing experience... 82 percent of developers now consider parallel programming to be important or even critical for the proper functioning of the software they develop.

Which seems to be a piece of great news. However, in the same survey, 28 percent of developers responded that they don’t use or don’t know about any tool that finds memory defects or does performance tuning, and 66 percent of developers don’t use any tool to find threading defects [18, p.4]. The lack of tool usage may indicate simply personal preferences of these developers in tool usage, but clearly this is not the case. More than half of the surveyed developers want tools to “guide them and help them parallelize their code”, to “avoid incorrect use of parallelization” [18, p.5]. In other words, many developers working on parallel programming are looking for some kind of frameworks that provides both a

model of parallelized thinking and the tools to streamline the coding process.

1.2.2 RAD750TM: The Current Standard

The RAD750TM developed by British Aerospace Engineering (BAE) Systems in the 1990s, is the current industry standard for space computing. BAE's original paper details the development of first generation RAD750TM below [3]:

BAE SYSTEMS has developed the RAD750TM, a fully licensed radiation hardened implementation of the PowerPC 750TM microprocessor, based on the original design database. The processor is implemented in a 2.5 volt, 0.25 micron, six-layer metal Complementary Metal-Oxide Semiconductor (CMOS) technology. Employing a superscalar Reduced Instruction Set Computing (RISC) architecture, processor performance of 260 million Dhrystone 2.1 instructions per second (MIPS) at 133 MHz is provided, while dissipating less than six watts of power. The RAD750TM achieves radiation hardness of 1E-11 upsets/bit-day and is designed for use in high performance spaceborne applications. A new companion Application Specific Integrated Circuit (ASIC), the Power Peripheral Control Interconnect (PCI), provides the bridge between the RAD750TM, the 33 MHz PCI backplane bus, and system memory. The Power PCI is implemented in a 3.3 volt, 0.5 micron, five-layer metal CMOS technology, and achieves radiation hardness of <1E-10 upsets/bit-day.

Since this thesis deals mainly with the software aspect of many-core programming, the focus is placed much more on the computational performance aspect of the processor, rather than the mechanism behind the radiation hardening process.

An improved second generation RAD750TM was also produced, and its features, in comparison to first generation RAD750TM, are listed below [10]:

Feature	1 st Generation RAD750	2 nd Generation RAD750
Technology	250nm RHBD	180nm RHBP
Foundry	Commercial	Rad-Hard
Clock Frequency	133MHz	200MHz
Performance	260MIPS	400MIPS
Vdd (Core/I/O)	2.5/3.3V	1.8/3.3V
Power (Full)	38mW/MHz	25mW/MHz
Power (Doze)	17mW/MHz	12mW/MHz
Power (Sleep)	3.4mW/MHz	1.5mW/MHz
Operating Temp.	-55 ^o to 125 ^o	-55 ^o to 125 ^o
Total Ionizing Dose	200krd(Si) Natural Space	2Mrd(Si) High Dose Rate
Single Event Rate (90% Geo)	~1E-10e/b-d	~1E-11e/b-d
Latch-Up	Immune	Immune

Table 1.1: RAD750TM Feature Comparison by Generation

1.2.3 Requirements of Space Computing

NASA space missions require both a flight segment and a ground segment. Both work hand-in-hand to accomplish mission goals. As NASA probes deeper into space, the communication time increases, necessitating increased spacecraft autonomy. The resulting trend is thus to move computational power from the ground to the spacecraft.

The requirement for each segment has been summarized in NASA's Space Technology Roadmap as below ¹ :

Pinpoint landing, hazard avoidance, rendez-vous-and-capture, and surface mobility are directly tied to the availability of high-performance space-based computing. In addition, multi-core architectures have significant potential to implement scalable computing, thereby lowering spacecraft vehicle mass and power by reducing the number of dedicated systems needed to implement on-board functions.

¹NASA's Space Technology Roadmaps have been in development since 2010, and was finalized early in 2012. In the report, agency specialists outlined the top technical challenges within 14 different areas of space technologies. Computer architecture, including the development of multi-core systems, was included in the report for technology area 11, by the title of *Modeling, Simulation, Information Technology and Processing*.

Flight Computing requires reliable, radiation-hardened platforms which, until recently, have been costly and limited in performance. *Ground Computing* requires supercomputing or other high-performance platforms to support petabyte-scale data analysis and multi-scale physical simulations of systems and vehicles in unique space environments. [6, p.1]

NASA would like to develop flight computing systems that are powerful enough to offload computation from the ground, or more importantly, to have a platform that strikes a balance in computing performance and energy consumption. This requirement is where multi-core computing comes into play. In NASA's words:

Multi-core architectures have significant potential to implement scalable computing, thereby lowering spacecraft vehicle mass and power by reducing the number of dedicated systems needed to implement on-board functions... In addition, multi-core enables numerous system architecture options not feasible with single-core processors. [6, p.9]

Yet, building an effective multi-core hardware system is hardly the main problem. Listed under *Top Technical Challenges*, six out of ten challenges are in the domain of software, with the remaining four being about simulation, modeling, and system architecture. [6, p.5]. Within the software challenges, the top two are *Eliminate the Multi-core "Programmability Gap"* and *System Software for Multi-Core Computing*. The former has to do with the programming method, while the latter has to do with the architecture and guidelines of parallelization. To explain these challenges in NASA's words:

Eliminate the Multi-core "Programmability Gap": For all current programming models, problem analysis and programming choices have a direct impact on the performance of the resulting code. This leads to an incremental, costly approach to developing parallel distributed-memory programs. What has emerged today is a "Programmability Gap", the mismatch between traditional sequential software development and today's multi-core and accelerated computing environments. New parallel languages break with the conventional

programming paradigm by offering high-level abstractions for control and data distribution, thus providing direct support for the specification of efficient parallel algorithms for multi-level system hierarchies based on multi-core architectures.

System Software for Multi-Core Computing: The first multi-core computing chips for space applications are emerging and the time is right to develop architecturally sound concepts for multi-core system software that can use the unique attributes of multi-core computing: software-based approaches to fault tolerance that complement radiation hardening strategies, energy management approaches that account for the energy costs of data movement, and programmability and compiler support for exploiting parallelism. [6, p.6]

It can be seen that in the realm of space computing, moving towards a multi-core architecture demands an immediate breakthrough in the area of software engineering, both in the domain of programming methodology and system architecture. Programmers working in the domain of parallel programming need an effective tool to guide them through the parallelization of applications, not only in designing programs, but also in coming up with parallelization strategies.

1.3 Solution Overview

In order to streamline parallel programming, we have identified several choices to find the ideal tool for programmers working in the domain. This section summarizes and analyzes these design choices.

Building and using a new language may prove to be more costly than necessary. There have already been quite a complete selection of different languages to choose from in the market, ranging from newly developed languages like *Erlang*, or rediscovered gems such as *Haskell*. The similarity all these languages share is their close tie to *Lisp*, and their root in functional programming. Though the ideas in functional programming are extremely helpful in solving multi-core programming problems, the syntax of *Lisp* is unfamiliar to programmers versed

in mainstream languages. The lack of adoption of such languages means less testing, less reliability, and hence less robustness.

Compiler-level auto-parallelization is not a mature technology yet, and its current incarnation imposes many restrictions on the way programs are written. One problem with existing tools such as *R-Stream* and *pMatlab* is the constraint auto-parallelization places on the code. Pointer operations and nested arrays induce errors that break the parallelization paradigm. Also, as high level compilers, they transform serial *C* code into quite unreadable parallel code. The lack of readability can generate high maintenance cost, little room for further optimization, and the dependency on the auto-parallelization tools.

NVIDIA®'s Compute Unified Device Architecture (CUDA) and Intel®'s Threading Building Blocks (TBB) are currently two of the most well-known names in the world of user-level libraries written for the purpose of parallelization. One problem with both libraries is their platform dependency, the former restricted to NVIDIA® graphics cards and the latter chained to Intel® CPUs. The other problem is that both in essence simply provide low-level function calls for programmers to use. The issue of how programmers can best structure their programs is unresolved. This approach opens up the freedom of building from the ground-up, but is not helpful in teaching parallel programming methodologies, or in streamlining the coding process of parallelization.

The solution, we propose, is a software framework that includes both a set of practices and a user-level programming library, tailored towards many-core programming. For the purpose of this thesis, the framework's name is Many-core Programming Library for Embedded Space Computing (MAPLES). The vision of how one takes advantage of such a framework is as follows. By adhering to the principles and restrictions of the framework, the programmers always know what the next step is to build a parallel application. They can quickly transform their ideas of a parallel program unto running code with the library, without knowing from the get-go every detail of the underlying design choices. This framework contains the functional thinking prevalent in parallel programming languages, provides more control than high-level compilers either outputting unreadable code or hiding the parallel nature of the program, and guides the programmers throughout the process of

writing applications for a multi-core platform. The underlying hardware architecture is also quintessential for the working of the framework. The 49-core processor called Maestro is used as the development and testing platform, and serves as the type of processors for which MAPLES is designed. Each core on the Maestro processor is capable of single and double precision floating point operations, runs its own instance of Symmetric Multi Processor (SMP) Linux, and communicates with each other using an underlying connection network called *iMesh* Network. The details about the hardware platform will be discussed later in Chapter 2.

1.4 Thesis Structure

Chapter 2 discusses the many-core hardware platform used to develop MAPLES, namely the Maestro processor. The background behind Maestro's development is inspected, along with its hardware specification, the Maestro development environment, and how the board is set up for the development of MAPLES.

Chapter 3 inspects the details of MAPLES, the software framework built for this thesis. The design principles essential to the framework and the features of the library is discussed.

Chapter 4 discusses the test case used to test MAPLES, specifically, a vision-based crater recognition algorithm developed at Draper. The working of the algorithm, and the parallelization efforts with and without MAPLES, and the benchmarks of its performance are discussed. The framework's effectiveness is showcased here.

Chapter 5 concludes the thesis by discussing the contribution the thesis makes in the world of space computing and parallel programming. Future work to expand the research effort in this thesis is also discussed.

Throughout this thesis, mono-space and italic font types are used for emphasis. A default Latex mono-space typewriter font like `this` is used for names of classes and functions. Italic font like *this* is used for names of programs, languages or products.

Boxes are used to enclose code chunks.

The reader is referred to the appendix for a list of grammar terms used in this thesis.

Chapter 2

Maestro Development Board: The Hardware Platform

The Maestro processor is the result of the On-board Processing Expandable Reconfigurable Architecture (OPERA) program funded by the National Reconnaissance Office (NRO). In 2007, National Reconnaissance Office (NRO) purchased the intellectual property of TILE64TM from Tiler Inc. for the development of Maestro, contracted out the hardware development to Boeing, and the software design to Information Sciences Institute (ISI). The purpose of the program is to “produce a radiation-hardened, state of the art, general purpose processor that’s 100x more capable than current space qualified general purpose processors” [1]. The Maestro processor was designed solely for space computing purposes. Its capability is currently being evaluated by NASA and other organizations.

In the following sections, the different aspects of the processor are discussed, including both the board specification for the hardware and the development environment for the software. The purpose of this chapter is not to explain every aspect of the processor, but to provide an understanding of the development platform, and the sort of hardware platform MAPLES supports.

Figure 2-1 shows an overview of the Maestro Development Board (MDB) [4].

