Mapping Loop-Based Programs onto a Multithreaded Processor

H
ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης
του Τμήματος Πληροφορικής
Εξεταστική Επιτροπή

από τον

Σαούγκο Δημήτριο

ως μέρος των Υποχρεώσεων

για τη λήψη

του

ΔΙΔΑΚΤΟΡΙΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

Οκτώβριος 2014

DEDICATION

This work is dedicated to Catherine. She fought until the very end and will always be in our hearts and minds.

ACKNOWLEDGEMENTS

This thesis contains the results of a research that started in 2008 but culminated in 2011 to 2012. Its progress reflected my personal life which was marked by both the brightest and the darkest of times. For this reason alone I would like to thank my Supervisor, George Manis, who not only guided me (and insisted on some choices even when I vehemently insisted that "that can't be done!") but also demonstrated Jobian levels of patience when dealing with me. I would also like to thank my consulting committee members Chris Jesshoppe and Nikolaos Papaspyrou for helping me improve on my work and my thesis. Of course special mention goes to my family, my parents George and Anthoula as well as my siblings, Vasilis, Catherine and Joan for all the support they have given me over the years. Finally I would like to thank my close friends George(x3), Mary, Helen, Socrates and Lee for being there for me when I needed them the most. This thesis would not exist without any of these people.

TABLE OF CONTENTS

TABLE INDEX

FIGURE INDEX

ABSTRACT

Saougkos Dimitrios. PhD Candidate, Computer Science Department, University of Ioannina, Greece. Graduation Month, Graduation Year. "Mapping Loop-Based Programs onto a Multithreaded Processor". Thesis Supervisor: Manis George.

This thesis offers some insight into the automatic parallelization of loops by introducing and describing a source-to-source parallelizing compiler developed from scratch called C2μTC/SL. Once basic notions and ideas on the field of automatic parallelization have been introduced, the SVP system is described in great detail. It is a novel proposal on multi-core architectures and is what C2μTC/SL targets as output. The SVP is a novel design for a multi-threaded processor that can be bundled together with an OS-on-chip as part of the chip's ISA (Instruction Set Architecture). Several of those SVP cores together form a microgrid. The programming paradigm followed by the microgrid is that of a family of threads. Each family executes independently and all the threads belonging in such a family run in parallel. A thread can create more ad-hoc families so a whole hierarchy of families can exist at any given time. Synchronization is achieved by a series of synchronizing channels that can carry information from one thread in the family to its neighbors. The whole system can revert back to complete sequential execution once all resources are taken. Two programming languages were created for the high level programming / abstraction layer of the SVP: μTC and SL. Both are explained later in the text however they both are extensions of the basic C language. They extend the language with a series of directives for the creation and execution of families of threads.

The C2μTC/SL source-to-source compiler is described afterwards: its purpose is to take as input any legacy C code and transform it into a parallel SL program. Originally its output was the μTC language but with the advent of SL it changed to that, hence the name C to μTC / SL (C2μTC/SL). The compiler's main target constructs are loops since a loop is where most of the execution time of an application takes place. Since SVP works with families of threads that resemble single-dimensional loops, transforming any kind of loop into a meaningful construct for the SVP is an important step. For that reason, loops are divided into single-dimensional and multi-dimensional ones with each category requiring a different transformation method.

Single-dimentional loops are further categorized by the number of the so-called loop carried dependencies that they have and are treated accordingly. Loops with no dependencies are just translated simply into parallel families. Loops with dependencies utilize the SVP's synchronizing channels to transfer data from one thread to the next in a dataflow manner. This action alleviates the weight of each thread having to access the global memory for a particular piece of data since

whatever it needs is simply transferred over via the synchronizing / shared channel. Once each thread finishes computation it pushes all relevant data back to the shared channel for use by the next thread. The combination of parallel executing independent data-flows (data-chains) and the synchronizing channel to reduce accesses to the main global memory brings tremendous increases in speedup and efficiency.

Multi-dimensional loops are also subcategorized into two groups. The first group is the one that contains no dependencies. Again each loop of the loop nesting is simply transformed to a fully parallel family and it is up to the SVP to run the code effectively. The second and most interesting group contains the perfect loop nestings with a static dependency vector. Lamport's hyperplane idea is applied in this case however there is a novelty: Instead of precomputing any loop transformation, it is up to the run-time environment to intuitively follow the dependency vector over the index space and discover the different hyperplanes per cycle. This novel idea gave birth to our first run-time algorithm: The fixed-size algorithm. It has the ability to apply the hyperplane idea, discovered while running the actual computation code, into the various tiles of a fixed size which divide the innermost dimension of the loop. The fixed size algorithm proved to work properly, however for optimal or even good results the size of the tile was needed to be known beforehand, effectively making the whole algorithm not particularly useful except as a stepping stone and also a great tool for comparisons.

This glaring weakness of the Fixed-Size algorithm was covered by its evolutionary "descendant": the Self-Adaptive algorithm. Working on the same principles as the Fixed-Size one, it can, at run-time, determine the optimal tile size to use at any given computation cycle by reducing it or increasing it according to the current needs.

Experimental results indicate that not only the Self-Adaptive algorithm fares very well with near-optimal results when compared with the Fixed-Size one, it is also shown that for that particular type of parallelism (run-time execution of parallel families discovered on the spot) the results obtained are the best possible results that can be obtained. The algorithms were also compared with a standard compile-time method (the hyperplane method) and it was found that their speedup is relatively close to each other. This combined with the versatility offered by a run-time system (like dealing with irregular index spaces) makes the Self-Adaptive algorithm especially appealing.

ΕΚΤΕΝΗΣ ΠΕΡΙΛΗΨΗ ΣΤΑ ΕΛΛΗΝΙΚΑ

Σαούγκος Δημήτριος. Υποψήφιος Διδάκτωρ, Τμήμα Πληροφορικής, Σχολή Θετικών Επιστημών, Πανεπιστήμιο Ιωαννίνων. Μήνας / Έτος. «Απεικόνιση Βρόχων σε Πολυνηματικό Επεξεργαστή». Επιβλέπων: Μανής Γεώργιος.

Η παρούσα διατριβή προσφέρει μία περιήγηση στον κόσμο της αυτόματης παραλληλοποίησης των βρόχων παρουσιάζοντας και περιγράφοντας παράλληλα ένα εργαλείο αυτόματου παραλληλισμού (πηγαίο σε πηγαίο) που δημιουργήθηκε εκ του μηδενός και ονομάζεται C2μTC/SL. Αφού παρουσιαστούν βασικές ένοιες στον χώρο του αυτόματου παραλληλισμού, το σύστημα SVP περιγράφεται: Μια καινοτόμος πρόταση στις πολύ-πύρενες αρχιτεκτονικές και αποτελεί στόχο - έξοδο του C2μTC/SL. Το SVP αποτελεί το σχέδιο για έναν πολύ-πύρηνο επεξεργαστή και έχει την ιδιότητα να εκτελεί ένα ολόκληρο λειτουργικό σύστημα το οποίο μπορεί να καταλαμβάνει μέρος του ISA (Instruction Set Architecture) του πυρήνα. Πολλοί από αυτούς τους πυρήνες μπορούν να συνδυαστούν στο λεγόμενο μικροπλέγμα (microgrid). Ο προγραμματισμός του microgrid στηρίζεται σε οικογένειες από νήματα. Κάθε οικογένεια εκτελείται αυτόνομα και όλα τα νήματα που ανήκουν σε αυτήν την οικογένεια μπορούν να εκτελεστούν παράλληλα. Επίσης, κάθε νήμα μπορεί να δημιουργήσει όσες οικογένειες χρειάζεται κατά βούληση. Με αυτόν τον τρόπο, μια ολόκληρη ιεραρχία από νήματα μπορεί να εκτελείται ανά πάσα στιγμή στο microgrid. Ο συγχρονισμός μεταξύ των νημάτων επιτυγχάνεται από την ύπαρξη μιας σειράς καναλιών που μπορούν να μεταφέρουν πληροφορίες από ένα νήμα σε μια οικογένεια στα γειτονικά του. Εάν οι πόροι του συστήματος εξαντληθούν, τότε το σύστημα είναι ικανό να επιστρέψει σε κατάσταση σειριακής εκτέλεσης. Δύο γλώσσες προγραμματισμού δημιουργήθηκαν για τον προγραμματισμό του microgrid σε ένα υψηλότερο επίπεδο: μTC και SL. Και οι δύο περιγράφονται στο κείμενο, και η βασική τους λειτουργία είναι να επεκτείνουν την γλώσσα C με τέτοιο τρόπο ώστε να μπορούν να ελέγχουν την δημιουργία και την εκτέλεση των οικογενειών από νήματα.

Στην συνέχεια ο αυτόματος μεταφραστής C2μTC/SL παρουσιάζεται και περιγράφεται: Ο σκοπός του είναι να δέχεται ως είσοδο ένα οποιοδήποτε πρόγραμμα γραμμένο σε C και να το μεταμορφώνει σε ένα παράλληλο πρόγραμμα SL. Αρχικά η έξοδός του ήταν η γλώσσα μTC αλλά με την εμφάνιση της SL ο μεταφραστής προσαρμόστηκε ανάλογα, οπότε και το όνομά του C2μTC/SL. Η βασική δομή για την οποία ενδιαφέρεται ο μεταφραστής είναι οι βρόχοι μιας και το μεγαλύτερο ποσοστό του χρόνου εκτέλεσης σε ένα πρόγραμμα είναι οι βρόχοι. Εφ' όσων το SVP δουλεύει με οικογένειες από νήματα που μοιάζουν με μονοδιάστατους βρόχους, η μετατροπή ενός οποιοδήποτε βρόχου σε οικογένεια νημάτων είναι ένα σημαντικό βήμα. Για τον λόγο αυτό, οι βρόχοι χωρίζονται σε μονοδιάστατους και πολυδιάστατους με κάθε

κατηγορία να χρειάζεται και διαφορετική αντιμετώπιση όσον αφορά την μετατροπή του κώδικα που χρειάζεται.

Οι μονοδιάστατοι βρόχοι χωρίζονται περαιτέρω σε κατηγορίες ανάλογα με τις εξαρτήσεις που βρίσκονται στον βρόχο (loop carried dependencies). Βρόχοι χωρίς εξαρτήσεις απλά μετατρέπονται σε πλήρως παράλληλες οικογένειες ενώ οι βρόχοι με εξαρτήσεις μετατρέπονται σε οικογένειες που χρησιμοποιούν τα ειδικά κανάλια συγχρονισμού του SVP για να μεταφέρουν δεδομένα από τον ένα νήμα στο επόμενο με την μορφή της ροής δεδομένων (data flow). Αυτού του είδους η μετατροπή επιτρέπει στα νήματα να έχουν τα δεδομένα που χρειάζονται χωρίς να χρειάζεται να τα αναζητήσουν στην κεντρική κοινή μνήμη, πράγμα «ακριβό» από άποψη χρόνου. Όταν κάθε νήμα τελειώσει τον υπολογισμό που του αναλογεί, όλα τα σχετικά δεδομένα μεταφέρονται στο επόμενο νήμα μέσω του ειδικού καναλιού επικοινωνίας του SVP. Ο συνδιασμός της εκτέλεσης παράλληλων ροών δεδομένων με την χρήση των ειδικών καναλιών επικοινωνίας προσφέρει μεγάλες αυξήσεις στην αποδοτικότητα και στην επιτάχυνση ενός προγράμματος.

Οι πολυδιάστατοι βρόχοι επίσης χωρίζονται σε υποκατηγορίες. Η πρώτη δεν περιέχει εξαρτήσεις και κάθε επίπεδο στον βρόχο μπορεί να μετατραπεί σε μια πλήρως παράλληλη οικογένεια αναθέτοντας στο περιβάλλον εκτέλεσης του SVP την εξισορρόπηση βάρους μεταξύ των πυρήνων του microgrid. Η δεύτερη (και πιο ενδιαφέρουσα) κατηγορία περιλαμβάνει βρόχους που περιέχουν στατικές εξαρτήσεις. Η προσέγγιση του Lamport με τα υπερεπίπεδα (hyperplanes) χρησιμοποιείται σε αυτήν την περίπτωση αλλά με μια καινοτομία: Αντί να γίνουν οι απαραίτητοι (δύσκολοι σε πολλές περιπτώσεις) υπολογισμοί σε χρόνο μετάφρασης, το περιβάλλον εκτέλεσης αναλαμβάνει να εντοπίσει όλα τα στοιχεία που μπορούν να εκτελεστούν παράλληλα ανά κύκλο εκτέλεσης ακολουθώντας διαισθητικά τον πίνακα εξαρτήσεων. Αυτή η ιδέα οδήγησε στην δημιουργία του πρώτου μας αλγορίθμου χρόνου εκτέλεσης: Τον αλγόριθμο σταθερού μεγέθους (Fixed Sized Algorithm). Είχε την δυνατότητα να εντοπίζει τα κρυμμένα υπερεπίπεδα την ίδια ώρα που εκτελούσε τον ίδιο τον κώδικα του προγράμματος. Ο χώρος αναζήτησης των δεικτών των βρόχων χωρίζεται σε μεγέθη σταθερού μήκους κατά το πιο εσωτερικό βρόχο. Ο παραλληλισμός επιτυγχάνεται μεταξύ των κομματιών σταθερού μήκους ενώ κάθε τμήμα εσωτερικά εκτελείται σειριακά. Ενώ ο αλγόριθμος δούλεψε σωστά, καλές επιταχύνσεις επιτυγχάνονται μόνο εάν το σταθερό μήκος ήταν κατάλληλα επιλεγμένο εκ των προτέρων, κάτι πρακτικά αδύνατον αφού κάθε πρόβλημα έχει το δικό του βέλτιστο μέγεθος. Αυτό το πρόβλημα μετέτρεψε τον αλγόριθμο σε ένα καλό πρώτο βήμα και σε ένα εργαλείο για συγκρίσεις.

Αυτή η αδυναμία του αλγορίθμου σταθερού μεγέθους καλύφθηκε με τον αλγόριθμο που υπήρξε ο εξελικτικός απόγονος του αρχικού. Τον αλγόριθμο αυτό-μεταβαλλόμενου μεγέθους (Self-Adaptive Algorithm). Χρησιμοποιώντας τις ίδιες αρχές με τον αλγόριθμου σταθερού μεγέθους, μπορούσε σε χρόνο εκτέλεσης να μεταβάλλει το μέγεθος τον τμημάτων βάσει κάποιων μετρικών από κύκλο σε κύκλο.

Τα πειραματικά αποτελέσματα δείχνουν ότι ο αλγόριθμος μεταβαλλόμενου μεγέθους επιτυγχάνει επιταχύνσεις σχεδόν ίσες με τα βέλτιστα αποτελέσματα για αυτού του τύπου τον παραλληλισμό. Οι αλγόριθμοι επίσης συγκρίθηκαν με μια τυπική μέθοδο χρόνου μετάφρασης και βρέθηκε ότι σε κάποιες περιστάσεις τα αποτελέσματα είναι κοντά. Αυτό μαζί με την ευελιξία της μεθόδου του χρόνου εκτέλεσης (π.χ. αντιμετώπιση μη ορθοκανονικών βρόχων) κάνει τον αυτό-μεταβαλλόμενο αλγόριθμο ειδικά ελκυστικό.

CHAPTER 1. INTRODUCTION

Concurrency in computation is by no means a new concept. It has existed since the 1960s and has steadily improved since then. The reason is simple, to speed-up an application, one either needs a faster CPU, or more than one CPUs sharing the computational load. Thusly, concurrent research was an entirely different research branch that took place in tandem with traditional CPU research. However, only recently has the existence of multiple *cores* in systems become prevalent. The latest generations of PC CPUs carry 2 or 4 or even 6 cores inside them and the trend has moved to include smart phones (it is common to see smart phones with 2 or 4 cores), tablets and more. It is safe to assume that with the current technology on CPUs reaching its limitations that multi-cores will become ever more prevalent in the technological world.

Programming a parallel system though is much harder than programming a sequential one. A coder will either write an application from scratch utilizing some parallel library, or will use pre-existing modules that have been proven to work and orchestrate them together. Moreover, there is plenty of legacy code in existence that was created with only one core in mind. The challenges involved with writing good parallel code coupled with the existence of sequential code led to the development of automatic parallelizing tools. These tools are compilers that either compile from source code to a different parallel source code (source-to-source) or compile to parallel binary code directly. Creating such an automatic parallelizing compiler though is not without its own challenges and the purpose of this paper is to describe such a compiler.

Prior to the presentation of our compiler, some general information is firstly required: The second chapter offers a small glimpse on the tremendously huge research work that has been done on the automatic parallelization area mentioning not only techniques and algorithms but whole compiler projects that existed (and some

still do). The third chapter offers some insight on some of the loop transformation techniques that exist before moving on to the fourth chapter which introduces the SVP architecture.

The SVP architecture is a novel contribution which describes a new type of multi-core system. Each core can carry its own OS as an extension of the instruction set and can achieve high memory latency tolerance coupled with low energy needs (and thusly low heat emission and distribution). Many SVP cores form a microgrid which is capable of offering true parallel execution of code as well as automatic resource allocation and graceful degradation when it starts to run out of resources. Its novel contribution is the existence of synchronized data channels that can impose an order on the execution of threads as well as carry data between threads in a dataflow manner. The same chapter also describes the programming language which was created specifically for the SVP: The μTC/SL language, an extension of C with added constructs that describe concurrency.

The fifth chapter presents the C2μTC/SL source-to-source automatic parallelizing compiler. A tool capable of reading in a code written in the C language, analyzing it to discover any potential for parallelism and finally outputting a different program in the SL language which has the same functionality with the original one, with the difference that it is faster since it takes advantage of SVP's mechanisms. Each type of loop is described alongside a way to transform it for the best possible results.

C2μTC/SL's main contribution though is its approach on the multi-dimensional loops with static dependency vectors. Borrowing heavily on the hyperplane (wavefront) idea, it utilizes a run-time algorithm which discovers the underlying hyperplanes. Instead of resorting to heuristic methods or expensive integer programming functionality to calculate the hyperplanes, it delegates that discovery to the run-time environment. The idea is simple: At any given time, when there is a known set of executing threads and a known dependency vector, by applying the vector to the set it is possible to find the set of the next computational cycle. It is an elegant and intuitive idea that of course became much more convoluted when it was actualized as part of the code.

Chapter six evaluates the outputs of C2μTC/SL. For each different loop type, an example is transformed into SL and then executed and compared with its original

form. More interestingly, the efficiency of the run-time algorithm is tested. A theoretical target is first calculated for three different examples and then it is proven that the run-time algorithm can reach it and even surpass it at some cases. It is also compared to some standard compile-time transformation method. The results are encouraging enough (as expected the run-time method can never compete against a method that lacks all of its overheads but it can get relatively close).

Finally, the last chapter (seventh) provides a discussion on everything mentioned in the previous chapters as well as a conclusion and general thoughts on current as well as future work.

CHAPTER 2. RELATED WORK

2.1. Introduction on Parallel Systems and Threads

2.2. Developer Tools which Enable Parallel Programming

2.3. Dependencies and Parallelism Detection

2.4. The Polyhedral Model and Related Methods

2.5. General Parallelization and Run-Time Methods

2.6. Overviews, Surveys, Tutorials and Books on Automatic Parallelization

2.7. List of Parallelizing Compilers

2.1. Introduction on Parallel Systems and Threads

Parallel systems appeared early on in the history of computation. Soon after, various types of systems had already existed and many more were on the way. In an attempt to classify the ever increasing types of parallel system, Flynn on his work on taxonomy [29] separated systems on whether they are Single Instruction or Multiple ones i.e. whether there is a single Control Unit (CU) (which can direct Processing Elements (PE)) or multiple ones and whether there is a single or Multiple Data Streams. The resulting classification can be seen on Table 2.1.

Table 2.1 Flyn's classification of Parallel Systems.

	Single Data stream	Multiple Data streams
Single Instruction	SISD	SIMD
Multiple Instructions	MISD	MIMD

From that table we can see that Flynn discerned four distinct categories:

1. **S**ingle **I**nstruction - **S**ingle **D**ata (*SISD*).

A single controller directs a single Processing element to operate on data from a single data stream. All conventional computers fall into this category.

2. **S**ingle **I**nstruction - **M**ultiple **D**ata (*SIMD*).