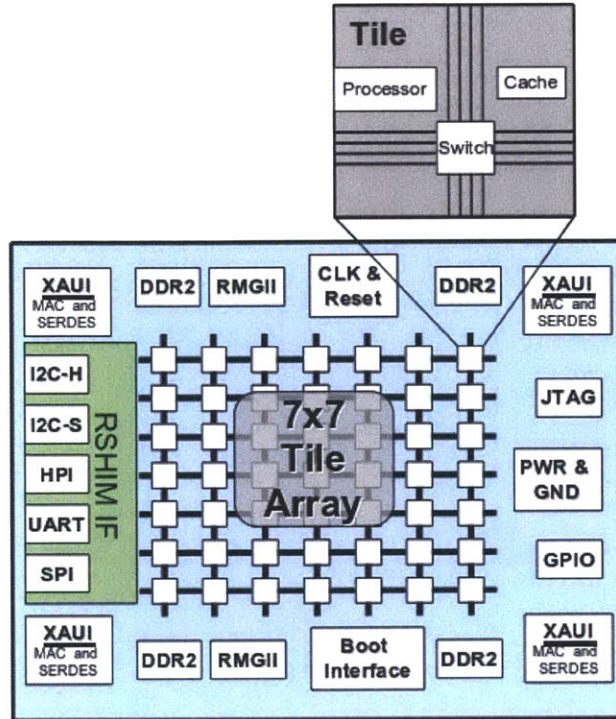


Figure 2-1: A Diagram of the Maestro Processor

2.1 Processor Specification

From OPERA’s program overview presentation [1], it can be seen that the Maestro processor is a modified version of the Tiler TLR26480 processor (TILE64TM), tailored for the purpose of space computing. The number of cores is reduced from 64 to 49 in the process of radiation hardening, and the clock speed is reduced from 600-900 Mega Hertz (MHz) to 440 MHz to save power. An IEEE 754-compliant, single-and-double-precision Floating Point Unit (FPU) is added to every core on the board. The resulting chip is a product of a 90 nm CMOS process, radiation hardened to avoid radiation damages in space, with the computation speed of 65 Giga Operations Per Second (GOPS) and 10 Giga Floating-point Operations Per Second (GFLOPS). Benchmarks have been done by ISI East to showcase Maestro’s performance [17] [15].

2.1.1 Tile Specification

In Tiler’s terminology, each CPU core on the board is called a “tile”. Figure 2-2 shows the components of a single tile [5]. Each tile runs its own instance of SMP Linux, and

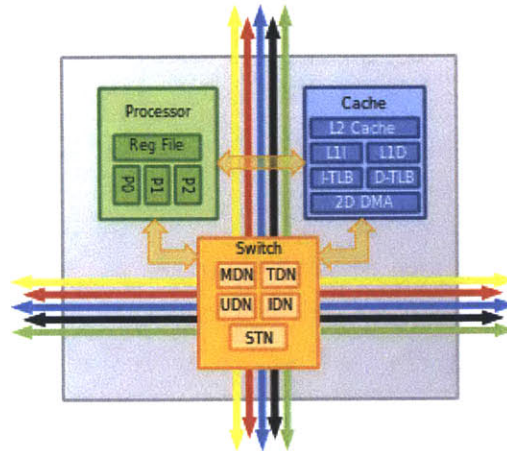


Figure 2-2: Detailed View of a Single Maestro Tile

is composed of a processor, a cache, and a switch engine [20]. According to the OPERA program overview, each tile’s CPU clock frequency is 440 MHz. However, due to thermal clock instability, the clock speed had to be dialed down to 260 MHz.

The processor engine is a conventional three-way Very Long Instruction Word (VLIW) processor with three instructions per bundle and full memory management, protection, and Operating System (OS) support. The Tile Processor includes special instructions to support commonly-used embedded operations in Digital Signal Processing (DSP), video and network packet processing, including:

- Sum of Absolute Values (SAD)
- Hashing and checksums
- Instructions to accelerate encryption
- Single Instruction Multiple Data (SIMD) instructions for sub-word parallelism

The cache engine of each tile contains the tile’s Translation Lookaside Buffer (TLB), caches,

and cache sequencers. For the MDB, each tile has 16 Kilo-Bytes (kB) L1 Cache (8 kB instruction cache and 8 kB data cache), 64 kB L2 cache, and a virtual L3 cache across the board for a total of 5 MB of on chip cache. The Cache Engine also contains a 2D Direct Memory Access (DMA) engine that orchestrates memory data streaming between tiles and external memory, and among the tiles.

The switch engine allows data to be transferred in-between tiles in five independent communication networks. These communication networks are named the *iMesh Network*.

2.1.2 iMesh Network: Inter-Tile Communication

Inter-tile communication is achieved through the board's *iMesh Network*, an essential design behind all the Tiler family processors. In this network, tiles are interconnected to each other via five independent networks: Static Network (STN), Tile Dynamic Network (TDN), User Dynamic Network (UDN), Memory Dynamic Network (MDN), and the I/O Dynamic Network (IDN). Every tile can send a 32-bit word every cycle [4] to any other tile through any of the five network links. Each network's functionality is discussed in the Tiler Processor Architecture Overview [20].

STN comprises the interconnect fabric of the Tiler *iMesh*, and allows extremely low latency transfer of data between tiles. This network is used by the internal software system to efficiently transfer operands from tile to tile, such as user-generated inter-process socket-like streams. STN is not used to move user data.

The other four networks are dynamic networks that transfer user data:

- Tiler user-level APIs are built on the UDN. User applications directly link in *C* library routines to provide a variety of convenient access modes. The UDN supports efficient streaming through a register-level FIFO interface.
- The IDN is used primarily for transfers between tiles and Input/Output (I/O) devices and between I/O devices and memory. A privilege-level protection mechanism limits access to OS-level code.

- The MDN is used for memory data transfer (resulting from loads, stores, prefetches, cache misses or DMA) between tiles themselves, and between tiles and external memory. Only the Cache Engine has a direct hardware connection to the MDN.
- The TDN use is similar to the MDN and supports memory data transfer between tiles. Only the Cache Engine has a direct hardware connection to the TDN.

At user level, UDN is the only network actively used by the application to transfer data between threads running on different tiles. The user-level APIs provide functions to send data between any given pair of tiles. The standard procedure for using UDN requires two phases described below (taken and summarized from Tiler's programming guide [19]).

First, the `tmc_udn_init()` function reserves a rectangle of tiles (referred to as a "UDN hardwall") for use by the calling application. This function should be called by the first task in the application, before creating any child tasks. It takes a `cpu_set_t*` parameter specifying all the CPUs that will access the UDN and allocates the minimum rectangle of tiles that contains those CPUs. If the `cpu_set_t*` is NULL, the routine allocates a rectangle that contains all the CPUs in the calling task block's affinity set. For example, the parent process does the following to create a UDN hardwall:

```
if (tmc_udn_init(NULL) != 0)
    tmc_task_die("Failure in 'tmc_udn_init()'");
```

After creating the UDN hardwall, the application can create threads or fork processes as necessary and bind them to particular CPUs using `tmc_cpus_set_my_cpu()`. Once a task is bound to a single CPU, it should call `tmc_udn_activate()` to enable that CPU to send and receive packets within the previously created UDN hardwall. For example:

```
if (tmc_cpus_set_my_cpu(my_cpu) < 0)
    tmc_task_die("Failure in 'tmc_cpus_set_my_cpu()'");
if (tmc_udn_activate() < 0)
    tmc_task_die("Failure in 'tmc_udn_activate()'");
```

Once the UDN has been activated, the application can use `tmc_udn_header_from_cpu()`

to create packet headers and send packets via the `tmc_udn_send_<N>()` functions. For example, the following code sends a five word data packet to demux queue #2 at `dest_cpu`:

```
DynamicHeader header = tmc_udn_header_from_cpu(dest_cpu);  
tmc_udn_send_5(header, UDN2_DEMUX_TAG, data0, data1, data2, data3, data4);
```

When this packet arrives at the destination core, the UDN hardware strips off the header and tag words and places the five data words into demux queue 2. To receive those five data words, do the following:

```
data0 = tmc_udn2_receive();  
data1 = tmc_udn2_receive();  
data2 = tmc_udn2_receive();  
data3 = tmc_udn2_receive();  
data4 = tmc_udn2_receive();
```

2.1.3 Memory Architecture

Memory for the Maestro processor is composed of two parts: the on-board cache and external memory. The on-board cache is located at each tile's cache engine, as shown in Figure 2-2. The external memory, as shown in Figure 2-1, is placed on two sides of the board, inside the Double Data Rate (DDR) chips.

Table 2.1 shows TILE64TM's cache specification. Same as TILE64TM, Maestro's execution engine does not stall on load or store cache misses. Rather, execution of subsequent instructions continue until the data requested by the cache miss is actually needed by another instruction. This cache subsystem is non-blocking and supports multiple concurrent memory operations.

DMA is implemented as part of the *iMesh* network and the Tiler programming Application Programmer Interface (API).

One thing to note is that since the tiles transfer all data using the underlying iMesh net-

Cache Features	L1I Cache	L1D Cache	L2 Cache
Size	8KB	8KB	64KB
Associativity	Direct-mapped	2-way	2-way
Line Size	64B	16B	64B
Contents	Instructions	Data	Instructions and data
Addressing	Virtually indexed, physically tagged	Physical	Physical
Allocate Policy	Allocate on read miss	Allocate on load miss	Allocate on miss
Write Policy	N/A	Write through, Store update on hit	Writeback
Minimum Write Width	16B	1B	1B
Error Protection	64-bit parity	8-bit parity	8-bit parity
Total On-Chip Cache Size is 5.12MB			

Table 2.1: TILE64TM/Maestro Cache Specifications

work, tiles closer to the edge of the board are closer to DDR chips, and hence faster at memory accesses. This difference in communication latency should also be considered when programming applications that access all the I/O networks.

2.1.4 Power Usage

The power usage of a Maestro board, with all cores turned on, is around 31 Watts (W) at peak and 20 W on average, which is less than ideal in the context of space computing, especially after comparing to the power consumption of 5 W by RAD750TM. There are, however, two ways to justify its use. We can (1) lower the power usage, or (2) raise the performance.

To lower the power output of the board, we can set tiles to enter into sleep mode in software in order to save power. Shutting down each tile will reduce the power usage by 270 mW [1].

Because Maestro uses four to six times the power of a RAD750TM, to match RAD750TM's power efficiency, its performance needs to be at least four to six times of RAD750TM. In NASA Office of the Chief Technologist (OCT)'s technology roadmap, the following goal has been stated [6]:

NASA should seek to concurrently advance the state of the art of two metrics

(sustained throughput and processing efficiency) of high-performance radiation-hardened processors. Goals are throughput greater than 2000 MIPS with efficiency better than 500 MIPS/W.

For a processor that uses about 30 W at peak, this means a throughput of $500 \text{ MIPS/W} \times 30 \text{ W} = 15 \text{ GOPS}$. By advertisement Maestro exceeds this figure (65 GOPS and 10 FLOPS). However, how this theoretical computational throughput translates to execution time in real-world applications is yet unknown.

2.2 Maestro Development Environment (MDE)

The Maestro Development Environment (MDE) is “the multi-core development environment customized for MAESTRO” [12]. It is a suite of applications designed to write code, compile programs, and deploy programs to the MDB. The development environment is based on the Tiler Development Environment. ISI East made the behind-the-scene adjustments for it to work on the Maestro platform. There are three main components of the development environment: the programming Integrated Development Environment (IDE), the compiler and the runtime monitor that connects the MDB and the host machine. For this thesis, the build version *Reservoir.1 for tile64 (2010-01-13)* was used.

2.2.1 tile-eclipse: the IDE

tile-eclipse is a modified version of Eclipse, the popular programming IDE. The interface looks the same as a regular version of Eclipse, except for the addition of the Tiler plugins. Figure 2-3 shows a screenshot of the MDE.

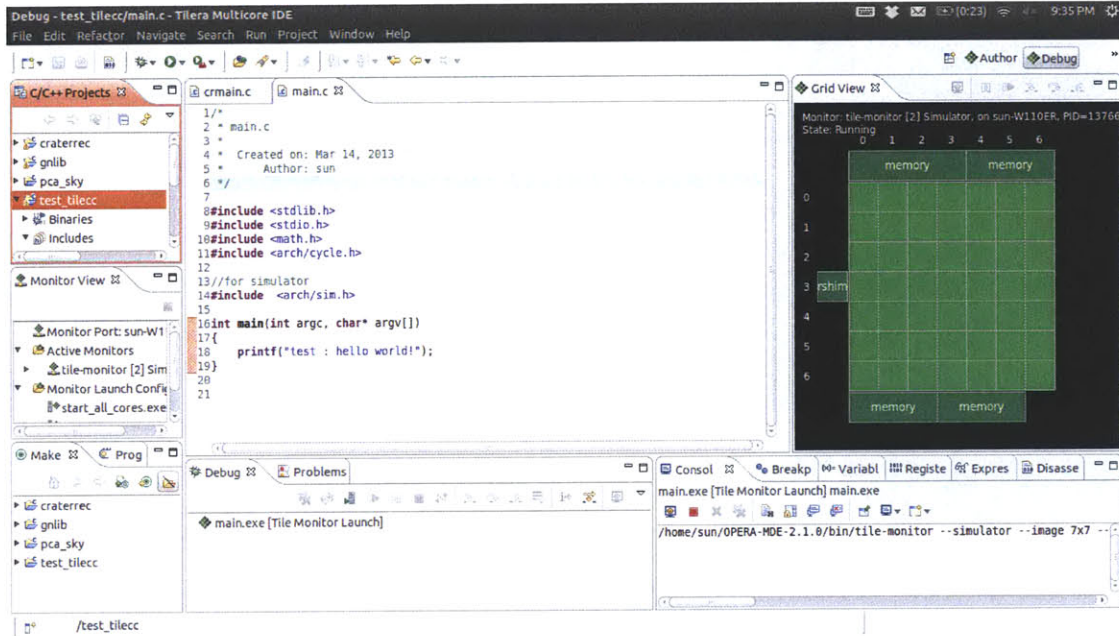


Figure 2-3: A Screenshot of MDE

The notable part about the MDE is the *Grid View* tab on the right side of the User Interface (UI). The tab part of Tileria’s UI plug-ins. The Grid View shows a visual representation of processes running on tiles, either from simulation runs or actual runs on the MDB. A closer look at a running program is shown below:

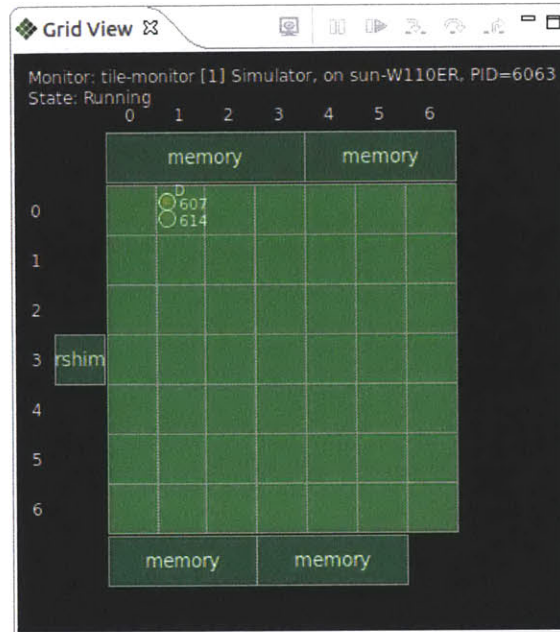


Figure 2-4: A Closer Look at the Grid View Tab

From the figure we can see there are two processes running on tile(0,1) on the board. The number 607 and 614 on top of tile(0,1) are the Process Identification (PID) numbers of the running processes, and the letter D indicates that the application is running in debug mode. The PID indicator appears when the process is running, and disappears when the processes finish running. When thinking about parallelization strategies and debugging, having a tool like this to visualize tiles' start and stop functionality is a very helpful.

2.2.2 `tile-monitor`: the Runtime Monitor

tile-monitor is the communication software used by the host machine to talk to the Maestro board. For board simulations, it simulates the ROM images, uploads program files to the image instance, runs them in a virtual environment, and displays the output to the host machine's command line. For actual board runs, it uploads program files to the board through the physical PCI-E connection, runs them on the board, and displays the output to the host machine's command line. The program can be run on the command line with the following syntax:

```
tile-monitor \  
  <target option\  
  <preparation commands at target\  
  <debugging option\  
  <run option\  
  --quit
```

<target option> indicates whether the program is being run on the physical processor board or on the simulated virtual machine. To run the program on the actual board, <target option> is substituted by `--pci --resume`, in which `pci` indicates that the program will be transferred via PCI connection, and `resume` indicates that after the program finishes running, the environment is switched back to the host machine instead of staying in the virtual environment. Similarly, to run the program in simulation, <target option> is substituted by `--simulator --image <path to image file>`, with the path to the ROM image provided on the command.

<preparation commands at target> indicates preparation commands that will be executed before the program is run. Typically its role is to create folders to contain the program files and upload the program files from the host machine to the target location. Commands are entered line-by-line, with every line being preceded by the argument sign `--`.

<debugging option> indicates whether the program will be run with debugger, and to which core is the debugger attached. Typical options include `--debug-on-crash` and `debug-tile <x,y>`, with <x,y> indicating the x and y coordinate of the core on the board.

<run option> is usually the last argument in a *tile-monitor* command, as it simply gives the final go-ahead to run programs, in the format of `--run <path to program>`.

A sample program may look like this:

```
/usr/bin/tile-monitor \  
  --simulator \  
  --image-file /home/sun/OPERA-MDE-2.1.0/lib/boot/7x7.image \  
  --mkdir /home/sun/workspace/tests \  
  --run <path to program>
```

```
--cd /home/sun/workspace/tests \  
--upload /home/sun/workspace/tests/start_all_cores.exe /home/sun/workspace/tests/  
    start_all_cores.exe \  
--debug-on-crash \  
--debug-tile 0,0 \  
--run +- /home/sun/workspace/tests/start_all_cores.exe +- \  
--quit
```

This sample script shows how one might simulate a run of the test program `start_all_cores.exe` for a 7×7 Maestro board Read Only Memory (ROM) image.

Chapter 3

MAPLES: The Framework

MAPLES, the Many-core Programming Library for Embedded Space Computing, is described in this chapter. MAPLES is both a many-core programming methodology and a software programming library, thus it has requirements on both the design and implementation of general-purpose software programs. Such requirements not only lead to a clean and usable library, but also limit MAPLES' functionality. The sections below will discuss under what condition is the framework relevant, what's the principle behind its workings, what are the things that it can do, and what are some of its limitations.

3.1 Programming Model

Designs usually revolve around a problem to solve and a vision of the solution. The vision of the MAPLES framework was to make programming for multi-core platforms easy. Specifically, MAPLES was designed to pursue a vision of a single board capable of executing an entire space mission:

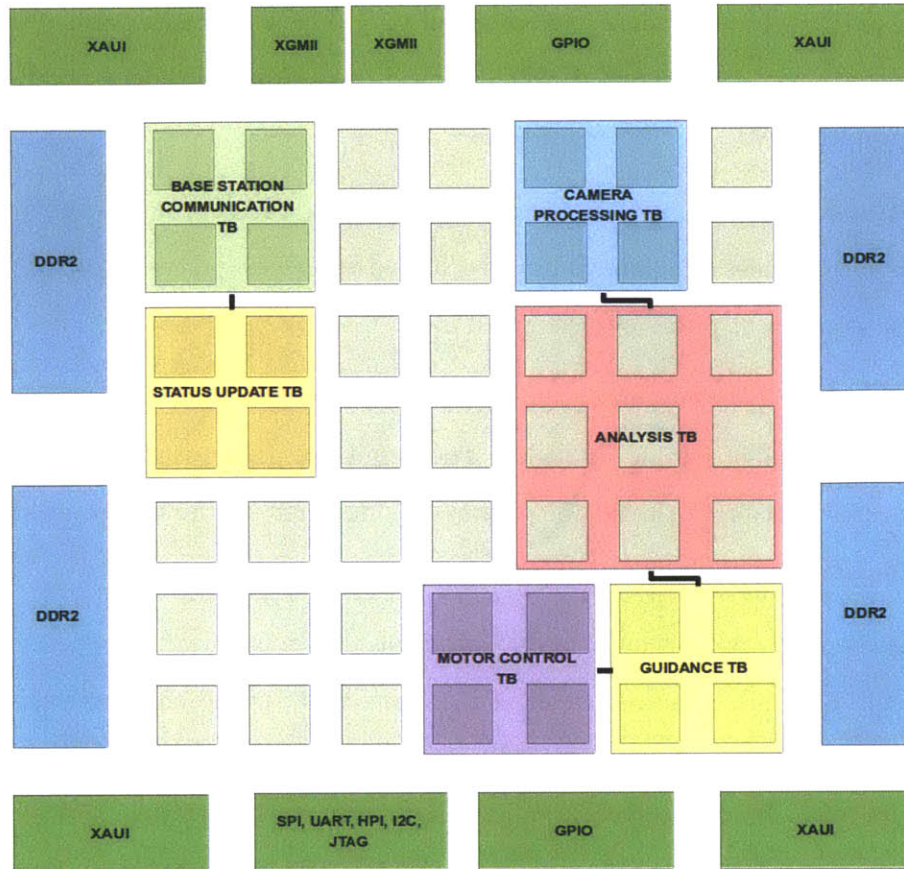


Figure 3-1: Multiple Applications Running on the Same Board

The diagram shows an MDB running different tasks at the same time. Each task block utilizes a block of CPUs, and communicates with other task blocks. An application can use any number of CPU cores, and can be composed of any number of parallelizable tasks. These tasks execute at the same time, and work together to provide the functionality of a complete program in a modular way.

3.1.1 Types of Targeted Problems

Ideally, programmers working on both embarrassingly parallel programs and non-embarrassingly parallel programs should be able to take advantage of the framework. To provide some background knowledge, an embarrassingly parallel problem is one for which little or no effort is required to separate the problem into a number of parallel tasks. This is often the case

where there exists no dependency (or communication) between those parallel tasks[8]. An embarrassingly parallel problem can be further broken down into data parallel and task parallel problems, with the difference being:

- Data parallel problems are solved by distributing the data across different parallel computing nodes. The parallelism is achieved when each processor performs the same task on different pieces of distributed data.
- Task parallel problems are solved by distributing execution processes (threads) across different parallel computing nodes. The parallelism is achieved when each processor executes a different thread (or process). Sometimes each processor executes on the same data, sometimes not.

Non-embarrassingly parallel problems are problems that do not exhibit obvious data or task parallelism. They are composed of sequential parts that depend on one another. In other words, the input of one sequential part depends on the output of another. This dependency creates a problem for the parallelization process and affects the resulting performance for the parallel version of the program. According to Amdahl's law, formulated by Gene Amdahl in the 1960s, a small portion of the program which cannot be parallelized will limit the overall speed-up available from parallelization[2]. The maximum possible speed-up is offered by the following formula:

$$S = \lim_{P \rightarrow \infty} \frac{1}{\frac{1-\alpha}{P} + \alpha} \quad (3.1)$$

with S being the maximum speed-up factor, α the fraction of running time the program spends on sequential (non-parallelizable) parts, and P being the number of running threads. Of course Amdahl's law doesn't take into account various details of the parallelization's implementation, but it provides a mathematical upper bound on the performance gain, and can serve as a good intuition during parallel programming.