A single controller directs multiple Processing elements to operate on data from multiple data streams. The old Vector computers (a vector is a single dimensional array, so a vector computer could operate a single instruction on various parts of the array simultaneously) belong to this category as well as the modern GPUs.

3. **M**ultiple **I**nstructions - **S**ingle **D**ata (*MISD*).

This category makes little sense in general. It involves a series of processing elements performing calculations on a single data stream. In theory such a system can be used for fault tolerance where a series of computers must agree on a result before it can be accepted as correct. No computer of this category has ever been created.

4. **M**ultiple **I**nstructions - **M**ultiple **D**ata (**MIMD**).

This is a rather diverse category of systems. It includes parallel systems with processing units and memory systems created especially with parallelism in mind, parallel systems built with off-the-shelf computers connected in some form of interconnection network and so on.

A 5th category was later introduced, the **S**ingle **P**rogram – **M**ultiple **D**ata (***SPMD***). More a programming style than an actual architecture itself, it became the dominant paradigm for parallel programming. The main idea is that a number of independent processors execute the same program at different points simultaneously. This means that a single computer / processor begins executing the code and at particular points in the code, it might spawn a parallel execution of that code. The way this programming style is implemented differs depending on whether it is applied on a distributed memory system or a shared memory one.

A Distributed Memory System is a parallel system consisting of a series of independent computers called nodes. Communication and synchronization are achieved by message passing over any network such as TCP/IP or Ethernet. A Shared Memory System is a computer with a series of CPUs which have access to the same

memory space. In such a system, the SPMD is actually a series of directives that mark areas of the code as ones that should execute in parallel. Once control reaches these points, the rest of the CPUs begin executing the marked code in parallel.

The most commonly used parallel construct in a Shared Memory System is the *Thread*. A thread is essentially a part of a program (a procedure or a function) which can run independently from the main program. In the presence of more than one processors / cores, threads can run simultaneously with the main program. Their characteristic is that their creation and destruction are relatively light-weight processes (especially when compared with Fork which duplicates the entire application) and that large number of threads can exist at any given time with a very small footprint on the host Operating System's (OS) resources. However, they are anchored to the main application so if the program ceases to exist, so do all threads associated with it. Threads share the same address space between themselves and the main program so, basically, Multi-Threaded programming, and applications in general, can only work on Shared Memory Systems.

2.2. Developer Tools which Enable Parallel Programming

In order to utilize parallelism, there exist various different tools and APIs which developers can utilize, depending on their applications and targeted architectures. A small (and by no means comprehensive) list of such tools follows:

- PThreads
- OpenMP
- MPI
- Nvidia's CUDA
- Intel's Cilk

First and foremost is the lib-pthread library. The API (Application Programmer Interface, a set of function / procedure calls that defines how a software component interacts with the rest) was composed by IEEE as part of the POSIX interface so that all POSIX-compatible OSes could offer the same functionality to applications and ease the transitioning of code from one platform to the next. The PThread interface offers the developer a multitude of tools with which to implement parallel applications (thread management like creating, destroying, detaching threads

and so on, mutex functions and condition variables). Figure 2.1 Demonstrates the creation of pthreads in the C language.

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *TheadBody(void *id)
{
    int id = (int)threadid;

    printf("thread #%d executing\n", id);

    return NULL;
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int result;
    int t;

    for(t=0; t<NUM_THREADS; t++)
    {
        printf("creating thread %d\n", t);

        result =
          pthread_create(&threads[t], NULL, ThreadBody, (void *)t);

        if (result)
        {
            printf("ERROR! return code=%d\n", result);
            exit(1);
        }
    }

    for (t=0;t <NUM_THREADS;i++)
     pthread_join(threads[t], NULL);

    return (0);
}
```

Figure 2.1. Using the Pthread Library to Create Threads.

OpenMP (Open Multi-Processing) ([67], [68]) is a higher level tool than threads (and PThreads) which allows multi-threaded programming in a cross-platform way enabling both task parallelism and data parallelism. It can be used with the C/C++ languages as well as FORTRAN and its main use is the transformation of loops into a series of threads that can produce the same result in a concurrent manner.

The programmer is responsible for marking the areas of the program that OpenMP will assign into threads using specific macros (in the case of the C/C++ languages, #pragma is used to mark code areas). The programmer is also responsible for identifying which variables are private, shared or induction ones and which variables are reduction variables in order for OpenMP to work properly and offer speed-ups to the original code. Once everything has been identified properly,

OpenMP divides the whole index space of the loop into a number of threads (each with each own id) that perform computation in parallel. Once all threads finish computation they *join* with the original program and control moves on.

Thread scheduling may also be configured by the programmer as OpenMP offers a series of different schedules with the *dynamic* schedule being the most popular, since it allows threads that have finished their part of the computation to pick up some of the remaining work that awaits computation. This leads to better load balancing at the cost of more expensive setting up and tearing down. The benefit of OpenMP is that it offers a higher abstraction level to the programmer alleviating the need of handling each thread manually and focusing on the actual idea behind the program itself. Another advantage of it is that if the compiler is not an OpenMP compatible one, it will just ignore all relevant #pragma directives and just compile the program into a classic sequential form. Clearly, as OpenMP is a thread-enabled API, its use was originally restricted only to Shared Memory Systems however a combination of Message Passing and OpenMP could circumvent this restriction. Additionally, extensions on the OpenMP model have allowed its use on non-Shared Memory Systems as-is.

Figure 2.2. shows a typical example of an OpenMP-enabled source code. It applies an iterative method for the computation of the value of pi. It is worth noting that the variable "sum" was declared as a reduction one (a summation variable) which caused the system to adapt accordingly and add all the values in parallel.

For the sake of completeness, MPI, CUDA / OpenCL and Cilk are also mentioned, as they are important parallelization tools. *MPI* (Message Passing Interface) [69] is an API that allows the programmer to transform any network of computers into a parallel Distributed Memory System. With MPI a programmer can divide a problem in smaller ones, scatter the data over the network to each computer for computation and then gather back the results from for the final result.

The clear advantage of MPI is that it provides an inexpensive way to perform complex computations quickly and easily without requiring any sort of shared memory between processors. Of course due to the fact that it relies on an interconnection network (such as Ethernet) as a data transfer medium, this means that it will get quite slowed down. In classic network cases (i.e. not ultra low latency ones)

the only way to offset the slow data transfer is to resort to coarse grain parallelism when working with MPI. Working on a purely distributed system where each CPU has access only on its own part of the data means that MPI is better suited for problems which can be divided cleanly and without any dependencies hidden in the loop. This makes MPI ideal for data parallel programs but inadequate to deal with task-based parallelism. Figure 2.3. shows a simple MPI program which calculates the value of pi over a network.

CUDA [70] is a relatively new (since 2007) tool for parallel computations. The main idea is that it opens the GPU of any system (which so far had been used only for graphics related calculations) to the programmer for general programming. GPUs support thousands of concurrent threads running simultaneously and by exploiting that any application can become an order of magnitude faster. The CUDA platform exists in various forms: from a series of libraries and compiler directives, to extensions of industry-standard languages like C/C++, FORTRAN and more. Due to the nature of a GPU (usually a SIMD machine), it is better suited for data parallel applications.

```
#include <stdio.h>
#include "omp.h"
double f(double a) {return (4.0 / (1.0 + a*a));}
int main ( int argc, char **argv ) {
        int i,n=1000000;
        double sum= 0.0, x, h, mypi;
        int chunk;
        h   = 1.0 / (double) n;
        chunk=n/4;
#pragma omp parallel private(i,x) shared(sum,n,h)
{
        #pragma omp for schedule(runtime) reduction(+:sum)
        for (i=1;i<=n;i++) {
                x=h*((double)i-0.5);
                sum=sum+f(x);}
}
        mypi = h * sum;
        printf("pi=%f\n",mypi);
        return (0);
}
```

Figure 2.2. Using OpenMP to Calculate the Value of pi in Parallel. Letters in Bold Indicate where the Computation Takes Place.

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
double f(double a) {
    return (4.0 / (1.0 + a*a));
}
int main(int argc, char *argv[])
{
    int done = 0, n=0, myid, numprocs, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    int  namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);
    while (!done)
    {
        if (myid == 0)
            if (n==0) n=100; else n=0;
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0) done = 1;
        else
        {
            h   = 1.0 / (double) n;
            sum = 0.0;
            for (i = myid + 1; i <= n; i += numprocs)
            {
                x = h * ((double)i - 0.5);
                sum += f(x);
            }
            mypi = h * sum;
             MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM,
                        0, MPI_COMM_WORLD);
        }
    }
    MPI_Finalize();
    return 0;
}
```

Figure 2.3. Using MPI to Calculate the Value of PI in Parallel. Letters in Bold Indicate MPI-specific Directives.

Finally, Cilk [71] is a general-purpose programming toolset containing the Cilk programming language and a runtime environment. It was originally developed in MIT and was later acquired by Intel. Its driving principle is that the programmer is responsible for exposing the parallelism in her code, identifying which parts can be fully parallelized. In turn, the runtime environment handles everything from delegating the work to any available processor/core to load balancing and scheduling. This attributes Cilk programs with the "compile once, run anywhere" capability. The language itself is a superset of C which supports the entire C language specification extended by a few keywords that offer the necessary functionality. Load balancing is achieved by a system of "work-stealing": Each idle processor can attempt to "steal" a

piece of workload in a queue of a non-idle processor through the scheduler. Since the package can be stolen from the end of the queue, it would be the last piece of load that its original owner would have to work on.

2.3. Dependencies and Parallelism Detection

The tools mentioned so far are the ones most commonly used when it comes to making an application which exploits hardware parallelization. However they are too low-level in their abstraction level and require the developers themselves to know when and how to properly use them. Automatic Parallelization solves that problem but it requires a different set of actions.

The first step to automatic parallelization is the examination and analysis of the source code of the application in question. The analysis will decide whether the code can be executed in parallel or not. Most of the analyzer techniques and tools focus solely on loops (for reasons that will be later described) and whether or not they carry dependencies.

A dependency between two statements (S1 and S2) in a code exists when both statements access the same memory location. Four types of dependencies exist: (i) Flow Dependency: S2 is flow dependent on S1 when S1 writes to a memory location which is later read by S2. (ii) Anti Dependency: S2 is anti dependent on S1 when S1 reads the value of a memory location which is later written by S2. (iii) Output Dependency: S2 is output dependent on S1 when S1 writes a value to a memory location which is later re-written by S2. (iv) Input Dependency: S2 is input dependent on S1 when S1 reads the value of a memory location which is later re-read by S2.

From all these types, Anti, Output and Input are not real dependencies and can be removed with various techniques that will presented later on. Dependencies that exist inside a loop but between statements of different index instances are called Loop Carried Dependencies. Dependencies that exist inside the same loop iteration are called Loop Independent Dependencies because they do not affect the re-ordering of the loop iterations in any way.

The existence of a dependency effectively imposes an order in the execution scheme of the loop statements (and iterations accordingly). Since an ordering exists, it becomes harder or even impossible to parallelize a loop with dependencies. The

ordering of a loop carried dependency on a loop can be seen by unrolling the loop. All statements need to be executed in the precise order that the dependencies allow.

Because of the existence of such dependencies, parallelizing compilers apply a series of tests on the statements of the loop in order to deduce whether the loop can be fully parallelized or not. These tests usually rely on array subscript accesses and can be either certain that there are no dependencies and hence the compiler can proceed to fully parallelizing the loop or be uncertain and thusly most compilers would just leave the loop intact. Since an array index can be any expression, usually the simple expressions in the form of $c1*i+c2$ are examined where $c1$ and $c2$ are constants. More complex expressions usually classify a loop as non-parallelizable. A random loop may contain statements which access an array in the following style:

$$Array[c1*i1+c2]=…$$
$$…=Array[c3*i2+c4]$$

A dependency will exist if $c1*i1+c2=c3*i2+c4$ or $c1*i1-c3*i2=c4-c2$. As with any Diophantine equation, if the Greater Common Divisor (GCD) of $(c1, c3)$ divides $(c4-c2)$ then the equation has a solution and hence a dependency exists. Hence the GCD test can safely reply that there is no dependency when the GCD of the left-hand side of the equation does not divide the right-hand side of it for every equation in the loop.

However the usual case has it that the left-hand side GCD equals to 1 which will always divide the right-hand side and hence its reply will be that there might be a dependency. Hence other tests came to existence to cover for this weakness. The extreme value test calculates the minimum and maximum possible values of the left-hand side of the dependency equation and compares it to the right-hand side. If the maximum value of the right-hand side is greater than the maximum of the left-hand side or if its minimum value is lesser than the minimum value of the left-hand side then there are no dependencies. A combination of the Extreme Value test and the GCD usually provide satisfactory results.

Another classic dependency analysis test is the Omega Test [56]. It uses the Diophantine equation to create an linear programming problem and then attempts to solve it quickly by applying Fourier-Motzkin Elimination. Even though at its worst case it completes in an exponential time, at most real life programs it finishes quickly

at a polynomial time. Other tests include the Lambda test [30] (an increased-precision form of the Extreme Value test), the I test [43] (a combination of the GCD and Extreme Value tests but more precise than the application of the two tests individually) the Generalized GCD test, built on Gaussian Elimination (adapted for integers) and the Power Test [64] (first uses the Generalized GCD test then it uses constraints derived from the program to determine lower and upper bounds on the free variables of the parameterized solution. Fourier-Motzkin elimination is used to combine the constraints of the program for this purpose)

There exist more methods for the detection of inherent parallelism. In [12] the authors approach actual real life complex programs and propose symbolic analysis in order to make conclusions about the code. Symbolic analysis, in general, relies on scanning all the statements of the code and for each statement mentioned information is kept about the potential values of all the involved variables. These value ranges can then be used to make deductions about various aspects of the code including array accesses and parallelism. Using symbolic analysis, the Range Test [18] extends the Extreme Value Test to support symbolic and non-linear array subscript expressions. In a similar manner, the same kind of analysis is used on [35] in order to discover parallelism that can be exploited between procedure calls.

Most of the tests (and especially symbolic analysis) rely on statically defined variables and their interactions in the code. In [49] the authors innovate by checking for the existence of heap based variables and data structures. Examples include linked lists, binary trees, heaps and so on. A methodology is presented where the algorithm tries to detect the shape of the dynamic structure and depending on that, determine what kind of dependencies exist in that structure.

Finally, loops which carry loop carried dependencies are examined in [25]. A dependency can be seen as a distance between loop iterations. These distances between array accesses form a vector. All the vectors are grouped together in a set called *distance* or *dependency* vector. If the distances are of a constant size throughout the computation then some degree of parallelism is possible as we will discuss later. The authors of this paper not only detect the possibility of the existence of hidden parallelism but also define the granularity that must be used for better results.

2.4. The Polyhedral Model and Related Methods

As mentioned in the previous paragraph, early automatic parallelizing compilers would operate on a black or white state. If there were no dependencies detected inside a loop, then the loop would be fully parallelized in various forms depending on the architecture. However, the existence of any dependency would signal the compiler to leave the loop completely intact and move on.

Lamport with his work on the Polyhedral model [46], introduced a methodology according to which a perfectly nested loop with a static dependency vector could be transformed into an equivalent loop whose innermost dimension could be fully executed in parallel. In this manner, even though it would be impossible to gain full parallelism, some partial form of it would still be exposed and exploited. If the whole index space of the loop is visualized in N dimensions then it is bound by a *polyhedron* and through transformations it is possible to have a series of *hyperplanes* move through that polyhedron. Each index set that belongs to a certain hyperplane is independent from the rest of the index sets on the same hyperplane. Since the hyperplanes resemble a wave moving through the data, this method is also referenced as the *wavefront* model. More information on the wavefront transformation can be found in Chapter 3.

In [65] the writers propose a unified transformation model that is based on matrices. Matrix transformations are an intuitive method that can be applied to nested loops and offer a variety of results according to the current needs. A special form of such a matrix is the unimodular matrix (a matrix whose determinant is equal to 1 or -1 composed of integers) and in that paper these matrices are the basis of the unified model proposed. Their technique can also be applied to general nested loops where the dependencies not only form a static dependency vector but also a more general direction vector (the distances are variable and only the directions are known). The use of unimodular matrices has also been proposed by [48]. In that paper an algorithm based on unimodular transformations is proposed which maximizes parallelism and minimizes communications while at the same time keeping a minimum degree of synchronizations in programs with arbitrary loop nests.

A general automatic source-to-source framework based on the polyhedral model that can optimize programs (even sequences of possibly imperfectly nested

loops) for parallelism and locality is introduced in [21]. This is achieved by the use of integer linear optimizations which aim to detect good tiling schemes that lead to better locality. Locality is important since it allows for better cache utilization and a great boost in efficiency overall. A similar methodology is described in [20], where an algorithm is described which can calculate hyperplanes of tiles in a sequence of arbitrarily nested loops which minimize communication and improve on data locality.

Finally, a framework that incorporates a series of methods and which is able to utilize a variety of functions including non-uniform and even non-unimodular transformations is proposed in [14]. In addition to the suggested framework, a series of improvements on existing algorithms are proposed.

2.5. General Parallelization and Run-Time Methods

It goes without saying that not all automatic parallelizing compilers and techniques in general are based on the polyhedral model. In [60] a technique is used for automatic array privatization. Array privatization is the analogue of scalar data privatization presented in Chapter 3. If it is safe to do so, an entire array can be copied to a thread's local memory for local accesses. Each concurrent thread has its own version of the array. Not only this technique can help increase efficiency but it also helps to remove false dependencies. Array privatization is an important part of any array access analysis and it enables the full parallelization of a loop and is especially useful for vector and super scalar machines. In the current paper, data flow analysis is used to identify privatizable arrays inside and between procedure calls. On the subject of vector machines, [1] introduces a method where dependency analysis is used in FORTRAN loops in order to transform them into parallel constructs which can be executed by vector machines for better data parallelism.

A compiler is proposed in [4] which not only applies a series of transformations on programs with the intent to minimize synchronization and data sharing but is also capable of re-arranging parts of an array and its layout in order to improve data locality and increase efficiency of the memory subsystem. An algorithm is also suggested in [5] which optimizes parallelism and data locality at the same time, but its novelty lies with the fact that the algorithm can target both shared memory systems and distributed memory ones.

The majority of analyses on loops would not try to tackle loops which contained procedure calls. The need to deal with function/procedure calls from the inside of a piece of code in question gave birth to interprocedural analysis and transformations. Such an analysis tries to apply the side effects of the procedure on the resulting call in order to help expose parallelism to the code in question. The most common interprocedural transformation is procedural *inlining* which substitutes the procedures code into the place of the call.

In [33] the authors suggest a methodology, according to which, two different kinds of interprocedural transformations are applied to loops which contain procedure calls (something that the original hyperplane method cannot deal with, since it requires that any function / procedure call must not alter data in any way, in other words, contain no side effects) for parallel code generation. Perfectly nested loops are also the main research target of [15] but its purpose is to use linear transformations for the parallelization of loops with no uniform dependencies.

Finally in [6] interprocedural analysis is used to determine the shape of dynamic data structures based on the heap and its subsequent parallelization while instruction level parallelism is the focus of [61]. It is an idea that any parallelizing compiler can use in theory, since it can be applied to any statement, inside and outside of loops. The aforementioned paper examines the limits of instruction level parallelism as well as the amount of said parallelism that exists inside a typical program.

There is also an entirely different category of methodologies for automatic parallelization. It incorporates the run-time environment into the solution of the problem. It is a bold and novel way of approaching this problem since the run-time environment by definition contains a lot more information that can be used. The methods that belong to this category usually gather enough information and perform some computation during compile time but the rest is delegated to the actual execution of the application.

[13] proposes such a solution. According to the authors, an automatic parallelization method is proposed which is split in two parts. The first part takes place during compilation and it generates code which will enable dependency detection between tiles at run time. At run time, execution of the generated code takes

place alongside the second part of the method which is responsible for proper load balancing between cores in order to improve scalability. Similar to that method is [57], where two pieces of code are generated during compile time. During execution, the first piece of code can be executed fully in parallel as it follows the dependencies' access patterns and the second schedules the execution of the threads. Array privatization and reduction are also applied in that method. Likewise in [59] there is an attempt to solve the problem by exploiting the run-time environment. Its main difference is that it is aimed at FORTRAN programs and that it proposes a different loop structure altogether: the "DoConsider". DoConsider encapsulates a number of transformations that can expose hidden parallelism in a loop with dependencies. Predominant is the wavefront transformation yet other topological methods are also used. In compile time a dependene analysis framework is created which is executed at the start of the code. During run time, both analysis / transformation and load balancing take place.