3.1.2 Planned Solutions to the Problems

Embarrassingly parallel problems are usually solved in a straight-forward way. An illustration of the solution is as follows:

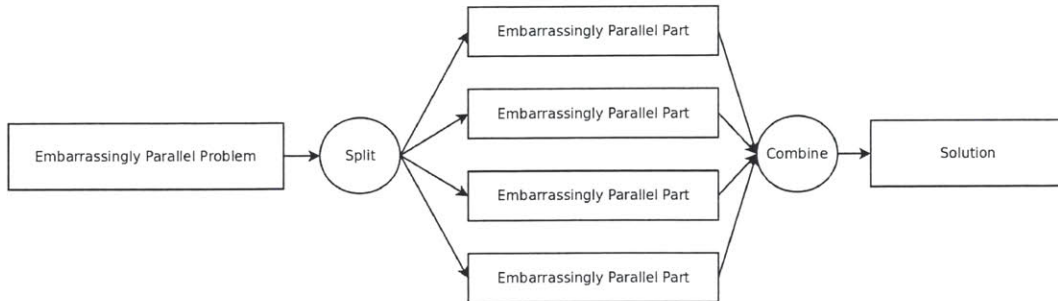


Figure 3-2: Illustration of How an Embarrassingly Parallel Problem is Solved

The embarrassingly parallel problem during the “Split” step is split up into replicated execution threads working on split data chunks. During the “Combine” step those replicated threads end, leaving the master thread with the solution. This illustration is reminiscent of the MapReduce model (shown below):

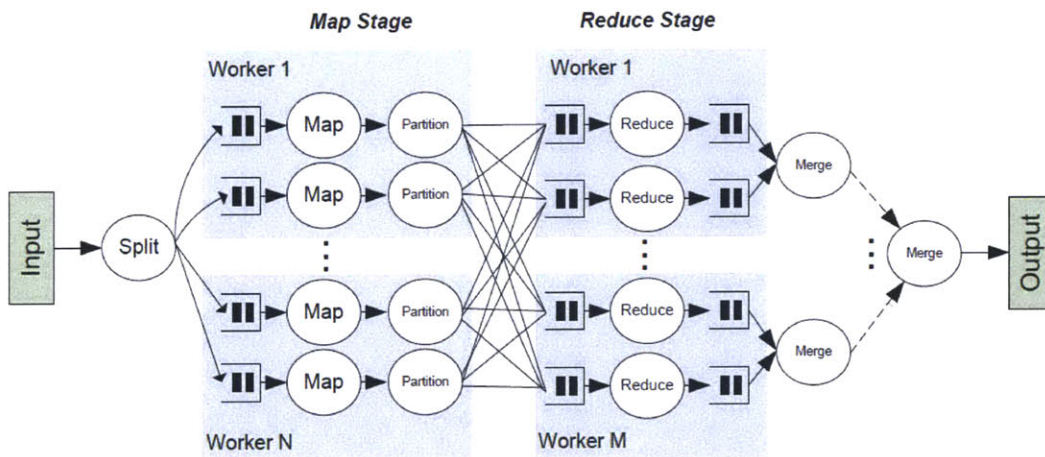


Figure 3-3: Illustration of How an Parallel Problem is Solved in a MapReduce Way [14]

Indeed the MapReduce model is an effort directed at solving embarrassingly parallel problems, with Apache Hadoop being its most successful implementation. In summary, the

execution of MapReduce is based on the “map” and “reduce” steps:

- “Map” step: The master node takes the input, divides it into smaller sub-problems, and distributes them to worker nodes. A worker node may do this again in turn, leading to a multi-level tree structure. The worker node processes the smaller problem, and passes the answer back to its master node.
- “Reduce” step: The master node then collects the answers to all the sub-problems and combines them in some way to form the output, the answer to the problem it was originally trying to solve.

Computations working on large datasets naturally follow this programming model, and thus the proposed solution should make it easy for programmers to write code for embarrassingly parallel problems, following this MapReduce programming model. In the solution, for the purpose of solving embarrassingly parallel problems, each parallelized task needs to have a clear abstraction of the master node in charge of mapping out inputs and collecting answers, and the worker nodes doing the heavy computation.

However, most real world problems are not embarrassing parallel. Oftentimes the solution requires a combination of data and task parallelism, along with some sequential parts. This poses many open-ended questions to the programmers: how to write the program differently to handle both the data and task parallelism; how to connect the sequential parts with the parallel parts; how the cores communicate within the same task; how they communicate in-between tasks; how many cores to use for each task; how much of a performance gain should be expected; and so forth. To answer these questions programmers need a guideline, essentially a cookbook, of how to parallelize a problem. The recipe needs to maintain a sequential workflow and contain multiple parallelized parts that can either run simultaneously or in a sequential fashion. The sequential operation can ideally be run in a pipelined fashion to increase throughput, and the number of cores can be easily adjustable for performance benchmark and optimization. The recipe, in MAPLES, centers around converting programs into pipelines of task blocks.

3.1.3 Task-Block Abstraction Design

Below summarizes the requirements MAPLES needs to have in order to facilitate parallel programming. Based on these requirements, MAPLES revolves around the concept of task blocks.

- Have a clear abstraction of master node in charge of mapping out inputs and collecting answers, and worker nodes doing the heavy computation
- Maintain a sequential workflow and contain multiple parallelized parts that can either run simultaneously or in a sequential fashion
- Ideally have the sequential operation run in a pipelined fashion to increase throughput
- Have easily adjustable number of cores allocation for tasks in order to help benchmark and optimize performance

Semantically, a task block is defined as a block of CPU cores that work together to achieve a common goal. Programmatically, a task block is defined as an instance of the class `TaskBlock`, and the fundamental abstraction within this framework. The class diagram in Figure 3-4 defines, at a high level, what MAPLES is.

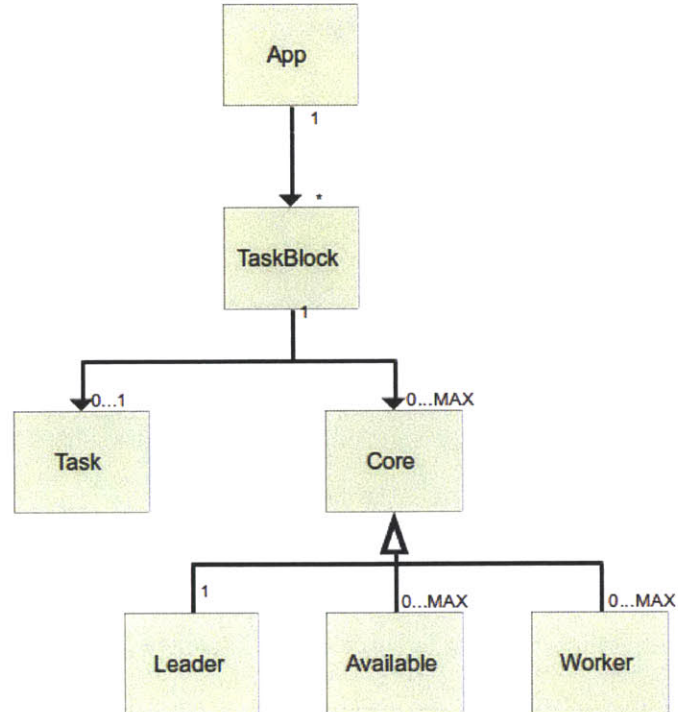


Figure 3-4: Class Diagram for MAPLES

An Application is composed of several TaskBlocks. Each TaskBlock runs one Task and parallelizes it to execute different data, on different cores, in a more-or-less MapReduce way. Each TaskBlock can contain as many cores as the underlying platform allows. Among the CPU cores, one becomes the **Leader** core, in charge of the “map” and “reduce” functions. The other cores become the **Worker** cores, in charge of computation and execution of the assigned function. Any unused core is initialized to be an **Available** core, ready to be integrated into any task block, as either a leader or a worker core.

A pseudo code of how an application can be defined and run using MAPLES’ programming library is as follows:

```

// For each task block,
task = new Task(function, dataArray)
taskBlock = new TaskBlock(tId, leaderCore, workerCoreArray)
taskBlock.task = task
  
```

```
taskBlocks.append(taskBlock)
...
// When it's time to start running the application,
application.taskBlocks = taskBlocks
application.run()
```

3.1.4 Communication Scheme

The communication protocol is abstracted out as an interface function ready to be populated. For data sharing inside each task block, according to what the programmer deems fit, either a shared memory model or a message passing scheme can be employed. For data sharing in-between task blocks, message passing is enforced as the only way of communication in order to generalize the MAPLES framework to the realm of distributed networking among multiple processor boards. This generality leads naturally to task blocks communicating across processor boards to engage in a more complex task network, for more complex space missions. Redundant task blocks running on different boards also provides the capability of board-level fault management.

However, since this thesis was developed on a single Maestro processor, communication among cores is a trivial software endeavor under this platform. Message passing between cores has already been implemented as function calls to utilize the UDN channel. Leader cores will be set up for passing UDN messages at the beginning of programs, sending and receiving UDN messages to and from other leader cores.

3.1.5 Program Run-Time Behavior

Using MAPLES requires an initialization process to set up the task block abstraction. The process is composed of four steps:

1. Booting and configuring each core
2. Setting up task blocks
3. Assigning tasks to all the task blocks

4. Kicking off the execution of the applications on board

The sequence of which the steps run can be summarized by the sequence diagram in Figure 3-5.

Boot

During the step of booting (represented by `boot` in the diagram), a start signal is sent to every desired core on board. An `AvailableCore` object is initialized for every core, waiting to be incorporated into task blocks. After each core is done initializing, it's ready to be added to any task block.

Setup Task Blocks

During the step of setting up task blocks (represented by `setupTaskBlock`), task blocks will be initialized according to the specification in the code. `AvailableCores` will be transformed to `LeaderCore` objects and `WorkerCore` objects, and added to the initialized `TaskBlocks`. In the context of Maestro architecture, UDN will also be initialized for the `LeaderCores`.

Assign Tasks

During the step of assigning tasks (represented by `assignTasks`), task functions are assigned to each task block, ready to be executed. Each `TaskBlock` object will be given a `Task` object, which includes a function definition and pointer and input data to be parallelized. The `LeaderCore` will pass the task function along with the divided data to each `WorkerCore`.

Run Application

During the step of kicking off application (represented by `runApp`), task blocks are given the go-ahead to start executing. For each task block, a `runTaskBlock` command will be executed, which starts the threads situated at every core. The program starts running.