2.6. Overviews, Surveys, Tutorials and Books on Automatic Parallelization

Since automatic parallelization tools have been around for a very long time, there is a lot of experience and expertise gathered on the subject. A series of overviews and tutorials exist that describe various methods and aspects of automatic parallelization for any architecture or programming language and paradigm.

The authors of [50] mostly focus on FORTRAN and discuss many common and uncommon traits a parallelizing compiler must have in order to efficiently generate parallel code for vector and multiprocessor systems. Standard compiler techniques are also examined and related to / compared with their corresponding parallelizing ones.

A survey on automatic parallelization techniques which covers a broad range from dependency analysis to program / loop transformations is the main subject of [12]. It even goes into the parallelization of recursive functions and ends up with an experimental study on the efficiency of several parallelizing compilers.

In [7] the authors present a comprehensive study on all the important parallelization techniques for C and FORTRAN. Each transformation is covered in depth, its purpose is clearly explained and examples are given for its applications on

various types of parallel (or even sequential) architectures. Tests on legality of each transformation are also explained and applied.

Dependency analysis is the interest of [37], both on whether dependencies do exist and if they do, which is the resulting direction vector. Various concepts are considered based on the dependency vectors that might exist, while computation on parallel, vector and serial DO loops (FORTRAN) is covered. Several transformation examples where data dependency analysis is required are given such as vectorization, concurrentization, scalarization, loop interchanging and loop fusion.

The writers of [45] present an overview in the form of a tutorial on the restructuring of sequential programs so as for them to have increased efficiency in parallel machines. Work (either previous or at that time current) on the transformations and partitioning of loop structures and data is presented. These transformations aim to improve parallelism, data locality and load balance. Finally a unified parallelizing framework is suggested by the authors.

The authors' aim in [39] is two-fold. At first, a comprehensive overview is given on parallelizing algorithms. Each algorithm is exhaustively analyzed, from the type of internal representation it uses to store the dependencies, to the code they generate and their optimizing criteria (for example if each algorithm aims for maximum parallelism, or minimal communication or even ease of code generation). The second part covers a discussion on a particular class of multi-dimensional schedule referred as shifted linear schedules and that algorithms based on that produce simpler code.

Finishing with the various surveys and overviews two more need to be mentioned. An early work on researching and documenting techniques on the parallelization of FORTRAN loops that contain dependencies is presented in [54]. Those techniques aim to transform loops in DOALL and DOACROSS forms while in [41] a rather comprehensive survey on a multitude of techniques that exist and used by parallelizing and vectorizing compilers is presented. In addition to all the aforementioned papers, there exist a series of books on automatic parallelization and associated compilers. [31], [9], [44], [63], [3], [10], [2], [26] and [66] is just a small sample of the work on this particular subject.

2.7. List of Parallelizing Compilers

This chapter will finish by listing a few well-known automatic parallelizing compilers. First in the list is the OSCAR compiler [34], [36]. OSCAR tries to exploit parallelism in multiple levels. It starts with parallelism existing between procedure calls, moves to loops, basic blocks and finally attempts to exploit the finest grain of parallelism possible by attempting to parallelize on a per-statement basis (instruction level parallelism). OSCAR consists of three parts. The first is the FORTRAN frontend which translates code to some internal representation (IR), then the middle part where all parallelizing transformations take place and finally there exist a series of backends, one for each target architecture. The range of various architectures is quite large as it encompasses SMP systems that use OpenMP, Clusters that support MPI and even the on-chip multiprocessor called OSCAR.

OSCAR decomposes a source program into three kinds of grain tasks namely MacroTasks(MTs) such as the Block of Pseudo Assignment statements (BPA), the Repetition Block (RB) and the Subroutine Block (SB). A BPA is defined as an ordinary basic block. However, a basic block is decomposed into several BPAs to extract larger parallelism when that basic block includes independent blocks. The compiler builds a Macro Flow Graph (MFG) which represents control flow among MTs. Next it analyzes the Earliest Executable Condition of each MacroTask to find maximum parallelism from a MFG. The Earliest Executable Condition for a MT represents a condition under which the MT can begin execution.

If a macro-task graph has only data dependencies and is deterministic, static scheduling is selected. In the static scheduling, an assignment of macro-tasks to threads is determined at compile time by the scheduler in the compiler. If a macro-task graph has control dependencies, the dynamic scheduling is selected to handle runtime uncertainties like conditional branches. The scheduling routines for the dynamic scheduling are generated by the compiler and inserted into a parallelized program with macro-task code. OSCAR also supports mechanisms for the reduction of Cache Conflict Misses.

The PROMIS compiler [23], [58] is multilingual, retargeting, parallelizing compiler. Again it is based on an internal representation (called Unified Internal Representation - UIR) but instead of opting for modular front-ends and back-ends,

both are integrated into the system. The designers made this choice because, at the time, modular systems lacked the ability to store and propagate dependency information. PROMIS exploits multiple levels of parallelism ranging from task-based parallelism, to loop level, to instruction level based on the target architecture. It relies on symbolic analysis which is further refined and augmented by pointer analysis for better results. Many standard optimization techniques are applied in the middle stage such as array privatization.

The frontend and backend operate on the same internal representation which maintains all vital program structures and provides a robust interface to users. The IR structures are semantic entities rather than syntactic constructs. It is based on the Hierarchical Task Graph (HTG) which is a hierarchical control flow graph overlaid with hierarchical data and dependency graphs. In the HTG hierarchical nodes capture the hierarchy of program statements and hierarchical dependency edges represent the dependency structure between tasks at the corresponding level of hierarchy. Therefore parallelism can be exploited at each level of the HTG: between statements, blocks of statements, blocks of blocks of statements and so on. The entire IR framework consists of the following: Symbol Table, Expression Trees, Control Flow Edges, Control Dependency Edges, Data Dependency Edges, Hierarchical Task Graphs and Call Graphs.

PROMIS aims at generating high performance code for the mainstream imperative programming languages such as C, C++ and FORTRAN. The IR represents a subset of the union of the language features of C++, FORTRAN and Java. This subset includes assignments, function calls, multi-dimensional array accesses and pointers arithmetic. Stack-based Java bytecode is translated into register-based statements and is applied with language independent analyses and optimizations. For example, exception detection code can be eliminated as deadcode if the compiler can prove the lack of exception. Such proof usually involves evaluation of symbolic expressions. If all catch blocks of a try block are eliminated the compiler may be able to convert the try block into a normal block.

The UIR propagates vital dependency information obtained in the frontend to the backend. Statements are represented as HTG nodes. During the construction of the HUIR (Higher UIR), expression trees are normalized to have a single side effect per

statement. Function calls and assignments to pointer dereferences are identified and isolated as separate statements. During IR lowering (from HUIR to LUIR – Lower UIR), complex expression trees are broken down to collections of simple expression trees, each of which is similar to quadruples. Data dependency information is maintained and propagated throughout the lowering process.

Symbolic analysis is performed via symbolic interpretation. Values (or ranges of values) for each variable are maintained by the interpreter in environments. These environments are propagated to each statement. Each statement is interpreted and its side effects are computed. These side effects are applied to the incoming environment of a statement resulting in new versions for the affected variables. Successive application of these side effects simulates the execution of the program. Pointer analysis is performed during interpretation.

Interprocedural analysis seamlessly integrates into the symbolic analysis framework. When a function call is encountered by the interpreter, its side effects are calculated and applied to the incoming environment, like any other expression. Once calculated, the side effects of a function call can be saved for subsequent interpretations. Several optimizations have been re-engineered within the symbolic analysis framework such as strength reduction, static performance analysis, induction variable elimination, symbolic dependency analysis and array privatization. Other techniques include constant propagation, dead code elimination and available expression analysis. The machine independent phase includes classical optimizations such as common sub expression elimination, copy propagation and strength reduction.

The Cetus Compiler Infrastructure [47], [8], [41], although not a full parallelizing compiler per se, is still a very helpful platform that can be easily molded into any kind of compiler the programmer wants. The Symbolic Manipulation provided includes the following techniques:

- $1 + 2*a + 4 - a$ => $5+a$ (folding)
- $a*(b + c)$ => $a*b + a*c$ (distribution)
- $(a*2) / (8*c)$ => $a / (4*c)$ (division)
- $(1-a)<(b+2)$ => $(1+a+b)>0$ (normalization)
- a && 0 && b => 0 (short-circuit evaluation)

Cetus' symbol table functionality provides information about identifiers and data types. Its implementation makes direct use of the information stored in declaration statements stored in the IR. There is no separate and redundant table storage. Cetus also provides data dependency analysis and tests: The framework identifies eligible loops. Eligibility currently defines the scope of dependency testing in Cetus. For example, it can handle perfectly nested loops and loops in the form for (i=lb;i<ub;i++) (canonical form loops). Loop information (such as loop bounds, loop step and enclosing loops) and array access-related information (such as array references, enclosing loops and parent statements) is collected in data structures and provided as input to the dependency test interface. The tests try to disprove dependency between a pair of array accesses and if unable to do so return a dependency vector representing the direction of dependency in each dimension of the iteration space spanned by the enclosing loop nest. Tests can be expanded to use standard tests like the GCD. The output of testing is a Dependency Graph (DG).

Cetus' Basic Parallelizing Transformation Passes include privatization, reduction variable recognition and induction variable substitution. Cetus also includes an automatic OpenMP to CUDA GPU translator and optimization techniques. It includes systems for dynamically adaptive applications which target MPI-based distributed irregular applications as well. More features include:

- Debugging aids: Cetus provides basic debugging support through the Java language which contains exceptions and assertions as built-in features. Cetus executes within a Java virtual machine so a full stack trace including source line numbers is available whenever an exception is caught or the compiler terminates abnormally.

- Readability of the Transformed Code

- Expression Simplifier

- Parallel Parsing: Use of Java threads to parse and generate IR for several input files at once.

- Detecting loop-carried dependencies in programs with dynamic data structures

Pointer analysis has also received significant attention. It can be divided into two distinct sub problems: stack-directed analysis and heap-directed analysis. The heap is represented as a storage shape graph and the analysis tries to capture some shape properties of the heap data structures. This type of analysis is called shape analysis and can help in detecting data dependencies induced by heap-directed pointers on loops that access pointer-based dynamic data structures, particularly in the detection of the loop-carried dependencies that may arise between the statements in two iterations of the loop. Shape analysis maintains topological information of the connections among the different nodes (memory locations) in the data structure. This representation provides a more accurate description of the memory locations reached when a statement is executed. The novelty is that this approach symbolically interprets the statements of the loop being analyzed and allows annotation in the real memory locations reached by each statement with read/write information.

Before the analysis the programs have to be preprocessed in order to normalize the pointer statements. That is, each statement dealing with pointers must contain only simple access paths each of which has the form p->field where p is a pointer variable and field is a field name. The following six simple instructions are considered:

- x = NULL
- x = malloc
- x = y
- x-> field = NULL
- x->field = y
- x=y->field

Basically the analysis is based on approximating by graphs (named Reference Shape Graphs – RSGs) all possible memory configurations that can appear after the execution of a statement in the code. Memory configuration means a collection of dynamic structures. Two statements in a loop induce a loop carried dependency (LCD) if a memory location accessed by one statement in a given iteration is accessed by the other statement in a future iteration with one of the accesses being a write access.

The POLARIS compiler [19], [55], [28] is a parallelizing compiler which uses FORTRAN codes as input and outputs FORTRAN code augmented with parallel directives. The main idea behind its design was the creation of a strong IR which would not allow any kind of error to exist and propagate to the output. Thusly the programmer is prevented from violating any rules and leaving the IR at an invalid or incorrect state. For that reason the IR contained not only static information but also data and data ownership information as well. Several transformation techniques are used such as inlining, induction variable elimination, symbolic dependency analysis, array privatization and even a framework exists for run-time analysis.

The SUIF (Stanford University Intermediate Format) Compiler system [32], [62] was originally designed to be a platform for research on high performance computing techniques on compilers. It is capable of producing code for multi-processors by detecting a coarse enough granularity size, ideal to be used for parallelization. Moreover, SUIF is equipped with a series of standard compiler techniques such as data dependency analysis, scalar and array privatization, reduction and induction variable elimination. In addition it employs basic data dependency tests on arrays to test whether two accesses are referring to the same location.

Interprocedural analysis is not actualized by the use of inlining but instead by analyzing the side effects of a procedure and then applying them to every statement in the code which calls that procedure. When the context differs, then a clone of that procedure is inlined and analyzed for further use. There is also a memory optimization module which allocates data in memory in continuous positions for shared memory systems. This improves data locality and cache usage while reducing false sharing.

Pluto [20], [21] is a source-to-source, automatic parallelization framework that uses the polyhedral model. It can transform arbitrarily nested loops with affine dependencies (defined as affine expressions of the indices and their coefficients) in such a manner where optimizations on locality and parallelism take place simultaneously. The approach aims at finding good hyperplanes (tiled) by applying integer programming with a cost function the authors developed. It has been assembled by a series of pre-existing tools such as CLooG (polyhedral scanning and code generator tool), Piplib (integer programming solver) and Polylib (a library that operates on objects made of unions of polyhedral) which were assembled together

with the authors' cost function in order to produce very efficient results. Code output can be either in OpenMP or CUDA.

Finally for the sake of completeness some lesser-known compilers will be briefly mentioned. Parafrase-2 [51], [52] is a parallelizing / vectorizing source-to-source code restructurer. It incorporates a series of analyses such as dependency analysis, overhead analysis and automatic scheduling. The renewed Paraphrase project was a system of code/design patterns. Each pattern would express high-level parallelism and had the ability to be refactored / redeployed in various heterogeneous hardware pools (a pool can have many different processing elements at each architecture). More information (with a list of papers on that project) can be found in [72].

The PARADIGM [11] compiler is a parallelizing compiler that targets multiprocessors that work with some form of message passing system (such as MPI). The compiler takes as input FORTRAN code and generates FORTRAN code augmented with message passing structures. PARADIGM employs a series of tools such data partitioning, communication costs estimation, exploitation of task and data parallelism and automatic support for multithreaded execution.

The ParaScope Programming Environment [24] is a parallel programming environment which incorporates the tools needed by researchers to create and debug parallel applications. It offers a parallel program editor, a compilation system and a parallel debugger. The editor assists the programmer by offering a series of analyses and interactive program transformations while the debugger uses run-time methods to detect and report timing-related errors.

Lastly, OMPi [27] is a lightweight, open source OpenMP compiler and runtime system for C, conforming to version 3.0 of the specifications. It takes C source code augmented with OpenMP #pragmas and produces transformed multithreaded C code, which can be compiled by the native compiler of the system. An optimized library has also been created which provides efficient runtime support. That library is linked against the program executable during compilation.

CHAPTER 3. LOOP TRANSFORMATIONS

3.1. Data Dependencies and the Polyhedral Representation

3.2. Loop Transformations

3.1. Data Dependencies and the Polyhedral Representation

In this chapter some of the most common loop transformations and restructuring methods will be presented. Most of the information presented in the entire chapter can also be found in more detail in [45] and [41].

Firstly a definition of dependency is needed. A *dependency* between two statements in the code exists when both statements access (by assigning a value or by referencing the value of) the same memory location (variable). According to that definition, four different types of dependencies can exist:

1. *Flow Dependency*. A Statement S2 is *flow dependent* on Statement S1 when S1 assigns a value to a variable which is later referenced by S2. It is also called a "WRITE before READ" dependency and is characterized as true dependency.

2. *Anti Dependency*. A Statement S2 is *anti dependent* on Statement S1 when S1 references the value of a variable which is later assigned by S2. It is also called a "READ before WRITE" dependency.

3. *Output Dependency*. A Statement S2 is *output dependent* on Statement S1 when S1 assigns a value to a variable which is later reassigned by S2. It is also called a "WRITE before WRITE" dependency.

4. *Input Dependency*. A Statement S2 is *input dependent* on Statement S1 when S1 references the value of a variable which is later referenced by S2. It is also called a "READ before READ" dependency.

It is obvious that from all the mentioned dependencies, Anti, Output and Input are not real dependencies and there are ways for them to be removed from code as will be later demonstrated. Generally, a (true) data dependency between two statements defines their execution order. Even after all the transformations and restructuring, those statements must still be executed in the original order as indicated by the dependency.

Dependencies that exist inside a loop but between statements of different index instances are called *Loop Carried Dependencies*. These kinds of dependencies can be made clear by completely unrolling the loop. Dependencies that exist inside the same loop iteration are called *Loop Independent Dependencies* because they do not affect the transformation of the loop in any way.

Let us consider a typical perfectly nested loop like the one presented in Figure 3.1. We can see that for a nesting level of *n* there exist n indices (i_1, i_2, …, i_n), each one with its own lower (L_1, L_2, …, L_n) and upper bound (U_1, U_2, …, U_n). These bounds can be a function of all the previous indices. So each for each i_k in the index space, there exist these inequalities: $L_k(i_1, i_2, ..., i_{k-1}) \leq i_k \leq U_k(i_1, i_2, ..., i_{k-1})$, where $1 \leq k \leq n$. The vector I which contains all those in I={i_1, i_2, …, i_n} is the *iteration space*.

```
for (i₁= L₁ ; i₁ <= U₁ ; i₁++)
        for (i₂= L₂(i₁) ; i₂ <= U₂(i₁) ; i₂++)

            . . .

            for (iₙ= Lₙ(i₁, i₂. …, iₙ₋₁) ; iₙ <= Uₙ(i₁, i₂, …, iₙ₋₁) ; iₙ++)
                Statements (i₁, i₂, …, iₙ);
```

Figure 3.1. A Perfectly Nested Loop in C. Unit Stride of 1 is Assumed.

There is another way to represent the problem: If we consider two n x 1 matrices, L and U that contain the lower and upper bounds respectively and the n x 1 matrix I which contains the indices of the loop, then we can construct 2 more matrices S_L and S_U in such a manner that $S_L * I >= L$ and $S_U * I <= U$. S_L is a lower triangular matrix and S_U is an upper triangular one. The second inequality can also be written as

$-S_U * I >= -U$. A typical example of such matrices is the identity matrix. The entire set $(S_L, -S_U, L, -U)$ is the *polyhedral representation* of the entire nested loop.

Considering a d-dimensional array A, two index instances i_1 and i_2 and two functions of i1 and i2, F and G respectively, then a loop carried dependency between i1 and i2 will exist if $A[F(i_1)]$ and $A[G(i_2)]$ reference the same memory position. This means $F(i_1)=G(i_2)$. The dependency problem then turns into a linear programming problem where $S_L * i_1 >= L$, $-S_U * i_1 >= -U$, $S_L * i_2>=L$, $-S_U * i_2 >=-U$ and $F(i_1) = G(i_2)$. The solution will show whether a dependency exists or not. Unfortunately, in this general case, the problem has been proven to be NP-Complete (it is equivalent to finding solutions to a system of Diophantine equations), thus a precise answer might take a long time to be computed. That is the reason that many of the dependency tests, such as the Banerjee, the Omega and the Range test exist. They can provide fast results under simplified conditions or special situations. A dependency test's reply can belong in one of these three outcomes: (i) A dependency exists, (ii) A dependency does not exist and (iii) Not sure. Tests that answer only (i) and (ii) are called *exact* tests otherwise they are called *inexact* tests.

Assuming a dependency between index instance $i=(i_1, i_2, ..., i_n)$ and $j=(j_1, j_2, ..., j_n)$ exists, then the vector $j - i = (j_1 - i_1, j_2 - i_2, ..., j_n - i_n)$ is the *dependency distance vector*. If the vector consists only of constants then it is called a static or uniform dependency vector. If the values are not constant then the vector which contains the signs of each subtraction sign $(j - i) = (\text{sign } (j_1 - i_1), \text{sign } (j_2 - i_2), ..., \text{sign}(j_n - i_n))$ is the direction dependency vector. For example, let us consider the loop in Figure 3.2.

```
for (i=0; i< 5; i++)
      for (j=0; j< 5; j++)
            A[ i ][ j ] = A[ i − 1 ][ j ] + A[ i ][ j − 1 ];
```

Figure 3.2. A Typical Example of a Perfectly Nested Loop in C with Two Loop Carried Dependencies.