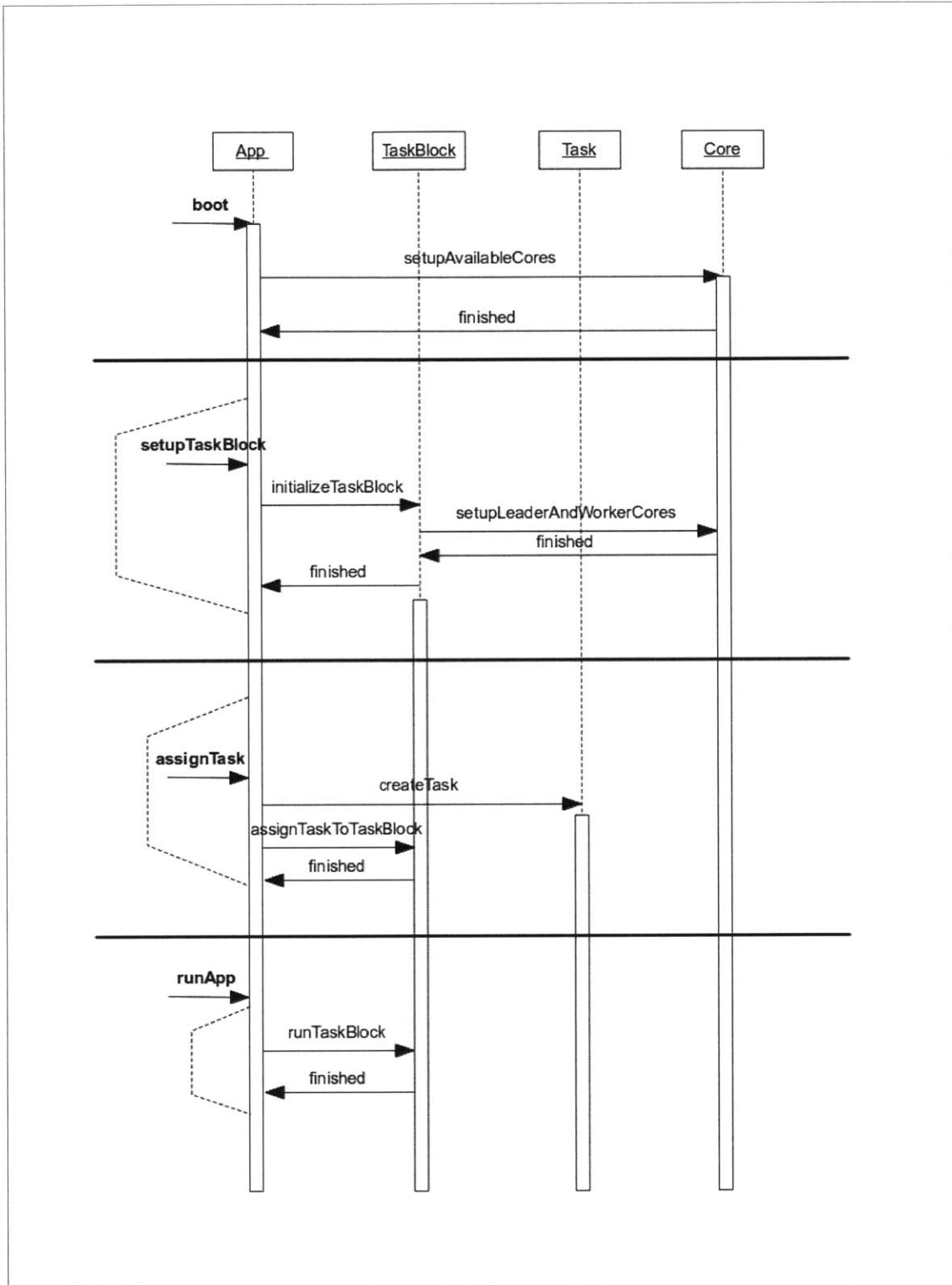


Figure 3-5: Sequence Diagram for MAPLES

Message listeners based at the UDN channel are running in every one of the `LeaderCores`, waiting for commands and alterations of tasks in real time. Task functions can be re-assigned, a core role can be changed, and cores can be taken in or out of task blocks. The operation of the program will keep executing until the point of termination defined in the code.

3.1.6 Space Computing Considerations

In the context of software engineering for space computing, two areas that demand great attention are:

1. Fault tolerant and real time operations
2. Effective power management

One key to achieving both is the ability to move programs and data around without interrupting the system [9], and MAPLES was designed with this idea in mind. Each level on the class diagram is mutable, in the sense that the role of each core, each task block, and hence each application can be altered in real time.

A core's role and a task block's task functions are assigned at compile time. However, both can be re-assigned during run time. This flexibility allows for the ability to not only move programs and data across different parts of the multi-core processor, but in extension also allows the replication of task blocks for fault tolerance purposes, and dynamic shutdowns of cores for power management purposes, a key requirement for long endurance space missions.

3.2 Intended Platforms

MAPLES's framework is designed for a general multi-core platform. A program expected to run on multi-core platforms should be able to benefit from being designed with MAPLES's methodology in mind. However, there are still certain limitations and assumptions made on the underlying platforms.

With the programming library, we try to make the platform as general as possible. We made all the abstractions necessary to ensure that the library is platform agnostic. All the platform dependent functions were put into wrappers and placed in a separate file. The library was implemented in *C++*. In terms of the operating system, the library is developed and compiled on SMP Linux.

However, the library does make assumptions about the underlying software and hardware platform. Although MAPLES has been designed with generality in mind, the library has only been compiled by *tile-cc* and tested on a MDB. Therefore, in order to use the library, a few Maestro like assumptions need to be true about the development platform:

- There is an underlying message passing scheme present among cores. In the Maestro architecture, the *iMesh* network provides application-level message passing capability through UDN and its Message Passing Interface (MPI)-style API written in *C*.
- There is support for the *gcc*-like compiler. Portable Operating System Interface (POSIX)-standard, GNU's Not Unix (GNU)-style Makefiles can be used for the build process. In the Maestro architecture, the compiler *tile-cc*, a derivative of the standard *gcc* compiler, is used.
- There is an underlying scheduling implementation to ensure that processes can meet their deadlines under real-time environments. In the Maestro architecture, SMP Linux runs on each core of the board managing task scheduling on each tile.
- There is a benchmark toolkit available to help with task and core allocation. In the Maestro architecture, the *tile-eclipse* IDE contains such a widget for monitoring and profiling program executions.
- There is a way to shut down cores to save power. Maestro CPU tiles can be put into sleep mode to save power.

In summary, the ideas in the framework of MAPLES has been thought out to be applicable to all types of many-core platforms, even though the implementation of the programming library contains certain restriction and assumptions for the underlying development plat-

form. The same ideas in MAPLES can very well be implemented in *C*, *Java*, *C#*, and of course, all variants of the *Lisp* dialects for their emphasis on functional programming.

Chapter 4

Test Case: TRN Crater

Recognition

In 2006, in response to NASA's Constellation program "Back to the Moon and Beyond", Draper Laboratory invested in Research and Development (R&D) aimed at improving lunar landing navigation accuracy for manned and unmanned (autonomous) landers [11] [16]. This vision guided, feature-based navigation algorithm integrated absolute navigation concepts including image feature extraction and matching to a database, with relative navigation including feature flow and tracking. The approach uses an Extended Kalman Filter (EKF) for vehicle navigation to integrate the two methodologies and thus capture the advantages of both techniques. This methodology was specifically developed for navigation in lunar orbiting or landing applications. Crater fiducial markers from the lunar surface are identified in an image and associated to a database. These markers are tracked across the image plane as the lunar lander moves in its orbit. Images used in this research to develop and analyze the algorithms were obtained from NASA's Clementine lunar mapping mission¹.

The Terrain Relative Navigation (TRN) algorithm was developed and tested through simu-

¹Clementine (officially called the Deep Space Program Science Experiment (DSPSE)) was a joint space project between the Ballistic Missile Defense Organization (BMDO) and NASA. Launched on January 25, 1994, the objective of the mission was to test sensors and spacecraft components under extended exposure to the space environment and to make scientific observations of the Moon and the near-Earth asteroid 1620 Geographos.

lation using Matlab. The crater recognition component of TRN was ported to C for testing on a RAD750TM platform. With the Maestro many-core architecture available, the lab has been actively looking for a way to parallelize the algorithm to take advantage of the new platform's computing power, and test the architecture's capability.

4.1 Algorithm Overview

The main components of the TRN algorithm are summarized in the following diagram [13]:

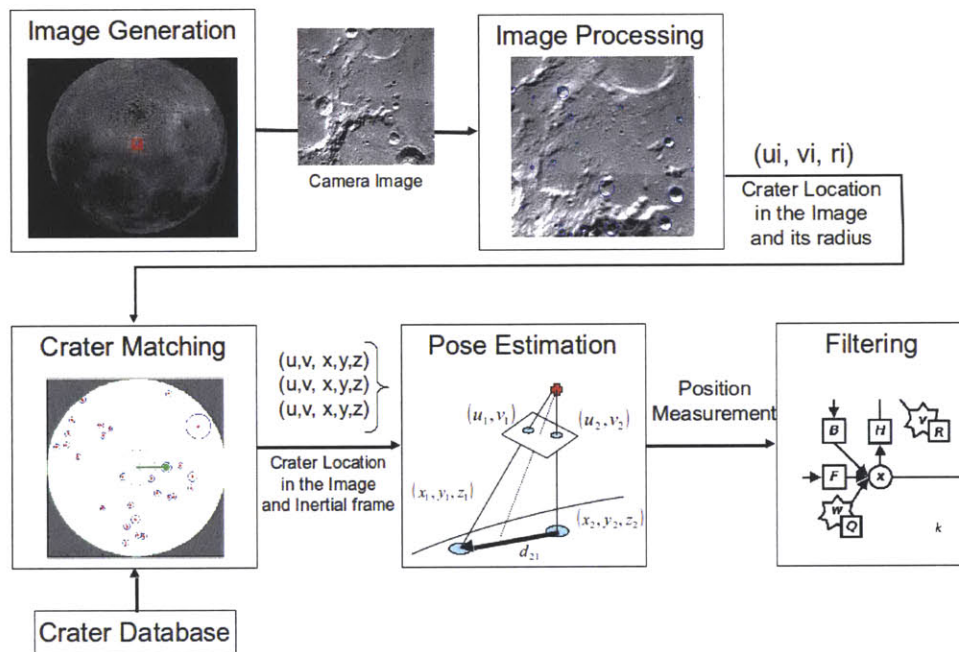


Figure 4-1: An Overview of the TRN Algorithm

During the image generation phase of TRN, a photo of the moon's surface is taken. The image is passed to the image processing phase, which detects the locations, sizes and shadow directions of the craters on the image. The information for all the craters is passed to the crater matching phase shown above, then combined with the inertial frame of the spacecraft, and compared to the crater database. The information is used to estimate the pose of the spacecraft, during the pose estimation phase. After the filtering phase, the pose is fed into other parts of the system where it is utilized to navigate the spacecraft. The algorithm then

goes back to the image generation phase to restart the process for the next image.

The core of the TRN algorithm lies in the image processing phase. Crater recognition, the real-time recognition of the locations, sizes and shadow directions of the craters, is a computationally intensive function. In order to take advantage of the many-core hardware architecture, along with the MAPLES software framework, we needed a basic understanding of the application. The TRN crater recognition algorithm is composed of the following parts:

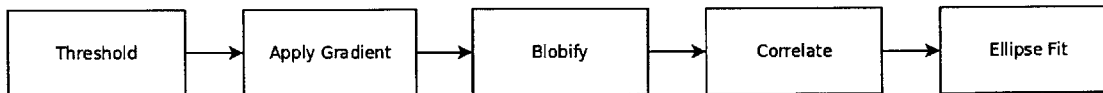


Figure 4-2: An Overview of the TRN Crater Recognition Algorithm

The algorithm is of a sequential nature, and contains five parts: threshold, apply gradient, blobify, correlate, and ellipse fit. A high-level description of what each part does is as follows:

1. During **Threshold**, a pre-determined threshold value is applied to the gray-scaled camera image. Pixels with brightness value above the upper threshold are separated into a high-threshold image, and pixels with values below the lower threshold are separated into a low-threshold image. Two image matrices are generated. Together with the original image matrices, the camera image is effectively transformed into a trinary (black, white, and gray) image.
2. During **Apply Gradient**, the angle of the Sun (pre-determined from the lander's inertial state) provides the basis for a directional gradient, which is 2D cross correlated with the image matrices to find the edges of possible craters in the image.
3. During **Blobify**, edges detected in the images are grouped together. Edges in the low-threshold image are grouped together to form dark edges of possible craters, and edges in the high-threshold image are grouped together to form bright edges of possible craters.
4. During **Correlate**, all possible pairs of dark and bright edges are correlated to see if together they form a circular crater.

5. During Ellipse Fit, for every one of the resulting circles in the Correlate phase, the (x,y) position and the radius r of the crater is computed.