In that example, first we compute the polyhedron representation. Since all bounds are fixed constants, we can see that $S_L = S_U = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, $I = \begin{bmatrix} i \\ j \end{bmatrix}$, $L=\begin{bmatrix} 0 \\ 0 \end{bmatrix}$ and

$U=\begin{bmatrix}4\\4\end{bmatrix}$ so $-U=\begin{bmatrix}-4\\-4\end{bmatrix}$. In the statement we can see that for any current iteration (i, j) the array A is referenced in two previous iterations, (i-1, j) and (i, j-1). This means that there exist two dependencies with vectors d1=((i)-(i-1),(j)-(j)) = (1,0) and d2=((i)-(i),(j)-(j-1))=(0,1). The set of all dependencies is the dependency vector $D=\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$ and the direction vector is the same since it carries the signs of the values of D.

3.2. Loop Transformations

3.2.1. First Pass Transformations

Before the actual transformations transpire and even before the dependency analysis of the code, parallelizing compilers usually perform a first pass of transformations. These transformations are mostly aimed at simplifying expressions in order to facilitate ease of analyzing array subscripts, loop bounds etc. Such transformations fall into the *idiom recognition* category (as they mostly search for certain expressions inside the code) and they include (but are not limited to):

Interprocedural Dependency Analysis. It is very probable that a computation might span across a multitude of different procedures inside the code of a program. This is the very essence of modular programming which allows for the creation of easy-to-read, well structured code. However due to the compiler's inability to know the side-effects of each procedure call (i.e. what effects a call might have to any other variable of the program), most procedure calls are left untouched by the automatic optimization tool which essentially means that the programmer is penalized for using procedure calls inside a loop. Interprocedural dependency analysis is the type of analysis that crosses the boundaries of procedures and analyzes side effects or trying to incorporate a procedure call into the automatic loop transformation. As has been mentioned earlier, one way to simplify a procedure call is to inline the entire procedure on the place of the call and then analyze the code as usual. This however might not be optimal.

False Dependency Elimination. Out of the four different types of dependencies, it has already been demonstrated that only one type is the true form of

dependency. The rest are false dependencies and they only obfuscate code. In order to simplify the code and be rid of anti-dependencies a compiler can either use data privatization or data expansion. An example of data privatization is given in Figure 3.3. In this example, assigning of any value to tmp seems like a critical section and that it can't be parallelized. However, tmp is first written and then read. It is a Write before Read type of dependency. By assigning the attribute PRIVATE to tmp the transformation makes sure that each instance of tmp will exist inside each thread's local storage and thusly the whole loop can be parallelized. Figure 3.4. displays the use of data expansion. Each different instance of tmp is assigned in a new temporary array which can be accessed at a later step for the actual computation. The PARALLEL directive in these cases means that the loop should be fully parallelized.

```
for (i=0; i<n; i++)                          PARALLEL for (i=0; i<n; i++)
{                                            {
        tmp=Code irrelevant to tmp;                  PRIVATE tmp=…;
        A[i]=tmp;                                    A[i]=tmp;
}                                            }
```

Figure 3.3. Using Data Privatization in Order to Simplify and Remove a False (Anti) Dependency.

```
for (i=0; i<n-1; i++)
        for (j=0; j<n-1; j++)
                A[i][j]=A[i+1][j] + A[i][j+1];
```
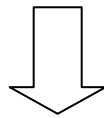
```
PARALLEL for (i=0; i<n-1; i++)
        PARALLEL for (j=0; j<n-1; j++)
                T[i][j]=A[i][j];
```

```
PARALLEL for (i=0; i<n-1; i++)
        PARALLEL for (j=0; j<n-1; j++)
                A[i][j]=T[i+1][j] + T[i][j+1];
```

Figure 3.4. Using Data Expansion in Order to Simplify and Remove a False (Anti) Dependency.

Symbolic analysis. Symbolic analysis is a general term. Most of the times, compilers who perform such an analysis monitor each variable and track its value range from statement to statement. This way, it is possible to be able to know all the ranges of all variables (including ranges of array subscripts as well as the values of the arrays themselves) and thusly reach some conclusions regarding the code in question. Certain loops that seem un-parallelizable might end up containing some parallelism and that is because certain indices might not end up overlapping or referencing the same memory location in order to create dependencies.

Induction Variable Elimination. An induction variable is one that its value is updated in each loop iteration in such a manner that it can be replaced by a closed mathematical formula. Figure 3.5. gives an example of an induction variable and its elimination from the code. "sum" is identified as an induction variable and is replaced by its closed mathematical formula (2 in the power of i) which at first creates an anti-dependency but one that is easily eliminated from the code in the final fully parallel loop.

```
sum=1;                                      PARALLEL for (i=1; i<=n; i++)
for (i=1; i<=n; i++)                        {
{                                                   sum= pow (2, i);
        sum=sum * 2;                                A[i] = sum;
        A[i] = sum;                         }
}
```

```
PARALLEL for (i=1; i<=n; i++)
        A[i]=pow (2, i);
```
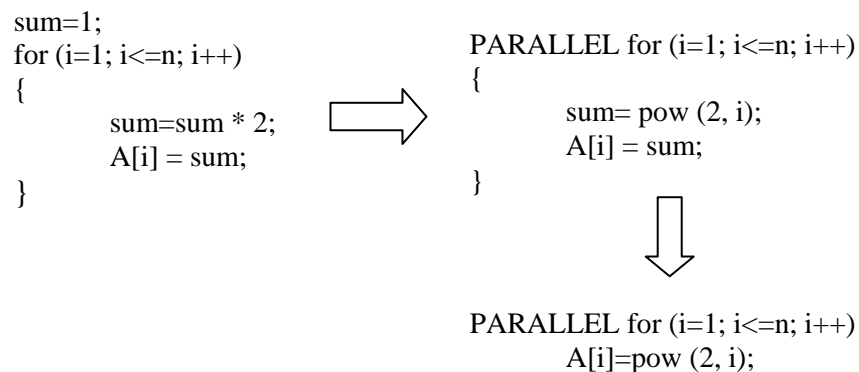
Figure 3.5. An Example of Induction Variable Elimination.

Loop Normalization. Most idiom recognition algorithms assume that any given loop index starts with the value of 0 and has a unit stride. If that is not the case for a loop then normalization transforms the loop in order to meet that requirement as is demonstrated in Figure 3.6.

```
for (i=2; i<2 * n; i+=2)                    for (i=0; i<n; i++)
        A[i]=…;                                     A[2*i+2]=…;
```
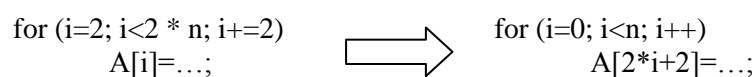
Figure 3.6. An Example of Loop Normalization.

Global Forward Substitution. By substituting all constant variables with the expressions they evaluate to, an automatic parallelizer can help make dependency analysis easier. Figure 3.7. shows an example of substitution.

```
ex=2*k+1;                                    for (i=0; i<n; i++)
for (i=0; i<n; i++)          ⟹                    A[i]=i+2*k+1;
        A[i]=i+ex;
```

Figure 3.7. An Example of Forward Substitution.

Loop Distribution. Loop Distribution is the technique where a single loop (probably nested, perfectly or not) is split into a series of different loops, each with the same iteration range as the original loop. Every one of the new loops carries a smaller part of the original loop's body as its own. This technique can be useful in improving cache usage and in the case of a multi-processor system where each processor can handle a single loop if they are independent from each other. Extra care must be taken to preserve the order of execution of dependent statements. Figures 3.8. and 3.9. give two different examples of Loop Distribution.

```
for (i=0; i<n; i++)                          for (i=0; i<n; i++)
{                                                    A[i]=C[i];
        A[i]=C[i];           ⟹
        B[i]=D[i];                           for (i=0; i<n; i++)
}                                                    B[i]=D[i];
```

Figure 3.8. An Example of Loop Distribution Which can Help Improve Cache
Performance.

```
for (i=0; i<n; i++)                          for (i=0; i<n; i++)
{                                                    A[i]=B[i];
        A[i]=B[i];
        for (j=0; j<n; j++)  ⟹              for (i=0; i<n; i++)
                C[i][j]=D[i][j];                     for (j=0; j<n; j++)
}                                                            C[i][j]=D[i][j];
```

Figure 3.9. Another Example of Loop Distribution Where an Imperfectly Nested
Loop is Split Into two Perfectly Nested Ones.

Loop Fusion. Loop Fusion is exactly the opposite act of Loop Distribution. It can be useful in cases where the overhead of a loop is significant and as such, it can lead to reduced overhead and better run-time speed overall. Such a case is when the loop is in fact, some parallel construct which requires time to set up all the threads necessary in order to complete execution. Fusion is possible when the legality of the dependencies is preserved and when the index ranges match (although if they don't, maybe some type of normalization might be possible to be applied to match the other). Figure 3.10 gives an example of Loop Fusion. In that example we can see that both loops can execute in parallel and so the new loop has a potentially smaller overhead than the initial two loops.

<pre>
PARALLEL for (i=0; i<n; i++)   PARALLEL for (i=0; i<n; i++)
        A[i]=B[i];   {
                                                  A[i]=B[i];
PARALLEL for (i=0; i<n; i++)   C[i]=D[i];
        C[i]=D[i];   }
</pre>
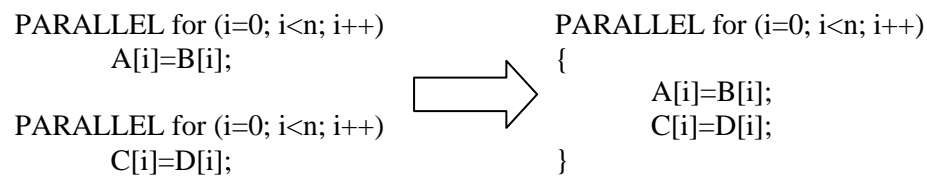
Figure 3.10. An Example of Loop Fusion. Two Parallel Loops are Fused Together with the Aim to Reduce Overhead.

Reductions. A reduction variable is one that exists in the form of multiple copies in a series of threads' local storages and the need exists to reduce them all into one final and single variable. Summing up an array is a usual example of such an action. Figure 3.11 shows an example of a reduction variable and how it can be transformed to exploit some parallelism. In this example, PE is the number of processing elements we can use to speed up the reduction and P is the ceiling of the result of the division of the total number of array elements n divided by PE. Essentially, P is the total number of partial sums we will calculate and then reduce. The first loop initializes the partial sums s[i] in parallel, then the second loop sums up the P different parts of n into each s[i] and the final loop calculates the final value of sum by adding up all the partial sums s[i]. If the granularity is coarse enough then the speedup of the parallelism is higher than any overheads that might exist.

$$P=\left\lceil\frac{n}{PE}\right\rceil;$$
PARALLEL for (i=0; i<P; i++)
    s[i] = 0;

sum=0;
for (i=0; i<n; i++)
    sum += A[i];

PARALLEL for (i=0; i<P; i++)
    For (j=i*P; j<(i+1)*P;j++)
        s[i]=s[i]+A[i];

sum=0;
For (i=0; i<P; i++)
    sum+=s[i];
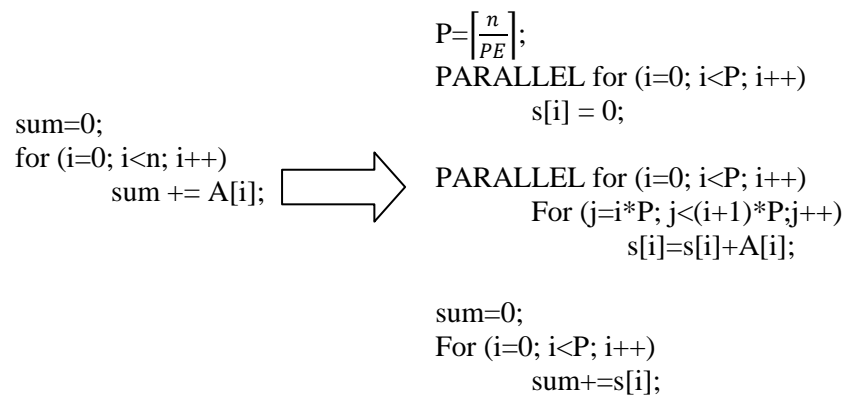
Figure 3.11. An Example of Reduction. The Summation of A into the Scalar "sum" is Partially Parallelized.

3.2.2. Unimodular Matrices

Choosing to represent the polyhedron and the dependencies of a loop via the use of matrices offers a significantly helpful tool when it comes to transformations: the *Unimodular matrices*. A Unimodular matrix is nothing more than an integer matrix whose determinant equals to 1 or -1. A loop transformation can be encoded inside such a matrix and then that matrix can be multiplied with the polyhedron and the dependency vector to produce a transformed loop. Unimodular matrices contain integer elements so that the transformed polytope will also contain integer values and its unimodularity guarantees a one-to-one mapping with a stride of one.

With the help of these matrices we can apply a series of transformations by multiplying their respective Unimodular matrices in the reverse order of the transformations' application. This way, *compound transformations* are created. At this point it is important to note that not all transformations are legal. In order for a transformation to be accepted for use, the new dependence vector D' must contain lexicographically positive dependences. In general, a tuple (a, b, c, ….) is lexicographically positive when the first non-zero element in the tuple is a positive number. Lexicographic positivity is a strong condition for all transformations otherwise anti-dependences will be created. There are some cases when the existence of anti-dependences might not matter but if a cyclical dependency appears then it is impossible for the compiler to produce any meaningful code.

3.2.3. Prime Loop Transformations

Once all idiom recognitions have transpired then an automatic compiler can proceed to perform the main or *Prime* loop transformations. In contrast to the first pass transformations, Primes do not seek to simplify some expression or find inter procedural dependences but, according to the current needs they usually aim to increase code efficiency (both in a parallel code but on occasion on a sequential one as well) and apply much more drastic alterations to a given loop. The most common of these transformations are listed below and elaborated upon:

Loop Tiling. Loop Tiling, Loop Blocking, or Strip-mining is a loop transformation technique aimed at increasing the efficiency of any sequential loop. The main idea is that any given loop can be transformed to an equal one but where the entire index space is partitioned in smaller tiles (of a fixed size each on every dimension) and then execution takes place on a per tile basis. Figures 3.12 and 3.13 give an example of a loop before and after tiling with its accompanying graphic illustration. It is important to note here that there must be no dependency conflicts with the change in the execution schedule so the ordering imposed by dependences is still preserved in the tiled version of the original loop.
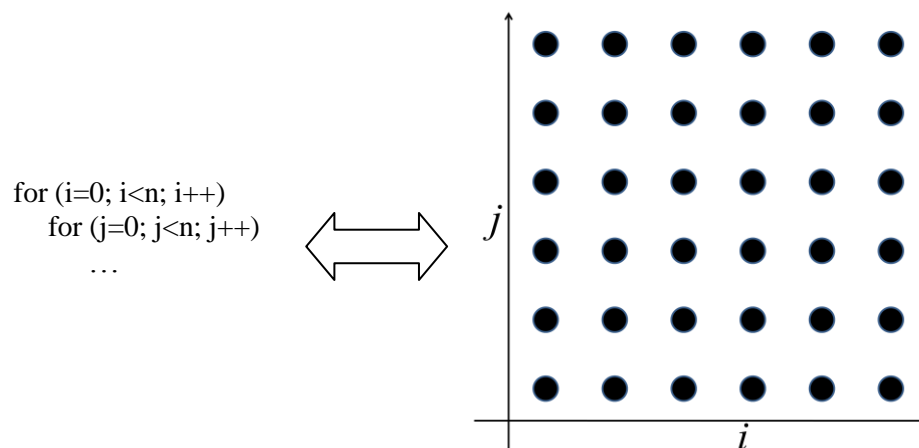
Figure 3.12. A Perfectly Nested Loop with Nesting Level of 2 and its Graphical Representation in the Two-Dimensional Space.

```
for (i=0; i<n; i+=3)
    for (j=0; j<n; j+=3)
        for (i'=i; i'<min(i+3, n); i'++)
            for (j'=j; j'<min(j+3, n); j'++)
                ....
```
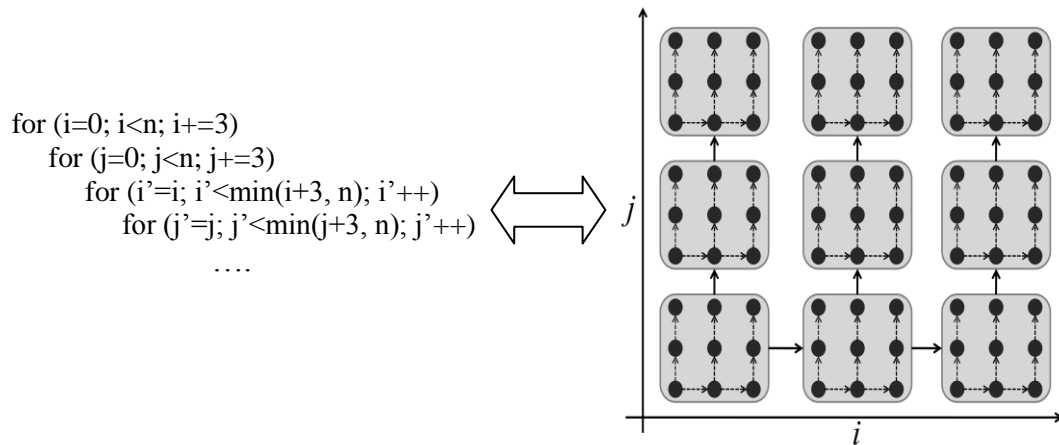
Figure 3.13. The Loop of Figure 3.12 and its Graphic Representation After a Tiling Transformation. A Stride of 3 was Used in Each Dimension.

The increase in efficiency is mostly accomplished by exploiting data locality in the CPU's cache. Tiling can also be a first step in various other transformations where each tile serves as the basic parallelization unit (in other words, coarse grain granularity can be achieved by first tiling close indices together and execute them in some sequential manner while each tile can execute independently from the others either in different CPUs or in different threads).

Loop Interchange. As the name suggests, the technique of loop interchange exchanges the levels of two iteration variables in a nested loop. A dependence of (a, b) becomes (b, a) which means that extra care must be given in order to safeguard the legality of the whole transformation. If b in that case is a negative number, then by performing interchange, the dependence is no longer lexicographically positive and the legality is forfeit. The Unimodular matrix for this operation is demonstrated in Figure 3.14 while Figure 3.15 gives an example of a nested loop before and after loop interchange. Loop interchange can generally improve efficiency by exploiting locality of reference and cache usage. It can also enhance inner or outer loop parallelization or enable vectorization. However it may also adversely affect performance if not enough care is given by hindering cache usage altogether. Overall, the effectiveness of interchange relies heavily on the underlying cache model the system's hardware architecture is using. It is important to state here that if the loop bounds of the original loop are not simple, then computing the new loop bounds is generally non-trivial.

$$\begin{bmatrix} i' \\ j' \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \Leftrightarrow \begin{matrix} i' = j \\ j' = i \end{matrix}$$

Figure 3.14. The Unimodular Transformation of Loop Interchange.

```
for (i=0; i<n; i++)              for (j=0; j<n; j++)
    for (j=0; j<n; j++)              for (i=0; i<n; i++)
        A[i][j]=i+j;                     A[i][j]=i+j;
```

Figure 3.14. A Nested Loop Before and After Loop Interchange.