The code for the crater recognition algorithm, written both under *C* and *Matlab*, serves as the basis for the parallelization.

4.2 Single-Core Benchmark

A benchmark of the single-core TRN crater recognition algorithm was done on RAD750 and the MDB to gain a performance comparison of the hardware architecture. The result is attached below:

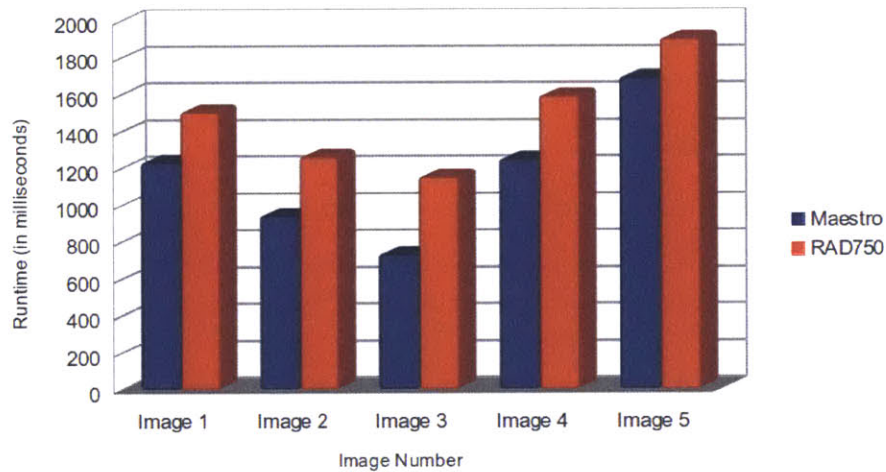


Figure 4-3: Benchmark Comparison of Unparallelized TRN Crater Recognition

The benchmark data is shown in the following table to allow mathematical analysis in the discussion:

(in milliseconds)	Image 1	Image 2	Image 3	Image 4	Image 5	Total
Maestro	1231	942	732	1247	1690	5842
RAD750	1505	1260	1150	1590	1900	7405
Runtime Ratio	0.818	0.748	0.637	0.784	0.889	0.789
Performance Ratio	1.223	1.337	1.571	1.275	1.124	1.268

Table 4.1: Unparallelized Benchmark Comparison Table

Through the benchmark, it can be seen that for the TRN crater recognition algorithm, each Maestro core performs roughly as 1.3 of a RAD750.

4.3 Pre-MAPLES Efforts

In this section the preliminary efforts to parallelize the TRN crater recognition algorithm are shown. These results were obtained prior to the development of MAPLES.

In the original *C* implementation, everything was written in a single master function. A natural extension and improvement over the written code was to separate each part of the algorithm into a different function. Code in the master function was split up into five sub-functions, each covering one part of the algorithm. Afterwards, a benchmark of the algorithm was done to see how much time has been spent on each part:

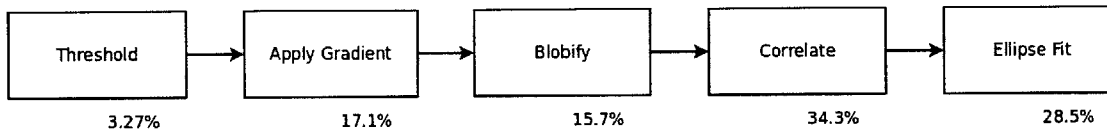


Figure 4-4: Execution Time Ratios of the TRN Crater Recognition Algorithm

The numbers show how much time each function spends relative to the entire algorithm. This benchmark information will help determine how many CPU cores are needed for each part of the algorithm.

The next step was the parallelization of the algorithm. A crude parallelization strategy was employed to emulate a baseline sequential-to-parallel porting processes. For each of the sub-functions, the data is split up and provided to a given number of cores to be computed. A global mutex lock and barrier are used to maintain thread consistency. At the beginning of each parallel function, data is split up into chunks, and given to different threads to be processed. Each thread is attached to a tile on the Maestro board, and all the threads execute concurrently. At the end of each thread's execution of the sub-function, the thread stops at the global barrier and waits for all the other threads to finish. After all

the threads join at the barrier, the algorithm proceeds to the next sub-function. Figure 4-5 below illustrate the parallelization scheme. Since Threshold only takes 3.27% of the

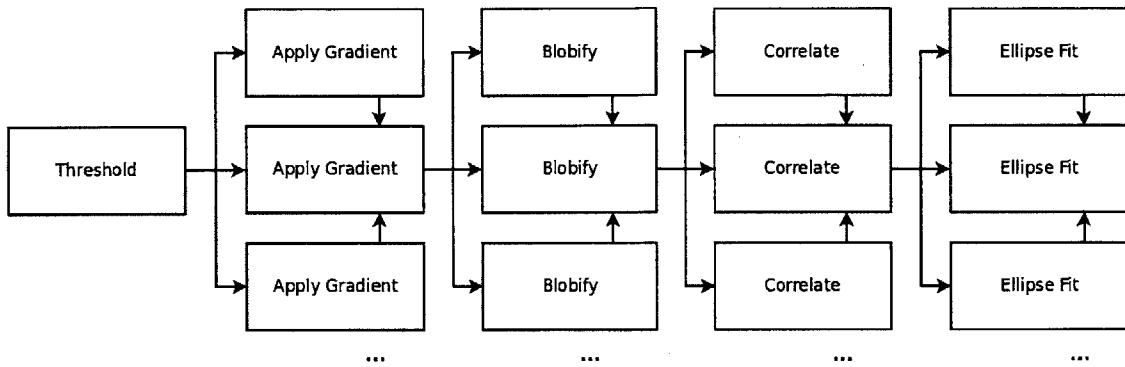


Figure 4-5: Crude Crater Recognition Parallelization, Design 1

runtime, it was left out of the parallelization process, as its parallelization was insignificant and actually increased the runtime because of the additional "splitting" and "combining" steps (see Figure 3-2). Each of the other sub-functions were spread across several cores, and the timing performance was benchmarked. Benchmark data showed that for many of the sub-functions, the cost of communication between cores can outweigh the benefit of parallelization as the number of utilized cores increases. Hence, after some redesign, the following parallelization scheme was employed:

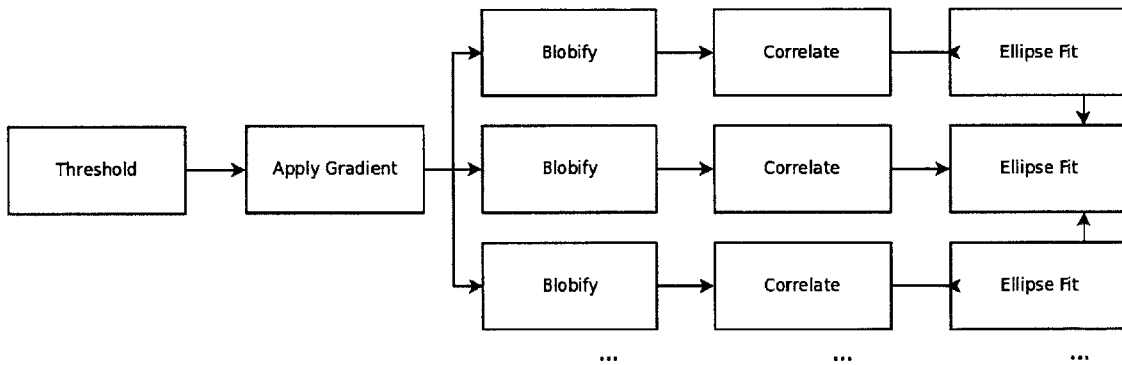


Figure 4-6: Crude Crater Recognition Parallelization, Attempt 2

Parallelization of the first two parts of the algorithm (Threshold and Apply Gradient) was eliminated, because Apply Gradient's code was structured in a sequential way that was unfit for easy parallelization. Blobify, Correlate and Ellipse Fit were combined

together, to save the splitting and combining costs between the three parts.

Expressed using an MDB core diagram on MDB, the program executed at each step like the following:

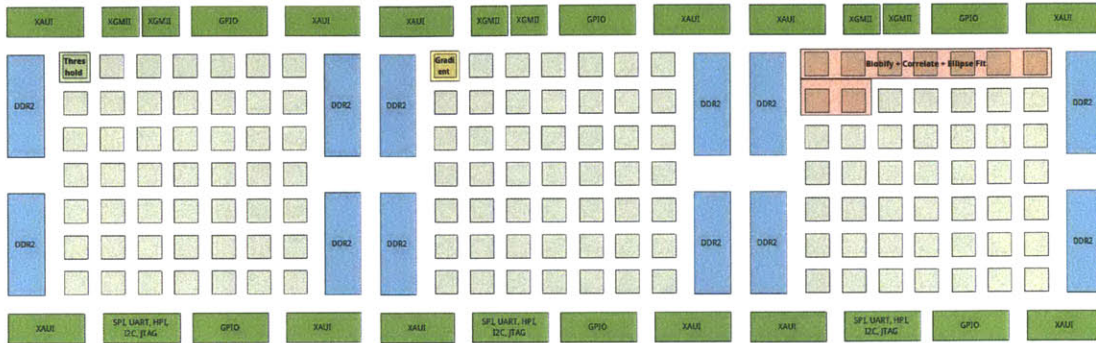


Figure 4-7: Crude Crater Recognition Parallelization on MDB Diagram

Next, the optimal number of CPU cores to use for the parallelization was determined. Numbers from 1, 2, 4, 9, 16, 25, 36, and 49 were used, and the parallelized algorithm was run for each case over five sample moon surface images taken from the Clementine’s High-Resolution Camera (HIRES) image database. The five images are all of size 512×512 pixels, translated from Portable Network Graphics (PNG) format to 2-D arrays of gray-scale, with values from 0 to 255.

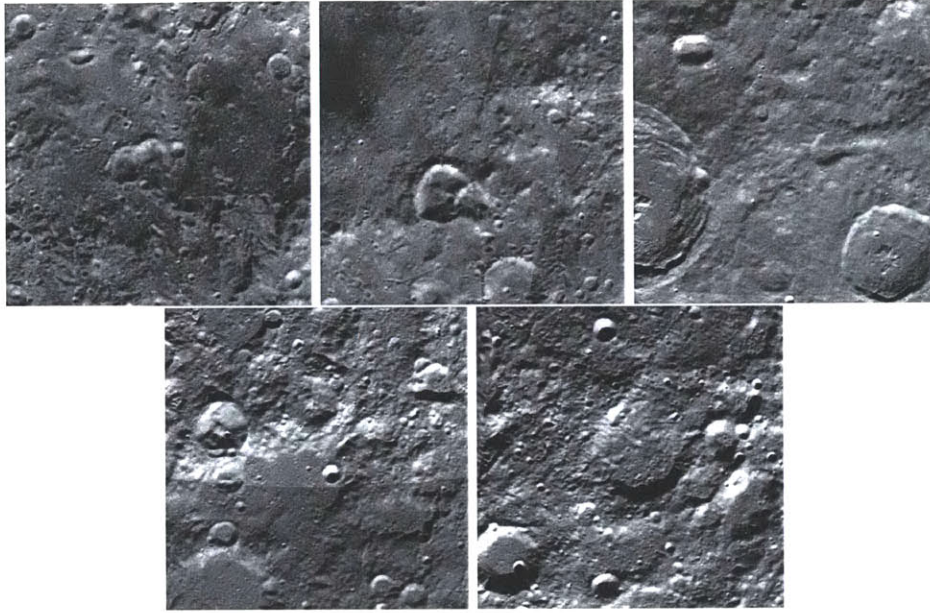


Figure 4-8: Sample Lunar Images used for Benchmarking

The result of the benchmark is summarized in Figure 4-9.

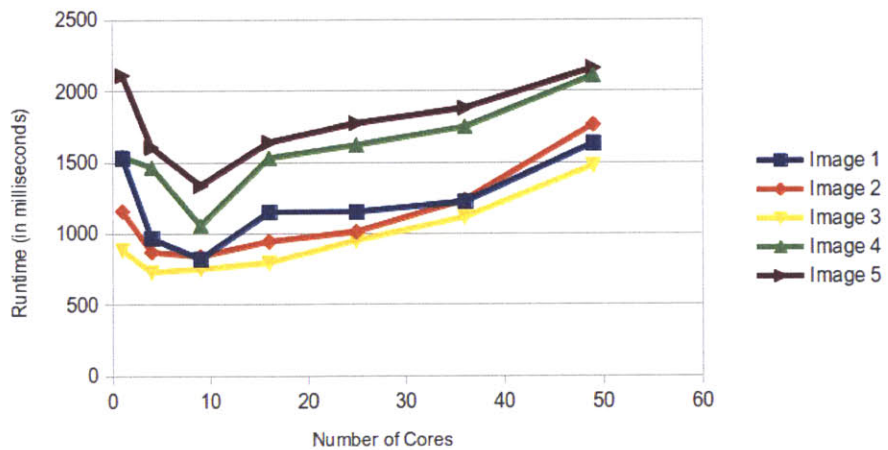


Figure 4-9: Benchmark Results with Varied Number of Cores

The runtime is calculated by dividing the number of cycles by the frequency of Maestro's CPUs (260 MHz). Initially, the runtime dips as the number of cores used increases. However, the runtime starts to increase after a certain threshold is passed (at nine cores). As the number of threads increases, there is more associated cost for core-to-core communication

and the creation of localized objects within each thread. If the computation gain is less than the cost of expanding threads, the performance diminishes. According to the benchmark results, the optimal number of cores used for the parallelization was determined to be nine. Benchmarks were also done for the five sample images on RAD750TM, and the results were compared to the Maestro result. The graph showing the comparison is attached in Figure 4-10:

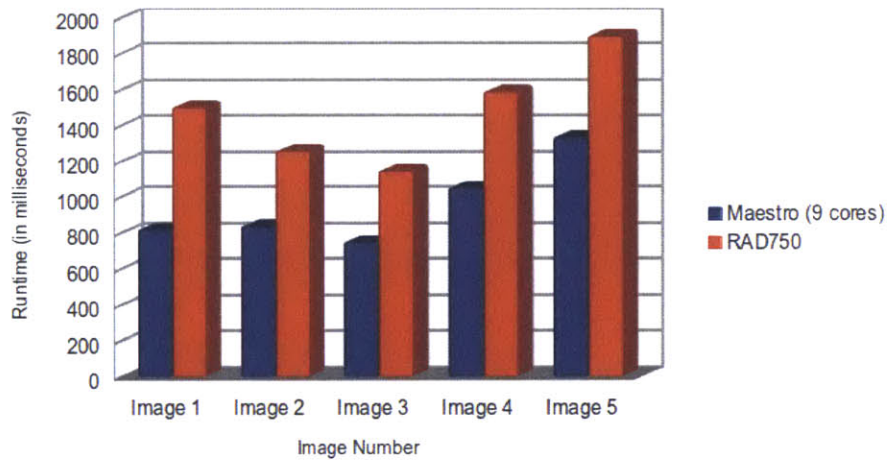


Figure 4-10: Benchmark Comparison between Crude Parallel Run on Multi-Core Maestro and Sequential Run on RAD750TM

On average, over the five images, the optimal configuration of nine-core parallelization decreases the runtime by 30% to 40%, which translates to a performance gain of about 40% to 60%.

4.4 MAPLES Implementation

With MAPLES, the code was reconstructed to make use of task blocks. Each component of the TRN algorithm was declared to be a task block, and the code within each sub-function was re-written to exhibit its functional nature, to be made stateless, and to be made independent from other functions in the algorithm. The allocation of the task blocks

over the cores on the board is shown below:

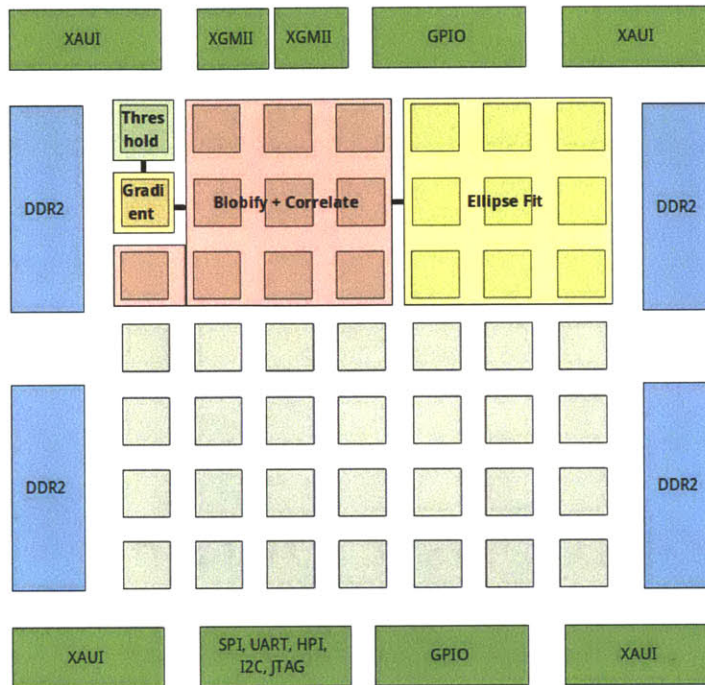


Figure 4-11: Core Diagram of Task Block Allocation for Crater Recognition

A Boolean value controls whether the task blocks are executed in a pipelined fashion or not. If pipelining is enabled, once the `Threshold` task block finishes its work with the first image, it immediately starts to work on the second image without waiting for the latter task blocks to finish. With pipelining enabled, greater performance gain is achieved. However, this makes predicting the runtime deadline for executing one image harder, since images are bundled together in the pipeline to be processed.

Most of the master mutex locks were removed from the code, and message listeners were installed within the `LeaderCore` of each task block to regulate thread consistency and the program flow. The sequential nature of the algorithm was preserved in the pipeline, and the parallel nature of each sub-function was utilized inside each task block.

The output from one block is passed to the next task block as the input, prior to its

execution. When there are no more images available for processing, the first task block in the pipeline (**Threshold**) sends a kill signal to the next task block (**Apply Gradient**), which puts the task of ending the threads onto the task queue, sends out a kill signal to the next task block, waits until the current operation on all worker cores ends, and ends the task block's operation.

Each task block now executes a part of the algorithm that has an explicit input and output:

1. For **Threshold**, the input includes the gray-scaled camera image. The output includes the two generated image matrices. The binary matrices are passed on to **Apply Gradient**. If pipelining is enabled, the threshold task block immediately starts executing the next image.
2. For **Apply Gradient**, the input includes the two generated image matrices, and the angle of the Sun. The output includes the edges of possible craters in the image, passed on to **Blobify**.
3. For **Blobify**, the input includes edges detected in the images. The output includes the dark and bright edges of possible craters, passed on to **Correlate**.
4. For **Correlate**, the input includes the dark and bright edges. The output includes the matched edges of craters.
5. For **Ellipse Fit**, the input includes the matched edges and resulting craters. The output includes the positions and radii of the detected craters.

The transformed algorithm is free of locks, but relies on the underlying iMesh network to transfer large amount of data. The success of this parallelization now depends on the speed of the communication network.

The resulting code is a mix of *C++* and *C*, since the library was written in *C++*, and the original algorithm was written in *C*. The compiler *tile-cc* compiled the code without warnings.

4.5 Parallelization Result

The pipelined parallelized algorithm and unpipelined algorithm were compiled into two executables, and transferred to the MDB. The MDB was reset between each run to ensure empty cache in each core before each algorithm is run. Benchmarks were obtained twice for each setting, and averaged for analysis.

For the unpipelined runs, the result is summarized below by the X-Y graph:

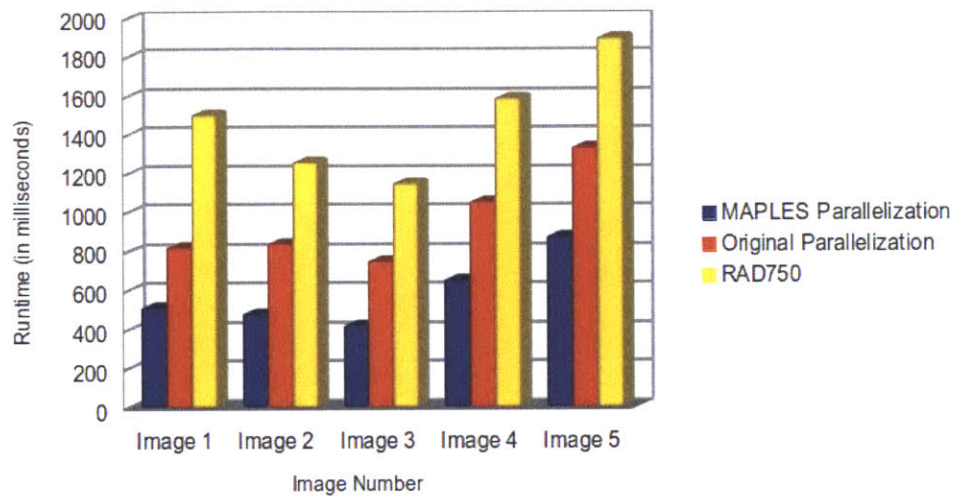


Figure 4-12: Benchmark Comparison of the Un-Pipelined MAPLES Performance

For the pipelined runs, because images were bundled together, the result is better summarized in Figure 4-13:

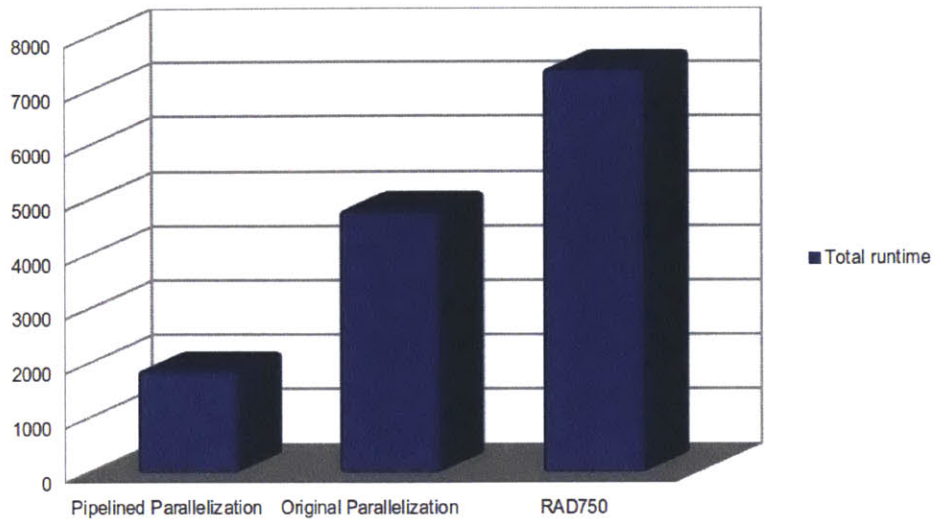


Figure 4-13: Benchmark Comparison of the Pipelined MAPLES Performance

All the benchmark data is shown in the following table to allow mathematical analysis in the discussion:

<i>(in milliseconds)</i>	Pipelined	Un-Pipelined	Original	RAD750
Image 1		514	822	1505
Image 2		482	840	1260
Image 3		422	751	1150
Image 4		655	1055	1590
Image 5		877	1337	1900
Total Runtime	1877	2950	4805	7405
Runtime Ratio	1.000	0.636	0.391	0.253
Performance Ratio	1.000	1.572	2.560	3.945

Table 4.2: Benchmark Comparison Table

The *Runtime Ratio* is the runtime of the pipelined version divided by the respective parallelization runtimes, and the *Performance Ratio* is $1/\text{Runtime Ratio}$. The pipelined algorithm running on Maestro performed better than the original RAD750TM by $3.95 - 1 = 2.95 = 295\%$.