Loop Permutation. Loop Permutation is a more general method of Loop interchange. For any perfectly nested loop of nesting level of n, then pairs of loops can swap their place in the nesting. Dependences obey that swapping as well. For example a dependence of dimensionality 3, (a, b, c) with a permutation of $\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix}$, becomes (b, c, a). The necessary Unimodular matrix for this transformation is constructed by swapping the corresponding rows of the identity matrix I (of a suitable dimensionality), as is demonstrated by Figure 3.15. Figure 3.16 displays the application of such a matrix on an index set. As this technique is a generalization of loop interchange then the automatic compiler needs to be aware of and avoid the same pitfalls as with loop interchange: A transformed dependency must never become lexicographically negative so again extra care is needed when applying this technique. The only way the compiler can be sure of any permutation's legality is to perform an analysis on all dependences. If all distances are positive then any permutation is legal.

$$\begin{bmatrix} 1 & 0 & 0 & & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & & 0 \\ \vdots & & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 1 & 0 & & 0 \\ 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & & 1 \\ \vdots & & & \ddots & \vdots \\ 0 & 0 & 1 & \cdots & 0 \end{bmatrix}$$

Figure 3.15. Creating a Permutation Unimodular Matrix by Swapping the Rows of the Original Identity Matrix.

$$
\begin{bmatrix} i'_1 \\ i'_2 \\ i'_3 \\ \vdots \\ i'_n \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & & 0 \\ 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & & 1 \\ & \vdots & & \ddots & \vdots \\ 0 & 0 & 1 & \cdots & 0 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \\ i_3 \\ \vdots \\ i_n \end{bmatrix} \Rightarrow \begin{array}{l} i'_1 = i_2 \\ i_2{}' = i_1 \\ i_3{}' = i_n \\ \vdots \\ i_n{}' = i_3 \end{array}
$$

Figure 3.16. Applying the Constructed Unimodular Matrix from Figure 3.15 to an Index Set.

Loop Reversal. Loop Reversal is a technique which reverses the bounds of a loop. For example, a simple loop with bounds L and U will be transformed to one with bounds −U and −L respectively. The corresponding dependence of that loop automatically switches sign. In a nested loop this effectively means that the loop which corresponds to the first positive distance in any dependence cannot be reversed otherwise the dependence will no longer be lexicographically positive. In a loop of nesting level 3, a dependence (a, b, c) becomes (a, -b, -c) after such a transformation is applied to levels 2 and 3. Loop reversal rarely possesses any inherent ability to increase code efficiency however it can help eliminate dependences and thusly pave the way for other optimizations.

Loop Skewing. Loop skewing is a technique where a dependency (a, b) is transformed into a form of (a, f*b + c) where f is the skew factor. The same skew factor is applied on the shape of the polytope representation and changes it into a new shape with a different representation (a skewed version of the original polytope). The fact that dependences retain their lexicographic positivity after such a transformation means that skewing is always safe to apply. In fact such a transformation is always possible to be discovered. Skewing is a very important transformation as it has the capability to expose parallelism in the innermost loop of a perfect nesting. Figure 3.17 displays a code example which denies any kind of parallelism at first sight (or even interchange for that matter).

```
for (i=1; i<n; i++)
    for (j=1; j<n; j++)
        A[i][j]=A[i-1][j]+A[i][j-1];
```

Figure 3.17. A Code Example where Skewing can Expose Hidden Parallelism.

It is straightforward to calculate the dependences in that code snippet: (1,0) and (0,1). Figure 3.18 shows a graphical representation of the loop's polytope and the corresponding dependences.
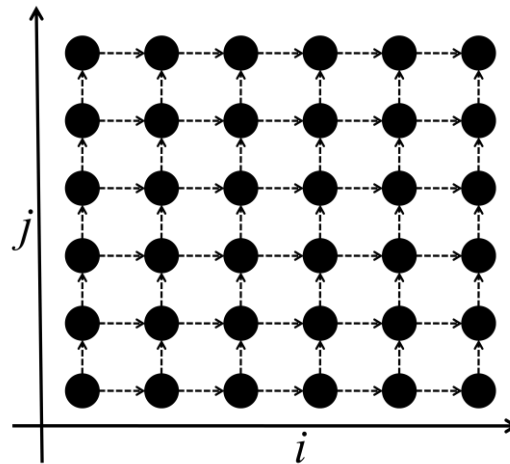
Figure 3.18. The Graphical Representation of the Loop and the Loop Carried Dependences it contains.

There are many different Unimodular matrices that can describe various types of skewing, however a typical one is described in Figure 3.19.

$$\begin{bmatrix} 1 & f_1 & f_2 & & f_n \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & & 0 \\ & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

Figure 3.19. A Typical Skewing Unimodular Matrix. f_1, f_2,…, f_n are the Skew Factors.

In our example the Unimodular matrix becomes $\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$ (skewing factor of one). If we apply this transformation to the index set, we get the new indices of the transformed loop: $\begin{bmatrix} i' \\ j' \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} i+j \\ j \end{bmatrix}$. The dependences are also transformed by the same matrix: $\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$. The new dependence vector has been transformed to the set D' = {(1,0), (1,1)}. Generally speaking,

calculating the loop bounds of a skewing operation is a non-trivial and difficult endeavor altogether, however in this example it is rather simple and straightforward.

We know that $i' = i + j$ (1) and that $j' = j$ (2). We also know that $1 \leq i \leq n - 1$ (3) and that $1 \leq j \leq n - 1$ (4). By adding (3) and (4) we get that $2 \leq i + j \leq 2n - 2 \Leftrightarrow 2 \leq i' \leq 2n - 2$ (5). Now we use (1) and (2) and solve for i and j: $j = j'$ (6) and $i = i' - j'$ (7). By combining (3) and (7) we can see that $1 \leq i' - j' \leq n - 1$ (8). This double inequality can be split into two separate ones: $1 \leq i' - j'$ (9) and $i' - j' \leq n - 1$ (10). Out of (9) we get that $j' \leq i' - 1$ (11) and out of (10) we get that $j' \geq i' - n + 1$ (12). Since $j' = j$ and by using (4) we learn that $j' \geq 1$ (13) and $j' \leq n - 1$ (14). From (12) and (13) we get that $j' \geq 1$ and $j' \geq i' - n + 1$. Since we need j' to always have valid values, then $j' \geq \max(1, i' - n + 1)$ (15). By combining (11) and (14) we know that $j' \leq i' - 1$ and that $j' \leq n - 1$. Again according to the same principle, we reach the conclusion that $j' \leq \min(i' - 1, n - 1)$ (16). Finally, by using (5), (15) and (16) we know the new loop bounds and can create the new transformed loop which is displayed in Figure 3.20. Figure 3.21 demonstrates the skewed result in the graphic representation of the polytope.

Observation of the skewed result makes the hidden parallelism obvious. For every different i' of the loop, all the j' belonging to that iteration of i' are independent from one another and so they can execute in parallel. This technique can also be applied after tiling in order to offer a more coarse grain form of parallelism. In the case where tiling has already been applied, then each position in the iteration space corresponds to a single tile instead of a single iteration instance of the loop. Skewing is a very important tool in the arsenal of a parallelizing compiler as it offers varying levels of granularity of parallelism that lies hidden in the innermost loop, and as such it is the most important part of the wavefront method.

```
for (i' = 2; i' <= 2*n-2; i'++)
    for (j' = max(1,i'-n+1); j' <= min(i'-1, n-1); j'++)
    {
        i=i'-j';
        j=j';
        A[i][j]=A[i-1][j]+A[i][j-1];
    }
```

Figure 3.20. The Skewed Result from the Original Loop of Figure 3.17 when the Matrix of Figure 3.19 was Applied on it.
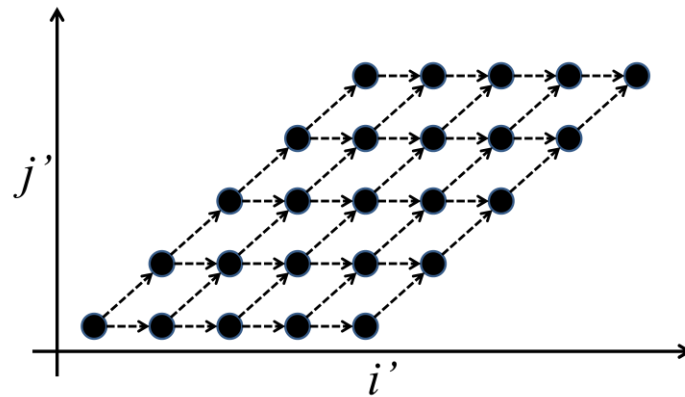
Figure 3.21. The Polytope Representation of the Skewed Loop Presented in Figure 3.20. The Inner Level Parallelism per Iteration of i' is Obvious.

The Wavefront. The wavefront method is a compound transformation which encompasses loop skewing, loop reversal and loop interchange / permutation. The main purpose behind the wavefront model is to find a series of hyperplanes, each covering a subset of the original polytope, with the property that all indices on a certain hyperplane are independent between them and can thusly be run in parallel. By visualizing the wavefront method on the code of Figure 3.17 (and after application of the skewing transformation discussed before) we can see (in Figure 3.22) that essentially the wavefront method creates an imaginary wave which moves through the data.
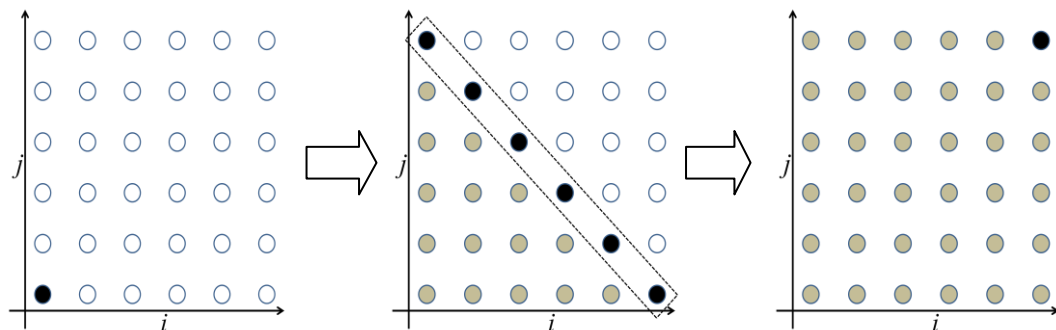
Figure 3.22. From Left to Right the Wavefront (Black Dashed Rectangle) Moves Through the Computation data. Grayed Points Indicate Already Processed Index Instances.

All index points on the front are the ones that can be executed in parallel in that iteration. This is the reason that this general methodology is called a wavefront and essentially, it is the hyperplane method originally proposed by Lamport where each hyperplane is driving the front by being executed sequentially and each front being executed in parallel. Calculating a proper compound unimodular matrix in order to have an efficient wavefront is a difficult task and there is no single direct algorithm for it. Most automatic parallelizing compilers resort to heuristic methods in order to pick the best transformation out of all the possible wavefronts that exist for any given problem. Finally, it is worth mentioning that the wavefront method (as well as the skewing one) require a uniform dependence vector in order to work properly.

CHAPTER 4. SVP

4.1. Introduction and Prerequisites

With Moore's law (an empirical observation made by Gordon E. Moore which states that the number of transistors in integrated circuits doubles every 18 months) still in effect, it is becoming increasingly clear that the only way to push forward with improving efficiency and speed in systems is via multi-core architectures. multi-core processors (multiple cores on chip) have the ability to utilize the ever increasing on-chip resources while simultaneously handling the increase of complexity of the circuitry.

However, there are some issues that need handling. A multiprocessor must define a model of parallelism that is similar to the sequential model that users have been accustomed to. In addition, binary compatibility across a range of different generations of processor implementations is very desirable. In the spirit of the sequential model, the Multiprocessor system should also be deterministic (which means that given a certain input it will always produce the same output) and ideally it can provide deadlock avoidance mechanisms. When it comes to the aspect of parallelism, a Multiprocessor needs to be able to capture and exploit maximal concurrency while at the same time gracefully degrade when it runs low on resources. As such, automatic resource allocation is an important prerequisite since hand-mapping applications onto available resources is not feasible (neither sensible).

4.2. The SVP Processor and Model

The SVP (Self-Adaptive Virtual Processor) model is a system designed and implemented to cover for all the afore-mentioned prerequisites. By design, it is a general concurrent processor model which bases its abstract execution model on a hierarchy of "*microthreads*". A microthread is an entity very similar to a regular thread (i.e. it is a sequence of sequentially executed statements that can run in parallel with other threads or the main application that spawned them) but with the added property of blocking its execution when there are no data available to them for calculations [16]. This essentially places the SVP into a more generalized SPMD category since its API exports directives for synchronization. SVP is designed to be deterministic and approaches parallelism in a highly dynamic manner through its ability to be self-adaptive. It is also meant to target the entire range of applications instead of just a few specific ones. The self-adaptiveness of the model is realized by three distinct properties: (i) It can capture the concurrency of an application in its entirety, (ii) It captures and enforces locality of communications between threads and (iii) keeps everything as dynamic as possible [38].

A novel property of the SVP is that it can be implemented in its entirety (including the run-time environment) in a processor's Instruction Set Architecture (ISA) and thus it can be considered as an Operating System (OS) on chip [22]. An ISA implementation offers the advantage of backwards compatibility with any pre existing sequential code (which is not affected at all) and also provides the ability for any SVP program to revert back to a sequential form of execution if such a need arises. A series of such cores (SVP cores) forms a *Microgrid*. A Microgrid offers binary compatibility over any cluster of such processors, is inherently scalable when it comes to both area utilization and performance and can support a great degree of parallelism through the use of a large number of Microthreads and high memory tolerance. The OS deals with managing any dynamically created content through *delegation*. Delegation refers to the process where a computation can be mapped to any part of the microgrid remotely during run time.

The abstract execution model of the SVP is quite general. Applications (and by extension, developers) need not concern themselves with any kind of mapping of threads or their scheduling, as the run-time system dynamically allocates resources to

threads as needed and the scheduling is achieved through synchronizing communication: There exist two types of synchronizing channels, the shared ones and the global ones. The existence of these channels inside the code decides whether threads will run sequentially or in parallel. Proper use of these channels (i.e. the channels are read from and written to when they are supposed to) guarantees a deadlock-free execution of the application.

Moreover, the execution model presents a recursive / hierarchical structure of parallelism. Microthreads do not exist autonomously but they are always part of a family. This makes the family the basic unit of work in the SVP model. Families can be of any arbitrary size (even infinite) and individual threads inside those families are created only when there are available resources. When there is a lack of resources, the model falls back to a sequential mode where the family executes entirely in its parent thread's context.

A thread has the ability to create another family and thusly a hierarchy of families is formed. The synchronizing channels exist solely inside a family between its threads. A smaller form of communication exists between a parent thread and its subservient family. The created family can receive data (in the beginning) and return data (after termination) to its creator but this is the only form of communication allowed between threads in the entire concurrency tree, at a user's level of perspective at least. Any other form would be at the very least inefficient and normally avoided.

The global synchronizing channel is immediately accessible by all the threads of a family (each thread can decouple data from any global channel) and offers a set of read-only data from the creator of the family. The shared channel works differently by adopting a data-flow behavior: Each thread of the family can read (decouple into shared variables) from that channel once and write back a value to it (couple) also once. If a thread finds no data inside the channel at the moment it tries to read it, then it blocks its execution until data is available.

Threads identify themselves inside the family by the use of an index and once a thread has written a value to the shared channel, then that value will be instantly accessible by the thread with the next index value. The original value in the chain is designated by the creator / parent thread, while its final value is accessible by the creator thread. Whenever a sequential form of execution is needed between the

threads of a single family, then a shared channel can also be applied to enforce an order of execution. In this case the value of the data moving from thread to thread is completely irrelevant as long as there is some data moving. Figure 4.1 demonstrates a typical SVP family of microthreads during its execution: Each time a thread writes a value on a shared channel, the next thread (which blocked on reading that channel) can resume computation. Local computations can take place in parallel outside of the reading / writing of the shared variable. The global channel is visibly available to all the family threads.
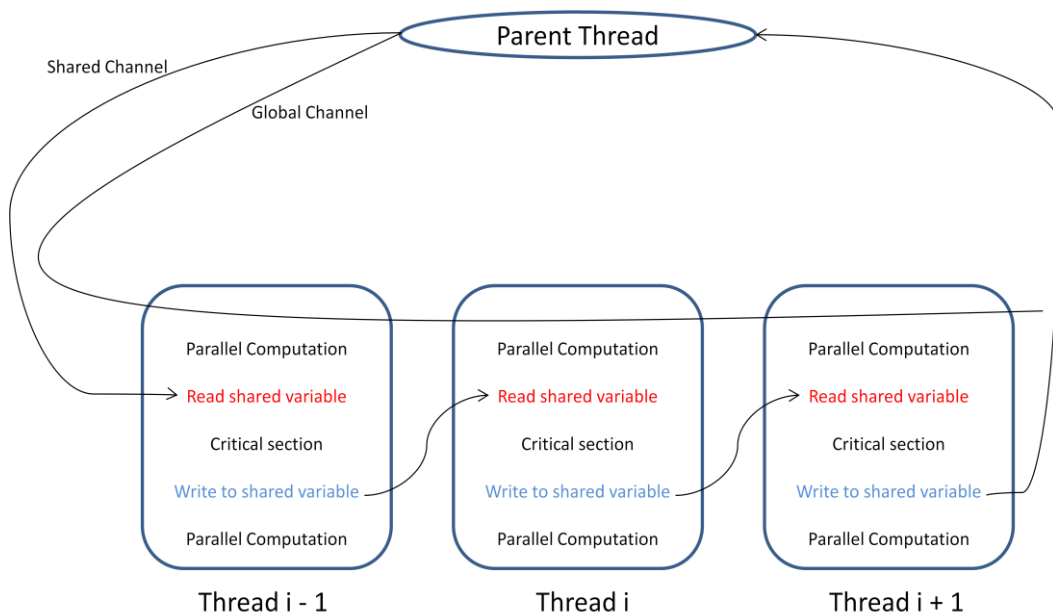
Figure 4.1. An SVP Family of Microthreads. The Global Channel is Available to all Threads While the Shared one Creates a Data-chain from One Thread to the Next.

In addition to those two types of channels, a global asynchronous memory (in the form of a flat address space) exists which is accessible by all threads. At any given time, each thread "sees" a view of a particular memory section which will remain consistent so long as no other thread writes to that particular place. Once the family finishes its execution, then all such "views" are shared and a final view of the entire subsection involved in the execution is considered to be at a stable, synchronized state. Specifically, the consistency model does not guarantee that a thread will see any changes performed by an unrelated thread at any given time [40], [17], [39]. Figure

4.2. demonstrates an SVP hierarchy with the bulk asynchronous memory available to all threads.

The global asynchronous memory coupled with the synchronizing channels (which offer parent-child and intra-family communication) are sufficient to capture all kinds of dependencies inside a program, since the synchronizing channel can impose the same ordering as a loop carried dependency and the rest dependency types aren't real dependencies. It is obvious though that the only way for flow dependences to be expressed in the SVP model in any sensible manner is through the use of dataflow semantics and the various communication channels. This means that legacy code cannot be executed as-is in a parallel manner. Certain types of transformations are required in order for dependencies (most importantly loop-carried ones) to be mapped into threads and families. In summary, loops (parallel or sequential ones) and function calls must be implemented as families where the blocking nature of the threads will offer the proper ordering of execution.
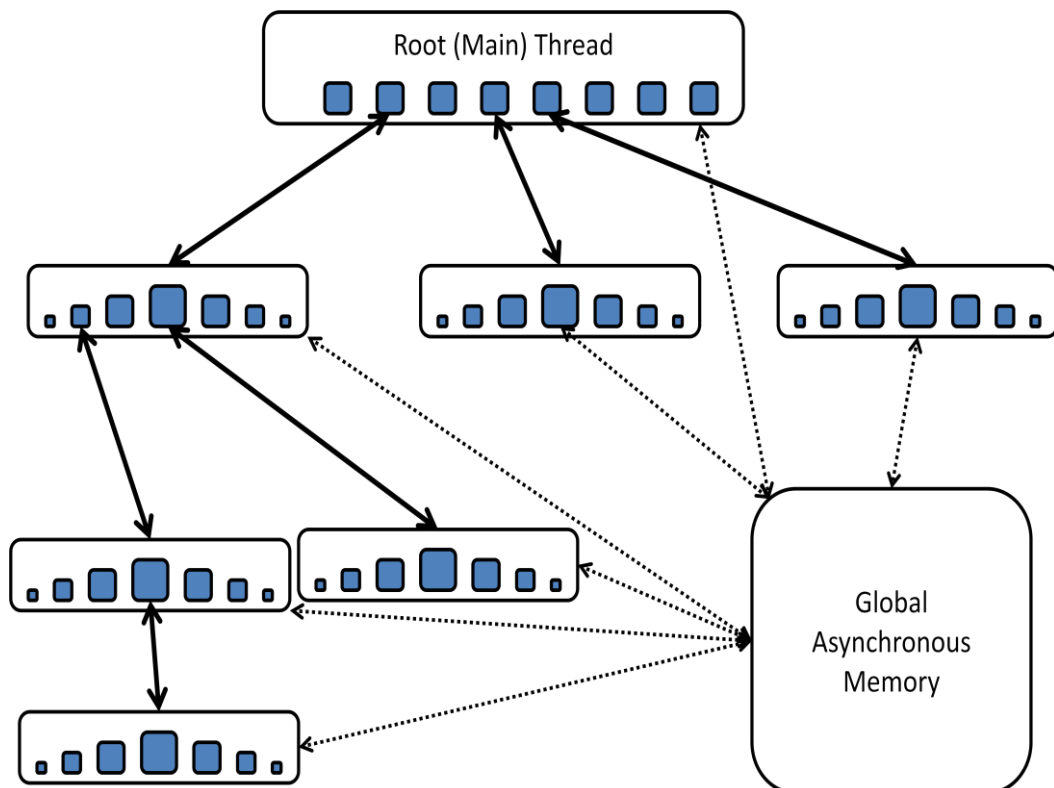
Figure 4.2. An SVP Hierarchy with the Accompanying Asynchronous Memory.

A typical example where the multiple types of communications channels are of use is the matrix multiplication. The code fragment that performs multiplication between two two-dimensional matrices A and B is displayed on Figure 4.3. The result is stored in a similar matrix C. For simplicity reasons we assume the matrix dimensions are n x n. It is clear from that code sample that the two outermost loops (i and j) that compute the elements of C are independent from the rest. The only loop carried dependency appears inside the innermost loop (k) where sum is updated once per loop iteration. In the SVP model this would translate to three families: i, j and k. The threads in family i are all independent between them (hence there is no need for a shared channel) and each thread invokes family j, where again its threads can be executed in a concurrent manner.

```
for (i=0; i<n; i++)
        for (j=0; j<n; j++) {
            sum=0;
            for (k=0; k<n; k++)
                    sum += A[i][k] * B[k][j];
            C[i][j]=sum;
        }
```

Figure 4.3. A Typical Code Fragment which Calculates the Product of two n x n Matrices.

Each of these threads (in the j families) initializes a thread-local variable sum with the value 0, and then invokes (spawns or creates) family k. This is where most of the computation takes place and indeed we can see that at first glance it is not possible to increase efficiency more. However, each of the threads in the k family first computes the product A[i][k]*B[k][j] and then updates the variable sum (which carries the total sum and hence is represented as a shared channel). Since each of those products is independent from the other threads, then it is prudent to have all the threads of the family k compute that product concurrently before beginning to update the total sum. In summary, each "k" thread performs the following steps: (i) calculate the result of the product A[i][k]*B[k][j] in parallel and store it in a temporary variable, (ii) perform a read on the shared channel "sum" (and block if it's not available), (iii) Add the temporary variable to the sum variable, (iv) write the value of the sum variable back to the "sum" shared channel, (v) terminate.

The hierarchy that such a computation creates is demonstrated in Figure 4.4: A tree of execution with three levels is created. Since the threads in level 1 (Family i) execute concurrently, this means that the n x n threads of level 2 (Family j) will execute concurrently and thusly each of the n x n elements of matrix C will compute independently from the rest. The k families display the dataflow created by the sum variable travelling through the shared channel. It is initialized at each parent locally, it traverses through all the threads one by one and then returns back to the originator.
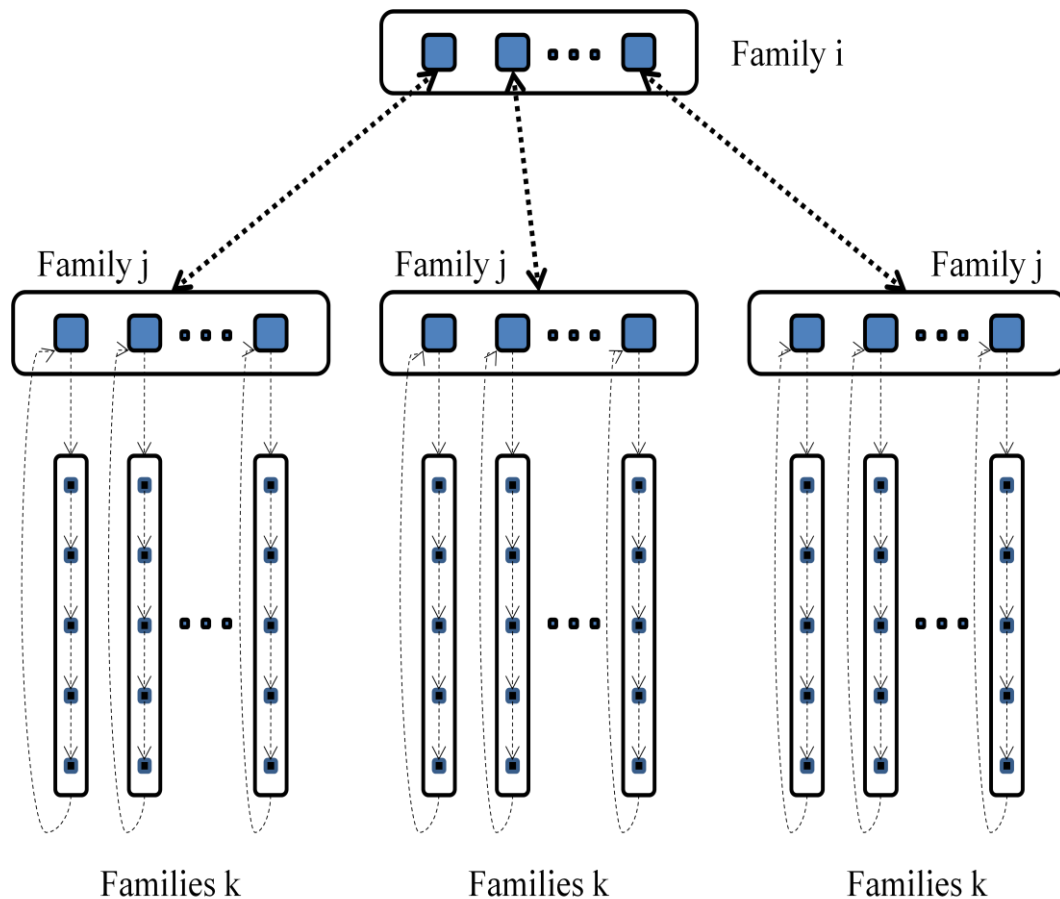
Figure 4.4. The Execution Hierarchy Created for the Concurrent Matrix Multiplication. Single-pointed Arrows Indicate Dataflow Direction.

4.3. The SL Programming Language

It was made abundantly clear that a programming language which could support the API exported by the SVP was needed. Initially a series of extensions were designed and added to the C language (a well-known language used world-wide) and

the μTC [37] language (micro-threaded C) was born. As development proceeded, certain problems emerged that led to the creation of the SL language. While μTC was implemented by modifying the *gcc* compiler, SL used a series of macros to help pass the code through the original unaltered gcc compiler and used post processing to provide the necessary functionality and optimization. It provides mechanisms for the bulk creation and synchronization of threads, the passing of variables and values through the global and shared channels and more. Some key macros and their explanation follows:

- **sl_def**(){code} **sl_enddef**. sl_def defines a thread with a programmer defined name and a defined return (usually void). A series of arguments is listed in the parentheses. Arguments are passed by value. sl_enddef denotes the end of the thread definition. Similar to the classic join for threads, sl_sync() will halt execution of the parent thread that created a family and wait till that family terminates to continue execution.

- **sl_create**(). It creates a family of threads whose index' starting value, ending value and step will be defined inside the argument list of sl_create.

- **sl_sync**(). The sl_sync macro causes the invoking thread to pause and wait until the created family has finished computation and returned control to the parent.

- **sl_index(variable_name)**. A macro that can be called inside a thread function code. It stores the index of the current thread to the variable designated by variable_name.

A more detailed description of the SL language can be found in the Appendix at the end of the thesis.

```
sl_def(fib, void, sl_shparm(int, _a), sl_shparm(int, _b), sl_shparm(int, _c))
{
        int a=sl_getp(_a);
        int b=sl_getp(_b);
        int c=sl_getp(_c);

        c=a+b;
        a=b;
        b=c;

        sl_setp(_a, a);
        sl_setp(_b, b);
        sl_setp(_c, c);
}
sl_enddef
```

```
sl_def(t_main, void)
{
        int a=0;
        int b=1;
        int n=5;
        int c;

        sl_create(,,2,n+1,1,,,sl_sharg(int, _a, a), sl_sharg(int, _b, b),
                 sl_sharg(int, _c, c));
        sl_sync();
        c=sl_geta(_c);

        printf("%d\n",c);
}
sl_enddef
```

Figure 4.5. Calculating the n_{th} Term of the Fibonacci Sequence. After the Thread's Termination, Reading the Shared Channel c Provides the Final Result.

By combining SL directives and standard C code, it is easy to create SVP applications that exploit concurrency. For example, consider the code that computes the n_{th} number of a Fibonacci sequence. For simplicity we assume than n is greater or equal to 2. Using SL over SVP this code would look like the one in Figure 4.5 which demonstrates the thread definition and invocation.

```
#include<stdio.h>

typedef int[10] type1;
typedef type1[10] type2;

sl_def(family_k, void, sl_glparm(type2, _a), sl_glparm(type2, _b), sl_shparm(int, _sum),
        sl_glparm(int, _i), sl_glparm(int, _j))
{
        sl_index(k);
        type2 a=sl_getp(_a); int ype2 b=sl_getp(_b);
        int   i=sl_getp(_i); int   j=sl_getp(_j);
        int tmp=a[i][k]*b[k][j]; int sum=sl_getp(_sum);
        sum+=tmp;
        sl_setp(_sum, sum);
}
sl_enddef


sl_def(family_j, void, sl_glparm(type2,_a), sl_glparm(type2,_b), sl_glparm(int, _i),
        sl_glparm(type2, _c))
{
        sl_index(j);

        type2 a=sl_getp(_a);
        type2 b=sl_getp(_b);
        type2 c=sl_getp(_c);
        int i=sl_getp(_i);

        int sum=0;
        sl_create(,,0,10,1,,,family_k, sl_glarg(type2, _a, a), sl_glarg(type2, _b, b),
             sl_sharg(int, _sum, sum), sl_glarg(int, _i, i), sl_glarg(int, _j, j));
        sl_sync();
        sum=sl_geta(_sum);
        c[i][j]=sum;
}
sl_enddef
```

```
sl_def(family_i, void, sl_glparm(type2, _a), sl_glparm(type2, _b), sl_glparm(type2, _c))
{
        Sl_index(i);
        type2 a=sl_getp(_a);
        type2 b=sl_getp(_b);
        type2 c=sl_getp(_c);

        sl_create(,,0,10,1,,,family_j, sl_glarg(type2, _a, a), sl_glarg(type2, _b, b),
            sl_glarg(int, _i, i), sl_glarg(type2, _c, c));
        sl_sync();
}
sl_enddef
sl_def(t_main, void)
{
        Type2 a, b, c;
        sl_create(,,0,10,1,,,family_i, sl_glarg(type2, _a, a), sl_glarg(type2, _b, b),
            sl_glarg(type2, _c, c));
        sl_sync();
}
sl_enddef
```

Figure 4.5. An Application which Concurrently Multiplies two Matrices a, b (10x10 size) and Stores the Result in the c Matrix.

Another more complex example is the matrix multiplication one, already described in Figures 4.3 and 4.4. The SL / SVP implementation is illustrated in Figure 4.6.

4.4. The Toolchain

The SVP's toolchain is simple and efficient. The main component is the SL compiler which takes as input a program written in the SL language and produces a binary output ready to be executed by an SVP-compatible multicomputer system. For the convenience of the developer, the compiler may output a binary file that is essentially sequential. This option exists so that the programmer can check whether the code works properly in a sequential manner before proceeding into the actual parallel form. In the case of a fully parallel code, a simulator system is also provided. That system is an environment capable of simulating any type and size of microgrid with the OS-on-chip attached. The simulator can be used both to debug code and to evaluate it. Once the simulation completes, the programmer receives a number of helpful metrics about the application such as total master CPU cycles and so on. Schematically the Toolchain can be visualized by Figure 4.6.

Since the SL language is an intermediate level between high level and machine level, it is not expected by a user to code in SL (although that is perfectly

acceptable and normal). Instead the Toolchain is augmented with two more tools: An automatic compiler which transforms sequential C code to SL (The C2μTC/SL presented in this thesis) as well as an automatic compiler which transforms SaC (Single Assignment C) [73] to SL. These two compilers allow legacy code in C and data parallel code in SaC to be automatically parallelized. The main idea behind the toolchain is that ideas can be expressed in a high level language such as SaC or a (rather) structured C code and then see them run in a many-core environment. "Communication" in the toolchain takes part completely via the use of files. Each program takes an input and generates an output which in turn is used as the input of the next program. The augmented Toolchain is depicted in Figure 4.7. More information on the SVP model can be found in [74] and [75].
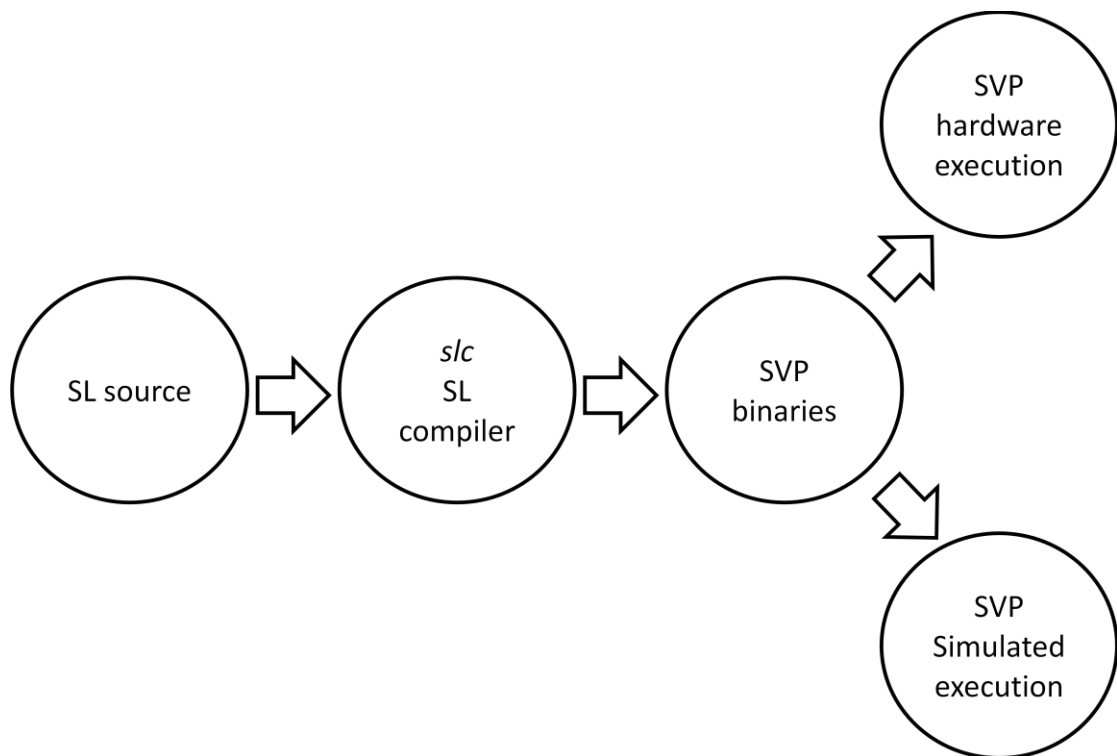
Figure 4.6. The Typical SL/SVP Toolchain.
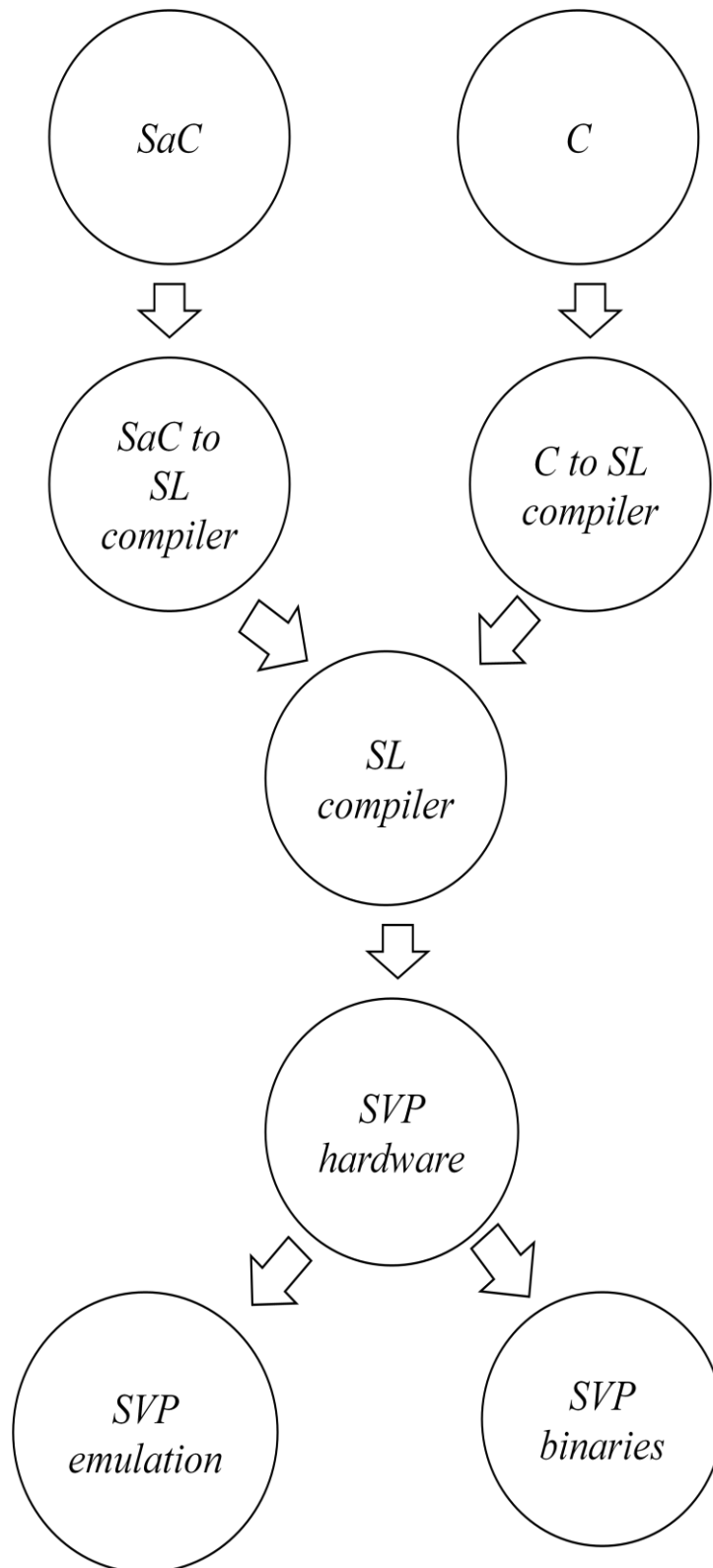
Figure 4.7. The Augmented SVP Toolchain.

CHAPTER 5. THE C2μTC/SL COMPILER

5.1. Introduction

5.2. One-Dimensional Loops

5.3. Multi-Dimensional Loops

5.4. From C to SL

5.1. Introduction

As it was stated in the previous chapter, SVP requires a different way of thinking when describing parallelism to the system through the use of SL. This also means that mapping loops created in a traditional sequential language (like C) onto SL automatically requires a new compiler. For that reason, C2μTC/SL was created. It is a source-to-source compiler which takes as input sequential C code and attempts to discover and expose as much of the hidden parallelism inside the code and then rewrite it into SL.

C2μTC/SL focuses on loop structures. The reasoning behind this design choice is threefold: (i) loops have the potential for high degrees of parallelism (ii) most of the execution time of a program is spent inside loops and (iii) the SVP model offers special mechanisms that help accelerate single-dimensional loops. Hence, C2μTC/SL's goal is the transformation of loops into families. In the case where no dependences exist inside the original loop, then everything is mapped onto completely parallel threads inside a family otherwise the synchronizing channels are used to impose proper statement order.