4.6 Discussion

Though Table 4.1 shows that each Maestro core is about 1.3 of a RAD750, the TRN crater recognition parallelization to 21 cores did not yield a linear performance gain in the order of $1.3 \times 21 = 27.3 = 2730\%$. Aside from the imperfect parallelization design and the cost that comes with parallelization, the sequential and non-embarrassingly parallel nature of the algorithm was also part of the problem.

Amdahl's law provides an upper limit on the performance gain of parallelization. A sequential run of the TRN crater recognition algorithm was benchmarked, showing the time ratios between its functional blocks (see Figure 4-4). Using the time ratio and the Amdahl's law in Equation 3.1, we can infer the maximum performance gain. In the final version, Blobify, Correlate, and Ellipse Fit are parallelized. Therefore, the sequential part of the algorithm contains Threshold and Apply Gradient, which constitutes $\alpha_{CR} = 3.27\% + 17.1\% = 20.37\%$ of the total runtime. Plugging α_{CR} into the formula yields:

$$\begin{aligned} S &= \lim_{P \rightarrow \infty} \frac{1}{\frac{1-\alpha_{CR}}{P} + \alpha_{CR}} \\ &= \lim_{P \rightarrow \infty} \frac{1}{\frac{1-20.37\%}{P} + 20.37\%} \\ &\approx 4.91 \end{aligned}$$

Since S is defined as the speed-up factor, the corresponding time ratio is $1/4.91 = 0.204$. The time ratios observed during experiments are 0.253 for the pipelined version, and $2950/7405 = 0.398$ for the unpipelined version. This difference between the Amdahl's prediction and actual parallelization benchmark shows that the parallelization performed in the thesis is not yet optimal. The reason for this difference lies most likely in the number of cores mapped to each task block. The decision on exactly how many cores to map to each of the task blocks were made in an arbitrary way, and did not go through an extensive testing of different combinations.

When running this algorithm, a *Performance/Power* comparison between Maestro and

RAD750TM should also be considered. For Maestro, the average power output is 20 W, and shutting down each core saves 270 mW. For RAD750TM, the average power output is 5 W (see Chapter 2). The parallelization for the crater recognition algorithm only used 21 cores on the MDB. Therefore, the rest of 28 cores can be shut down to lower the power usage of the board to $20 \text{ W} - 270 \text{ mW} \times 28 = 12.4 \text{ W}$. By taking into account of the performance ratios, we see that for the pipelined parallelization, the *Performance/Power* (written as P) ratio is:

$$\begin{aligned} P_{\text{maestro}} : P_{\text{rad750}} &= \frac{3.95}{12.4 \text{ W}} : \frac{1}{5 \text{ W}} \\ &= 1.56 : 1 \end{aligned}$$

For the parallelization of the TRN crater recognition algorithm, a processor switch from RAD750TM to Maestro is thus justified. Even though the performance is not as much as we expected from the difference in the numbers of CPU cores between the processors, we do acknowledge that the speed-up of the algorithm through parallelization is effective.

Chapter 5

Conclusion

This chapter summarizes the contributions of this thesis, and suggests future work for improvements.

5.1 Contributions

Through the design, implementation and testing of MAPLES, this thesis has made three significant contributions.

This thesis proposed a software framework for general purpose many-core programming. It provided both a model to guide application parallelization, and a software programming library to use for application implementation. The framework's effectiveness was proven through a parallelization of a sample application, in this case, a TRN algorithm for crater recognition.

Through the work done in this thesis, the Draper-developed crater recognition algorithm was parallelized and benchmarked (See Table 4.2). Master mutex locks were removed, and functions were placed in task blocks using the MAPLES library. Compared to the baseline performance of the algorithm on RAD750TM, the MAPLES parallelization running on Maestro performed faster by 151% without pipelining. If pipelining was allowed, the performance gain processing the five sample images, over the baseline performance on RAD750TM,

was 295%. The parallelization used 21 Maestro cores.

This thesis also included one of the first applications written for the Maestro processor. Benchmarks were done to evaluate the capability of Maestro, and the process of programming and running software on the Maestro architecture was documented. Although a significant speed-up on the order of the number of cores was not observed in the comparison between the test application running on Maestro and on RAD750TM, this thesis justified Maestro's usage through a *Performance/Power* comparison with RAD750TM, and suggested a direction for future development of many-core applications.

5.2 Future Work

A number of directions can be pursued in terms of expanding the work done in this thesis. Some are related to the MAPLES framework, others related to the TRN algorithm and the Maestro architecture.

Although the feature of dynamic task allocation was planned for MAPLES, it has not been implemented and tested due to time constraints. Being able to dynamically change cores' tasking at runtime is essential for effective fault management, and should be pursued in the future.

The lack of automatic configuration for optimal number of cores used in each task blocks is dissatisfactory as well. To enable more effective parallelization, automatic profiling, benchmarks and dynamic core-count configuration should be added to the features of this framework.

The crater recognition and TRN algorithms tested so far work on sample image files. Maestro provides high-speed I/O ports for connection with high data rate sensors such as cameras. Real-time image input (or simulated real-time image input) can be used to test (1) the vision-based navigation Algorithm, and (2) the Maestro board's I/O communication capabilities.

Also, parallelization has currently only been done to the crater recognition part of the TRN.

Other parts of TRN can be ported over to Maestro and run on the same board. because only then can the vision of “single board, entire mission” seen in Figure 3-1 be realized.

Acronyms

API	Application Programmer Interface
ASIC	Application Specific Integrated Circuit
BAE	British Aerospace Engineering
CMOS	Complementary Metal-Oxide Semiconductor
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DDR	Double Data Rate
DMA	Direct Memory Access
DSP	Digital Signal Processing
EKF	Extended Kalman Filter
FPU	Floating Point Unit
GEO	Geosynchronous Orbit
GFLOPS	Giga Floating-point Operations Per Second
GNU	GNU's Not Unix
GOPS	Giga Operations Per Second
GPS	Global Positioning System
GPU	Graphics Processing Unit
HEOMD	Human Exploration and Operation Missions Directorate
HIRES	High-Resolution Camera
I/O	Input/Output
IDE	Integrated Development Environment
IDN	I/O Dynamic Network
ISI	Information Sciences Institute
kB	Kilo-Bytes
KPP	Key Performance Parameters
LEO	Low Earth Orbit

MAPLES	Many-core Programming Library for Embedded Space Computing
MDB	Maestro Development Board
MDE	Maestro Development Environment
MDN	Memory Dynamic Network
MFLOPS	Mega Floating-point Operations Per Second
MHz	Mega Hertz
MIPS	Millions of Instructions Per Second
MPI	Message Passing Interface
NASA	National Aeronautics and Space Administration
NRO	National Reconnaissance Office
OCT	Office of the Chief Technologist
OPERA	On-board Processing Expandable Reconfigurable Architecture
OS	Operating System
PCI	Peripheral Control Interconnect
PID	Process Identification
PNG	Portable Network Graphics
POSIX	Portable Operating System Interface
R&D	Research and Development
RISC	Reduced Instruction Set Computing
ROM	Read Only Memory
SAD	Sum of Absolute Values
SIMD	Single Instruction Multiple Data
SMD	Space Missions Directorate
SMP	Symmetric Multi Processor
STN	Static Network
TBB	Threading Building Blocks
TDN	Tile Dynamic Network
TLB	Translation Lookaside Buffer
TRN	Terrain Relative Navigation
UDN	User Dynamic Network
UI	User Interface
VBN	Vision-based Navigation
VLIW	Very Long Instruction Word
W	Watts

References

- [1] *OPERA IP Release Day: Program Overview*. OPERA Intellectual Property, December 2008.
- [2] G. S. Almasi and A. Gottlieb. *Highly parallel computing*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1989.
- [3] R.W. Berger, D. Bayles, R. Brown, S. Doyle, A. Kazemzadeh, K. Knowles, D. Moser, J. Rodgers, B. Saari, D. Stanley, and B. Grant. The rad750tm-a radiation hardened powerpctm processor for high performance spaceborne applications. In *Aerospace Conference, 2001, IEEE Proceedings.*, volume 5, pages 2263 –2272 vol.5, 2001.
- [4] The Boeing Company. *Radiation-Hardened Onboard Processing Expandable Reconfigurable Architecture Design and Development Study*, July 2008. Maestro Chip Requirements: CDRL A020.
- [5] Linux Devices. Massively multicore processor runs linux. *LinuxDevices.com*, August 2007. <http://www.linuxfordevices.com/c/a/News/Massively-multicore-processor-runs-Linux/>.
- [6] Mike Shafto et al. Modeling, simulation, information, technology and processing roadmap: Technology area 11. In *NASA’s Space Technology Roadmaps*. National Aeronautics and Space Administration, April 2012.
- [7] Doyle et.al. High performance spaceflight computing (hpsc): an avionics formulation task study report. Technical report, NASA Office of the Chief Technologist, 2012.
- [8] Ian Foster. *Designing and Building Parallel Programs*, chapter Section 1.4.4. AddisonWesley, 1995. ISBN 9780201575941.
- [9] K.P. Gostelow. The design of a fault-tolerant, real-time, multi-core computer system. In *Aerospace Conference, 2011 IEEE*, pages 1–8, 2011.
- [10] N.F. Haddad, R.D. Brown, R. Ferguson, A.T. Kelly, R.K. Lawrence, D.M. Pirkl, and J.C. Rodgers. Second generation (200mhz) rad750 microprocessor radiation evaluation. In *Radiation and Its Effects on Components and Systems (RADECS), 2011 12th European Conference on*, pages 877 –880, sept. 2011.
- [11] Matthew J. Hale. Terrain relative localization for lunar entry, descent, and landing. Master’s thesis, Massachusetts Institute of Technology, June 2007.
- [12] ISI East. *OPERA Maestro Wiki*. <https://opera.isi.edu>.

-
- [13] B. Orecchio, E. Sun, and T Tzanetos. Space maestro: Student final presentations. Internal use. Unpublished.
 - [14] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 13–24, 2007.
 - [15] K. Singh, J.P. Walters, J. Hestness, Jinwoo Suh, C.M. Rogers, and S.P. Crago. Fftw and complex ambiguity function performance on the maestro processor. In *Aerospace Conference, 2011 IEEE*, pages 1–8, 2011.
 - [16] Leena Singh and Sungyung Lim. On lunar on-orbit vision-based navigation: Terrain mapping, feature tracking driven ekf. In *Guidance, Navigation and Control Conference and Exhibit*. American Institute of Aeronautics and Astronautics, Honolulu, HI, 2008.
 - [17] Jinwoo Suh, K.J. Mighell, Dong-In Kang, and S.P. Crago. Implementation of fft and crblaster on the maestro processor. In *Aerospace Conference, 2012 IEEE*, pages 1–6, March.
 - [18] UBM TechWeb and Dr. Dobbs. The parallel programming landscape. *The State of Parallel Programming*, 2012.
 - [19] Tiler Corporation. *Multicore Development Environment: Programming the Tile Processor*, April 2009. UG205.
 - [20] Tiler Corporation. *Tiler Processor Architecture Overview*, November 2009. UG100.