Due to the fact that an SVP family is by definition a single-dimensional entity, translating multi-dimensional loops with loop carried dependences to families is a non-trivial task. That is why C2μTC/SL differentiates between loops of a single dimension and loops of multiple dimensions and acts accordingly in each case.

5.2. Single-Dimensional Loops

Single-dimensional loops can be mapped directly on SVP families and are categorized based on whether they contain loop carried dependencies (which indicates that a loop can be fully executed in parallel) or not and on what kind of ordering the loop carried dependencies impose on the execution (which even though it denies full parallelism, some might still be possible to expose). There are several categories that emerge based on this distinction and a list of them (alongside their transformation to SL) follows:

5.2.1. Loops without Dependencies

This is the simplest category of loops. A typical example looks like the one in Figure 5.1. (c is considered a constant or an expression which does not access A in any way). Figure 5.2 illustrates a slightly different loop that belongs to the same category: Even though there is a reference to A on the left-hand side of the assignment, there is no loop carried dependency, since each iteration of i only references itself and no other.

```
for ( i=0; i< N; i++)
      A[i] = c;
```

Figure 5.1. Typical Loop Without Dependencies.

```
for ( i=0; i<N; i++)
      A[i] = A[i] + c;
```

Figure 5.2. Another Example of a Loop Without Dependencies.

The way to transform these loops is quite simple and straightforward. A family of threads is created with the same bounds and stepping as the original loop and without any synchronization channel since each thread inside that family can execute in parallel. The code that each thread executes is the same code as the loop body, augmented with statements that deal with the decoupling of values from the

various global channels into local variables. The transformed code is depicted in Figure 5.3. Invocation of that family from the parent thread is illustrated in Figure 5.4.

```
sl_create( thread, void, sl_glparm(int, _c), sl_glparm(int *, _a) )
{
        sl_index( i );
        int *A=sl_getp(_a);
        int c=sl_getp(_c);

        A[i] = A[i] + c;
}
sl_enddef
```

Figure 5.3. The End Result of the Transformation of the Loop in Figure 5.2.

```
sl_create(,,0,N,1,,,thread, sl_glarg(int, _c, c), sl_glarg(int *, _a, A) );
sl_sync();
```

Figure 5.4. Invoking the Family of Threads of Figure 5.3 from the Parent Thread.

5.2.2. Loops with a Single Dependence

A more complicated situation arises when a loop carries a single dependence of an arbitrary length of x, where $x \geq 1$. In the extreme case where x=1 (a unary dependency), the original (pre-transformation) code looks like the one in Figure 5.5. The index space with the appropriate dependences is visualized in Figure 5.6.

```
for (i=1;i<N;i++)
        A[i]=A[i-1]+c;
```

Figure 5.5. A Typical Example of Unary Dependency.
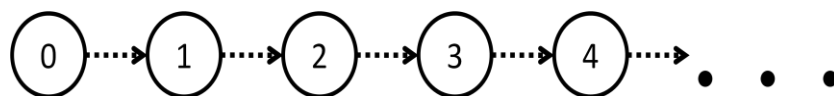
Figure 5.6. Visualization of the Index Space that Figure 5.5 Produces. The Dashed Arrow Indicates the Direction and Length of the Loop Carried Dependence.

It is clear that the synchronizing channel mechanism must be used to ensure proper statement order inside the family of threads that will replace this loop. However, since each iteration is expecting the result of the previous one, it is a perfect opportunity to utilize the synchronizing memory's ability to transfer data between threads. By passing the result of the computation of each thread to its successor through a shared variable, then each thread will not need to read the value from the global memory (A[i-1] per i) before it will perform its own calculation. This mechanism offers a high increase in efficiency by utilizing SVP's channels (that can be implemented in hardware) in a smart manner. Figure 5.7 displays the transformed result alongside its invocation code from the parent thread. Statements in bold indicate the beginning and end of the critical section inside the thread.

```
sl_def(thread, void, sl_shparm(int,_shared), sl_glparm(int *,_a), sl_glparm(int, _c) )
{
        sl_index(i);
        int c = sl_getp(_c);
        int *A = sl_getp(_a);

        int result;

        int shared=sl_getp(_shared);

        result=shared+c;

        A[i]=result;

        sl_setp(_shared, result);
}
sl_enddef

sl_create(,,1,N,1,,,thread, sl_sharg(int, _shared, A[0]), sl_glarg(int *, _a, A),
        sl_glparm( int, _c, c) );
sl_sync();
```

Figure 5.7. The Transformed Result of the Code in Figure 5.5.

All the family threads will initialize (decouple) their variables in parallel, calculate the result variable in a critical section, store it in the respective global memory place and then pass it over to their successor thread in the chain through the

shared variable. In order for this computation to work properly, the original value of A[0] must be passed through to the first thread through the synchronizing channel and that is the purpose of the initializing part in the *sl_create* statement. It is worth noting here that the dependent family is always executed on a single core and allows multiple threads to tolerate high memory access latencies. Additionally, although it would make more sense for the compiler to emit the sl_setp directive as early as possible in the code to allow for maximum parallelism, such a feature is not currently supported.

A more general example of the single dependence category is depicted in Figure 5.8. (code) and the corresponding index space visualization is illustrated in Figure 5.9.

```
for (i=x;i<N;i++)
    A[i]=A[i-x]+c;
```

Figure 5.8. A Typical Code Example of a Uniform Dependency with Length x.
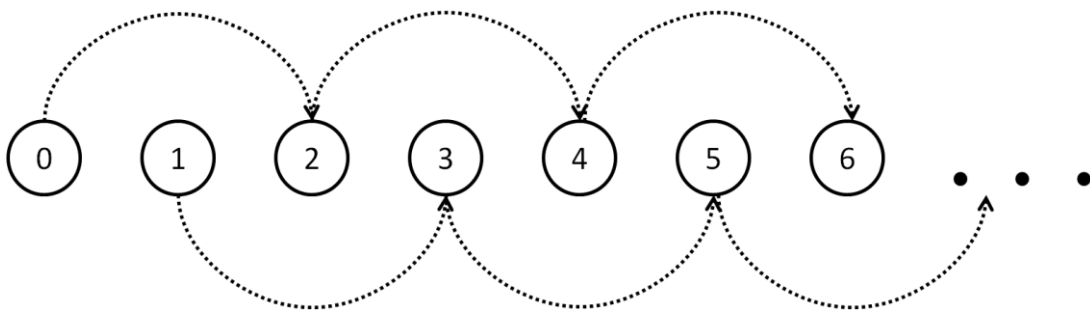
Figure 5.9. Index Space Visualization of a Single Dependence of Length x=2.

The way C2μTC/SL deals with such a situation is a bit more complex than the previous case: A dependence of length x, creates a series of implied data chains. Careful examination of Figure 5.9 shows that indices 2, 4, 6, 8, … belong to one data-chain while indices 3, 5, 7, … belong to another. Moreover, those two data-chains are completely independent from one another. Generally, a single dependence of length x, implies x completely independent data-chains. The first contains the index set (x, 2x, 3x, …) the second contains the set (x+1, 2x+1, 3x+1, …) etc. with the final one containing the set (2x-1, 3x-1, 4x-1, …).

In essence, even though a dependence exists, there is still parallelism to be exploited. All x data-chains can be executed in parallel which signifies a theoretical (in an ideal universe) increase of efficiency by a factor of x compared to the sequential model of execution. Taking into consideration the fact that each data chain is implemented by a single family (with a single shared variable which carries the value throughout the family), and that all x families need to run in parallel, which makes these families themselves children of another family of concurrently running threads, means that the hierarchy in the end is a bit more complex than the previous one since it now involves one more level in the concurrency tree. Figure 5.10 demonstrates the transformed code for such a paradigm alongside the invocation of the whole hierarchy that needs to be called in the parent thread.

```
sl_def(sequential, void, sl_shparm(int, _shared), sl_glparm(int *, _a), sl_glparm(int, _c) )
{
        sl_index(i);
        int *A=sl_getp(_a);
        int c=sl_getp(_c);
        int result;
        int shared=sl_getp(_shared);
        result=shared+c;
        A[i]=result;
        sl_setp(_shared, result);
}
sl_enddef

sl_def(parallel, void, sl_glparm(int *, _a), sl_glparm(int, _c) )
{
        sl_index(i);
        int *A=sl_getp(_a);
        int c=sl_getp(_c);
        sl_create(,,i,N,x,,,sequential, sl_sharg(int, _shared, A[i]), sl_glparm(int *, _a, A),
                sl_glparm(int, _c, c) );
        sl_sync();
}
sl_enddef


sl_create(,,x,2*x-1,1,,,parallel,sl_glarg(int *, _a, A), sl_glarg(int, _c, c) );
sl_sync();
```

Figure 5.10. Transforming the Code of Figure 5.8. Notice the Increase in Hierarchy Complexity.

5.2.3. Loops with Multiple Dependencies

As the last test case, loops with multiple dependencies are examined. A typical one dimensional loop with a series of different dependences is depicted in Figure 5.11, while Figure 5.12 demonstrates the (rather complex) index space. More specific cases might lack some of the dependences displayed, yet they are no different in their transformation than the general case. Considering that there are a total of x dependencies in the loop, we can see that in the end there will be x different shared variables, each shifting one place per iteration, and all of them are used to calculate the final result for every thread.

```
for (i=x; i<N;i++)
        A[i]=A[i-1] + A[i-2] + … + A[i-x] + c;
```

Figure 5.11. A Loop With x Different Dependencies.
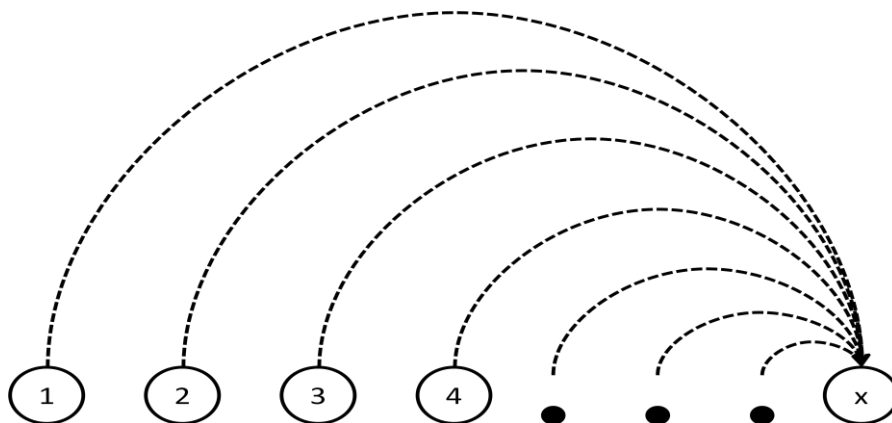
Figure 5.12. Visualization of the Loop of Figure 5.11.

```
sl_def(thread, void, sl_shparm(int, _s1), sl_shparm(int, _s2), …, sl_shparm(int, _sx),
        sl_glparm(int *, _a), sl_glparm(int, _c) )
{
        sl_index(i); int result;
        int *A=sl_getp(_a);
        int c=sl_getp(_c);

        int s1=sl_getp(_s1), s2=sl_getp(_s2), …, sx=sl_getp(_sx);
        result=s1+s2+s3+…+sx+c;
        A[i]=result;

        sl_setp(_s1, s2); sl_setp(_s2, s3); …; sl_setp(_sx, result);
}
sl_enddef
```

```
sl_create(,,x,N,1,,,sl_sharg(int, _s1, A[0]), sl_sharg(int, _s2, A[1]), …,
        sl_sharg(int, _sx, A[x-1]), sl_glarg(int *, _a, A), sl_glarg(int, _c, c) );
sl_sync();
```

Figure 5.13. Transformation and Invocation of a Loop with Multiple Dependencies.

It is worth noting here that at first glance nothing is gained. Both the original and the transformed code run sequentially. However the transformed code passes all the relevant data from thread to thread via the hardware synchronizing channel which helps alleviate the burden of accessing the global memory for every element needed. This helps increase speedup quite substantially.

5.2.4. Loops with Anti-Dependencies

As has been mentioned at a previous chapter, anti-dependencies are not true dependencies. When such a case of false dependency emerges, C2μTC/SL ustilizes a typical false dependence elimination technique: it copies the original array into a temporary array and then performs the actual computation. Figure 5.14 demonstrates code with an anti-dependence while Figure 5.15 shows the transformed code.

```
for ( i=0; i<N-x; i++)
        A[i] = A[i+x]+c;
```

Figure 5.14. A Typical Loop with an Anti-dependence.

```
sl_def(thread, void, sl_glparm(int *, _A), sl_glparm(int *, _Temp),
        sl_glparm(int, _c), sl_glparm(int, _step))
{
        sl_index(i);

        int *A=sl_getp(_A);
         int *Temp=sl_getp(_Temp);
        int step=sl_getp(_step);

        if (step==1) Temp[i]=A[i]; else A[i]=Temp[i+x]+c;
}
sl_enddef
```

```
sl_create(,,x,N,1,,,thread, sl_glarg(int *, _A, A), sl_glarg(int *, _Temp, Temp),
        sl_glarg(int, _c, c), sl_glarg(int, _step, 1));
sl_sync();

sl_create(,,0,N-x,1,,,thread, sl_glarg(int *, _A, A), sl_glarg(int *, _Temp,
        Temp), sl_glarg(int, _c, c), sl_glarg(int, _step, 2));
sl_sync();
```

Figure 5.15. Transformation and Invocation of the Anti-Dependence Loop.

This method completes the task with maximal parallelism albeit at the cost of reserving extra memory for the temporary array.

5.3. Multi-Dimensional Loops

Multi-dimensional loops are again divided into two major categories. Loops free from loop-carried dependencies and ones with dependencies. As we already know, lack of dependencies completely removes the need for maintaining any ordering in the execution of the code. So these kinds of loops are trivially transformed into fully parallel families. Each level in the loop-nesting corresponds to a family that executes completely in parallel. This creates a loop hierarchy similar to the one of the matrix multiplication example but without the sequential innermost loop.

However, in the case where loop carried dependencies do exist, the status quo changes. There is an ordering imposed in multiple dimensions now. C2μTC/SL can transform perfectly nested loops with a dependence vector into parallel constructs by utilizing the idea of Lamport's hyperplane method. However, since finding the optimal execution schedule for the hyperplances is not a trivial case, C2μTC/SL opted for a novel solution: Instead of pre-calculating the entire transformation (in compile time), most of the calculations are delegated to run-time. The hyperplanes are intuitively discovered and scheduled by tracing the dependence vector while executing the loop body. The whole algorithm is quite complex and so it will be presented in two steps: (i) The fixed size algorithm, which is the original and the main idea behind (ii) The self-adaptive algorithm which builds on (i) but completes it.

5.3.1. The Fixed-Size Algorithm

The main idea of the run-time algorithm is simple. If at any given moment, we know which sets of indices (index tuples) can execute, then by applying the dependence vector on that set, we can find out which tuples will execute at the next step. Consider Figure 5.16 which displays an excerpt from a random state of program execution. The grayed out index points indicate which indices execute at the current time (execution cycle). By applying the dependence vector (the arrows) in the set of indices currently executing, we can derive the set of indices that will execute in the next cycle (the white ones).

This algorithm emulates a mechanism where each index tuple locks down on itself (through the use of semaphores, one for each dependence) and each index tuple that executes, sends an unlock signal to the ones that depend on it (according to the vector). Since it is not possible to have a system with that many semaphores and in order to emulate the mechanism we need an n-dimensional array (2-d in the aforementioned example) which is initialized with 0 in all its cells. This array will store the number of dependences each cell satisfies at any given time. Before that array is created however, tiling needs to be applied to the index space since this algorithm does not offer satisfactory results at the finest level of granularity (as will be demonstrated in Chapter 6. Tiling is only applied on the innermost dimension as is demonstrated by Figure 5.17.
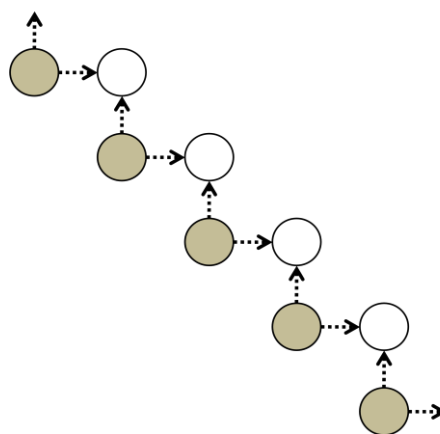
Figure 5.16. A Random State of the Index Space of a Nested Loop with two Dimensions. Arrows Indicate Dependences (2 in this Example).
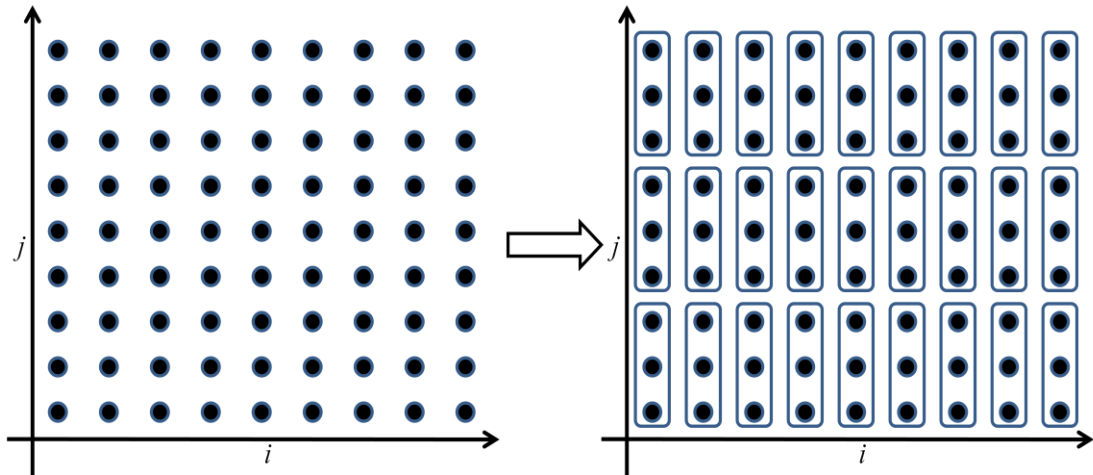
Figure 5.17. A Two-Dimensional Index Space Before and After Tiling. Each Tile has a Length of 3.

There are two reasons explaining the single dimensionality of the tiles: (i) A single dimensional tile can be applied on the SVP logic and architecture as a single family and thus be efficiently executed thanks to the relevant mechanisms. (ii) By organizing indices of the innermost loop (of the loop nesting) together cache usage is improved since these elements are usually mapped in neighboring memory addresses. Once tiling has been applied with a length of N per tile, then the algorithm begins execution (note that everything described is done during the actual execution and not during compile time) and a n-dimensional array is created with each cell corresponding to a tile. The cells of the array are then initialized with the number of dependences the corresponding tile has satisfied at the start of the execution. All the tiles that satisfy all their dependences store their index coordinates (in the form of tuples) inside a set of tuples V_1. At this point, a two-step computation takes place:

1. Create $|V_1|+1$ threads and synchronize. $|V_1|$ returns the number of tuples stored inside V_1. All threads "Perform Computation".
2. If $|V_2| = 0$ then computation ends. Otherwise copy V_2 into V_1, clear V_2 and goto 1.

"Perform Computation" comprises of the following steps (in the form of a pseudo-code function):

```
Perform_Computation()
{
        index i;
        if (i<length((V₁)+1)
        {
                create family of N threads with coordinates of the i_th tuple in V₁;
                sync_family ;
                return ;
        }
        else
        {
                for each tuple v in V₁
                        for each dependence d in dependence vector D
                        {
                                index tuple t= v + d;
                                array[t]++;
                                if (array[t]==length(D))
                                        add t to set V₂;
                        }
        }
}
```

In short, at any given computation cycle, $|V_1|+1$ threads are created that run concurrently. $|V_1|$ of them perform the actual original code's computation. This is actualized by creating a single family per V_1 tuple with size of N threads. The first thread in the family has the coordinates of the particular tuple that spawned the family while the rest of the threads follow on from that. Each of these families is executed in a sequential manner which means that parallelism in this algorithm is exploited between different families (hence the coarse level of granularity).

While those $|V_1|$ threads execute their computation, the last thread (called the *scheduler* thread) is actually traversing $|V_1|$ and adds each dependency from the dependence vector to each tuple of V_1. The new tuples that are produced that way are used as coordinates on the array which stores the number of currently satisfied dependences. Each of these new cells' values are increased by 1 per dependency and if that value reaches the total number of dependences, then those coordinates are added in set V_2. This means that V_2 stores the coordinates of the index tuples that will execute in the next computational cycle. If at the end of the computation cycle V_2 is empty, then this means that the entire index space has been covered and the computation ends.

For the sake of completeness a pseudo-code example is provided to help clarify its inner workings and help provide better understanding of the self-adaptive algorithm. Suppose the original code looks like the one in Figure 5.18. In this example there are clearly two dependences at work: (1,0) and (0,1). This means that the dependence vector D is the set D={(1,0), (0,1)}. C2μTC/SL will output the entire algorithm as the transformed version of the loop and everything else will take place at run time.

```
for (i =1; i < n ; j ++)
    for ( j = 1; j < n; j++)
        a[i][j] = a[i-1][j] + a[i][j-1];
```

Figure 5.18. The Original Code to be Transformed. The Corresponding Dependence Vector D={ (1,0), (0,1) }.

The algorithm, firstly, creates (dynamically) a two-dimensional array of size n x (n / N) where N is the fixed size of the tiles that will be used and then initializes the whole array. Initialization is pre-computed by the compiler and as such, it is tailored to that particular problem. This happens in order to reduce the initialization overhead. In the current problem with the particular dependence vector, the algorithm initializes the entire array except the first row and column with the value 0, the entire first row and column with the value of 1 and the corner at that intersection (index tuple (0,0)) with the value of 2. Since the total number of dependences is 2, that particular tuple is added to V_1. V_2 is set to be empty at this point. Figure 5.19 demonstrates the array as it is inialized. With this setup we know how many dependencies each tile has satisfied, which tiles can execute and which ones should stay dormant.

1	0	0	0	0	0
1	0	0	0	0	0
1	0	0	0	0	0
1	0	0	0	0	0
1	0	0	0	0	0
2	1	1	1	1	1

Figure 5.19. The Dependence Array as it is Initialized for a Nested Loop with a Dependence Vector D={ (1,0), (0,1) }

C2µTC/SL analyses the dependence vector and outputs the necessary code that will have each cell to be initialized with an appropriate value. Figure 5.20 demonstrates how these values are assigned to an array given a dependence vector of {(a, 0), (0,b)} where a and b are greater than 0.
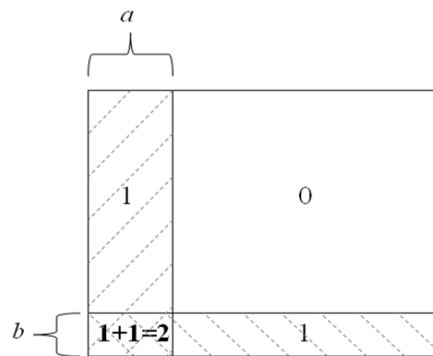
Figure 5.20. How the Dependence Array is Initialized Based on the Dependence Vector {(a,0),(0,b)}.

Cells near the edge of the array will always have at least one dependence satisfied (the one that comes from outside the grid) and cells where those areas intersect will contain the result of the summation of the each comprising area. In the case of having a dependency with negative components then the upper row of the array needs to be also initialized with appropriate values. Additionally, dependencies non-parallel to the axes are split into multiple dependencies parallel to the axes. I.e. a dependency of (1,1) becomes (1,0) , (0,1).

The main loop of the algorithm is simple enough:

```
while (true)
{
        Create (sizeof(V1) + 1) threads;
        Synchronize threads();
        if (sizeof(V2)==0) then break;
        Copy(V2, V1);
        Empty (V2);
}
```

Each thread runs the following code:

ThreadBody

```
{
        Thread_index in;

        if (in < sizeof(V1)) then
        {
                coordinates[] = V1.tuple[in];
                i = coordinates[0];
                j = coordinates[1];

                create a sequential family of N threads with thread body
                        the main program procedure( i, j);
                Synchronize threads();
        }
        else
        {
                for ( a = 0 ; a < sizeof(V1) ; a ++)
                {
                        coordinates[] = V1.tuple[a];
                        i=coordinates[0];
                        j=coordinates[1];
                        if (i+1 < n)
                        {
                                Array[i+1][j]++;
                                if (Array[i+1][j]==2) addToSet(V2, i+1, j);
                        }
                        if (j+1 < n)
                        {
                                Array[i][j+1]++;
                                if (Array[i][j+1]==2) addToSet(V2, i, j+1);
                        }
                }
        }
}
```

The addToSet procedure adds a new index tuple into V2. Each set is essentially a dynamic array which can continually expand when the need for more data arises. The main program procedure actually executes the original loop body:

```
main program procedure (i, j)
{
        Thread_index in;

        a[i][j+in] = a[i-1][j+in] + a[i][j+in-1];
}
```

Once the whole grid of coordinates is filled the V_2 set will eventually come up empty and the computation will end. The dependence array at a random state looks like Figure 5.21. In this example, the bottom left tiles indicate computations that have already taken place. The light grey ones indicate the tiles being computed in the current cycle. The scheduler thread follows each arrow (which signifies a dependency) and increases the number in that cell by 1. The dark grey tiles will all end up with 2 dependencies satisfied and thusly they will be added to the V_2 set for calculation in the next cycle.

1	0	0	0	0	0
2	0	0	0	0	0
2	2	0	0	0	0
2	2	2	0	0	0
2	2	2	2	0	0
2	2	2	2	2	1
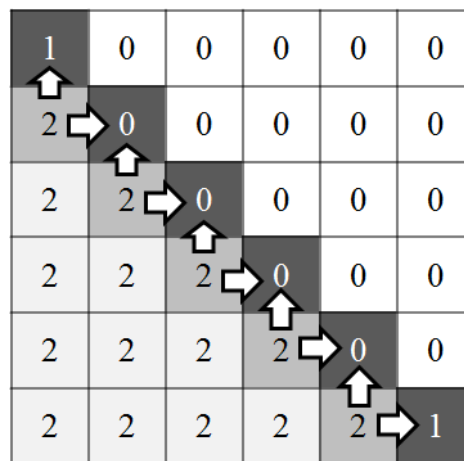
Figure 5.21. The Dependency Array at a Random State During Execution.

This run-time algorithm does away with trying to solve an NP-Complete problem and instead aims to intuitively discover the underlying parallelism. No hyperplanes are calculated, instead the dependence vector is applied on the index space and sets of tuples that can execute concurrently are discovered and scheduled. The end result is similar to any of the pre-computed methods which reduce the problem to a linear algebra one, while offering no need for heuristics. An added advantage of moving the solution to the run time is that irregular loops (i.e. triangular ones) can be dealt with exactly the same way by mapping the exploration space into the loop bounds. This versatility however does not come without a cost. There is an overhead incurred both during initialization (even if that can happen in parallel for maximal efficiency) and when the scheduler thread is running concurrently with the

rest of the computing threads so this run-time method will never be able to achieve the speedups offered by other methods however it can get rather close.

5.3.2. The Self-Adaptive Algorithm

The fixed-size algorithm described in 5.3.1 (so called due to the fact that the tiles are of a fixed pre-determined size) proved to be efficient however a disadvantage became soon apparent. The size of the tile was not, and could not be, known beforehand at the beginning of the execution. This number is a crucial parameter for the efficacy of the whole algorithm and picking the proper size proved to be a challenge not easily solved in the existing form of the run-time system.

The problem stems from the fact that too small a size results in too many tiles running in parallel, while too large a size means too few parallel tiles execute per cycle. The former situation means that each V_1 set is too large and consequently the scheduler must spend too much time traversing it while the rest of the computation threads have finished their computation. This in turn means that the main loop will idle for some time until the scheduler finishes. In the opposite situation, the scheduler finishes rather quickly however there is not enough parallelism to offset the overheads and so performance suffers.

It is reasonable to assume that the best solution lies somewhere in the middle: Where both the scheduler and the computation threads finish at the same time. Since that is practically impossible to achieve, a better solution would require all threads to finish their task as close to each other as possible. Measurements have validated this assumption, hence our best approach to a good tile size is the one that will create as many parallel tiles as are needed so as not to overwhelm the scheduler thread. Since this magic number is dependent on the problem, it becomes apparent that there is no method of calculating it. This led to the creation of a new algorithm, based on the fixed size one, however equipped with the ability to alter that tile size during execution in order to fine tune execution and aim for the optimal result. The self-adaptive algorithm is the next logical step to the fixed-size one. It incorporates all the versatility of moving the solution to the run time while at the same time abolishes the need for pre-existing knowledge of the tile size (or even resorting to some heuristics).

In order for the self-adaptive system to work, various changes and additions to the main algorithm were needed.

Firstly, a methodology was required which could determine at any given computation cycle whether the tile size needs to be increased or decreased: The execution time of all tiles that run in parallel and the execution time of the scheduler thread during each iteration are measured. Once each tile finishes, it stores its total execution time (in master CPU cycles) in an array. At the iteration's end, the slowest tile is selected and its timings are compared with the scheduler's ones. A *distance* between those two numbers is calculated which models the value of one as a percentage of the other. According to that distance then the following take place:

- If the absolute value of the distance is less than or equal to 0.25 (25%) then the two numbers are considered close to each other and no change is needed in the tile size.

- Otherwise:
  - If the scheduler finished before the tiles, then more tiles are needed to keep the scheduler occupied and hence the tile size needs to be reduced by 1.
  - If the scheduler finished after the tiles did, then fewer tiles are needed so the tile size needs to be increased by 1.

The main idea of the self adaptive algorithm is that once a particular tuple of indices finishes execution, then the following tuple in the lexicographic order will execute as well. In order for this to happen, a dependency is needed with the form of (0, 0. . . 0, a). When such dependence exists then "a" is considered to be of value of 1 since the length is irrelevant: the next tuple will execute from the point the current one ends. If there is no such dependency in the dependency vector then loop interchange is applied with the aim of creating one.

The algorithm solves the problem in an idealized index space that starts at (0, 0… 0) and its volume extends in all dimensions ad infinitum. The tiles before their execution transform those coordinates into proper index variables by adding the offsets for each dimension. The solving part is only interested in sets of indices in the form of (a, b, c, d… 0) since it is not possible to calculate which family in the innermost dimension can start due to the fact that the task size changes all the time.

However, based on the premise that once a tile starts working in the (a, b, c, d… 0) coordinates, we know that all of its subsequent successors will always be added in the queue to be executed since the dependency (0, 0, 0, …, 1) is always satisfied.

There is one final element that is needed for the self-adaptive algorithm to work properly. A method is needed to keep track of the index space that has been already covered by computation. This is necessary since with all the fluctuations of the tile size, a tile might be created with a length that surpasses this limitation and thusly be in danger of ruining the dependency order. To avoid such a situation, an extra array is used which stores the lengths covered for each coordinate of the form (a, b, c, …, 0). This array is called the *front* since it tracks the computational front as it expands over the index space from iteration cycle to iteration cycle. With everything mentioned so far in mind, during each iteration cycle the following series of events takes place:

1. The set of coordinates from the V_1 set is passed to the processing threads. There they are converted into proper index coordinates (by adding the corresponding offsets) and then the current front in the innermost dimension is assigned to be the starting coordinate. The current tile size is added and the ending coordinate is calculated. If it exceeds the loop bounds or the adjacent front (in the case where the task size grew since the previous cycle) then it is clamped accordingly. A second array which acts as a temporary front is updated when this computation finishes with the new front value for the current coordinates.

2. A thread family creation takes place which runs sequentially and performs computations on the set of the calculated indices. This family is timed and the amount of cycles it took is stored in an array.

3. The scheduler thread computes the next set of indices but it is only interested in families that will begin execution in the innermost dimension. Once the total number of dependencies satisfied reaches the total number of dependencies, then the particular tuple is added in the set of indices to be executed in the next cycle. The scheduler also adds to the same set the lexicographical successors of the tuples that are already running, as long as they don't exceed the front or the loop bounds.

4. The new temporary front array values are copied in parallel to the current front values.

5. The tile that took the longest time to complete is selected and its total time is compared to the time that the scheduler thread needed to complete and their distance is calculated. Once the distance is known,

(a) If the distance is lesser than or equal to 25% tile size remains the same.

(b) If the distance is greater than 25% and the scheduler finished first the tile size is reduced by one since more tasks are needed.

(c) If the scheduler finished after the computations then the tile size is increased by one.

This continues until the V2 set returns empty which signifies the computation's end. The whole algorithm in pseudo-code form follows:

The dependency array now changes and has its dimensionality reduced by one (since there is no point tracking dependences in the innermost dimension). In the case of the previous example with a two-dimensional loop and a dependence vector of {(1,0),(0,1)} it looks like Figure 5.22.

2	1	1	1	1	1	1	1	1	1

Figure 5.22. The Initialized Dependence Array for a Dependence Vector of D={(1,0),(0,*)}

By following that dependence array, we can see that in this example the first column with coordinates (0,0) gets "activated" first. Once it is activated, its successors (all tiles with coordinates in the form of (0,x)) will be queued for execution one by one. When the first tile finishes the second column (1,0) will activate and begin executing and so on.

The main tile procedure is:

```
main program procedure (i, j)
{
        Thread_index in;
        j = j + in;
        a[ i ][ j ] = a[ i − 1 ][ j ] + a[ i ][ j − 1 ];
}
```

The ThreadBody now becomes as follows:

```
ThreadBody
{
        Thread_index in;
        if ( in < sizeof(V1)) then
        {
                coordinates[ ] = V1.tuple[ in ];
                i = coordinates[ 0 ] + offset_i ;
                Depending on the status of Nold and Ncurrent
                calculate the "newFront[]" "length" and "coordinate" variables

                clockStart=getClock();
                if (length > 0) then
                {
                        create a sequential family of length threads with
                                thread body the main program procedure( i, j);
                        Synchronize Threads();
                }
                clocks[in]=getClock() – clockStart;
        }
        else
        {
                clockStart = getClock();
                for (a = 0; a < sizeof(V1) ; a++)
                {
                        coordinates[] = V1.tuple[a];
                        i = coordinates[0] + 1;
                        j = coordinates[1];
                        if (i>=offset_i AND i<n and j==0) then Array[i][j]++;
                        if (Array[i][j]==2) then addToSet(V2, i, j);
                        i = coordinates[0];
                        j = coordinates[1] + 1;
                        if (Front[i+offset_i] < n) then addToSet(V2, i, j);
                }
                Clocks[in]=getClock() – clockStart;
        }
}
```

The main while loop also changes into the following (Ncurrent stores the current tile size):

```
Nold = Ncurrent;
while (true)
{
        Create sizeof(V1) + 1 threads
        Synchronize threads();
        if (sizeof(V2)==0) then break;

        Copy NewFront[] to Front[] in parallel;
```

```
        max=Clocks[0];
        for (a=1; a<sizeof(V1); a++)
                if (Clocks[a]>max) then max=Clocks[a];

        percentage=(Clocks[sizeof(V1)] – max) / Clocks[sizeof(V1)];

        Nold = Ncurrent;

        if (Absolute(percentage) > 0.25) then
        {
                if (percentage < 0) then Ncurrent--;
                else Ncurrent++;
        }
        Copy(V2, V1);
        Empty (V2);
}
```

Figure 5.23 illustrates a random state of the dependency array. It is also worth noting that the only time the dependence (1,0) is taken into consideration when it points to a coordinate in the form of (a, 0) otherwise it is completely ignored since each tile queues the one above it in the V_2 set. Light grey tiles indicate the ones that are executing in the current iteration cycle while the arrows point to the ones that will be queued for execution in the next cycle. In that particular state, we can see that five on the "columns" have already been activated. Each activated column will keep rising until the loop bounds are reached. At the same time, each tile running on a column checks the front value of the column on its left in order not to move past it. Such an action might result in some computation taking place before its data are ready and produce false results. As the scheduler traverses all the running tiles, it eventually will notice that the fifth column increases the value at its right by one and this signifies that in the next cycle the sixth column can be activated as well.
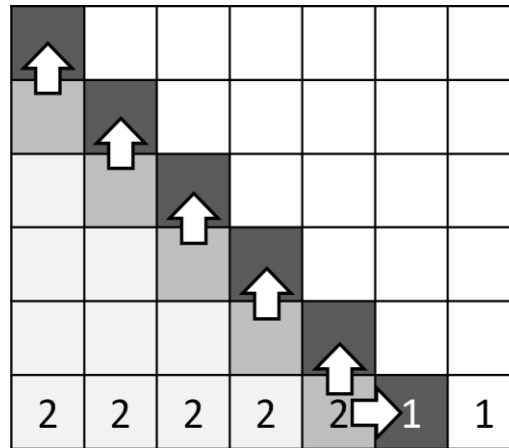
Figure 5.23. A Random State of the Dependency Array with the Executing Tiles.

5.3.3. Anti-dependences

While, in single-dimensional loops, anti-dependences are simply treated by utilizing a temporary array to copy the current one, when it comes to multi-dimensional loops, C2μTC/SL uses a different transformation. Instead of creating a copy array which might require large amounts of memory, it treats anti-dependences as dependences. For example a dependence vector of D={(-1,0), (0,-1)} is a vector that contains anti-dependences. In this case, C2μTC/SL without altering the code at all, multiplies the vector with the number -1. The new vector becomes D'={(1,0), (0,1)} which is a vector with dependences. When this happens it is a simple matter of employing the Self-Adaptive algorithm to deal with the problem.

This method solves the anti-dependence problem without sacrificing more memory and at the same time with some amount of parallelism exploited (although not full parallelism as would be the original case). This solution incorporates dependences and anti-dependences into one problem. An example loop which carries both types of dependences is one with a dependence vector of D={(1,0), (0,1), (-1,0), (0,-1)}. By switching the signs of the anti-dependences the new vector becomes D'={(1,0), (0,1), (1,0), (0,1)} which after simplification (since the same dependences appear more than once) ends up as D'={(1,0), (0,1)}. That way, the same Self-Adaptive algorithm that would have to be employed in the first place takes care of the anti-dependence problem as well in a parallel manner. More information on the Self-Adaptive Algorithm can be found in [1] from the Author's Publications.

5.4. From C to SL

Due to a series of software engineering related choices (affected by time constraints), C2μTC/SL works on a subset of the C language. In particular, the compiler only allows and attempts to parallelize the main function on an application's source file. Any other functions can be declared and implemented in other external files. The final executable can be produced by compiling all of the files together.

Since C2μTC/SL does not try to perform any sort of inter-procedural analysis, that is not a problem by itself. In addition, the existence of any jump statement (like *goto* or *continue* / *break*) is not supported. Jumps disrupt the natural flow of the code and can give the impression of a loop to the loop analysis component when jumping back into the code. Similarly, *break* and *continue* can also cause flow control problems hence they too are unsupported. Additionally, global variables are not supported; all should be declared inside main. Appendix B illustrates the subset of the C grammar (in BNF form) that is formally supported by C2μTC/SL. Unsupported programs do go through but the output of the compiler cannot be predicted and is at best random and chaotic and may even fail to compile.

Regardless of the source code being properly supported or not, the actual transformation is a two-part process: (i) Phase one entails parsing and analyzing the original source code and its loops. If everything goes well, an equivalent to the source code is produced but in a different, intermediate representation (IR). The IR contains the entire source code, broken down in basic blocks, with partially simplified expressions and where flow of code is only directed with gotos. Phase One is performed by an external compiler tool, called CoSy [76]. (ii) Phase Two is using the output of phase one (IR, loop analysis) as input to produce the final result. The code from the IR is reverse-transformed back into C-type code while knowledge of all the loops (index variables, boundaries, step values, basic blocks included in the loop) is used for loop analysis. That kind of analysis however first needs the loops to be organized into single units and to be examined as such units. These units form the basis of C2μTC/SL's functionality as they are the fundamental blocks that get analyzed and transformed (depending on the analysis). These loop groups are called *Masterloops*.

5.4.1. The Masterloops

      A masterloop is nothing more than a perfect nesting of loops. It contains, in its loop body, statements as well as more masterloops. All analyses and transformations take place on a per masterloop basis and they are all independent from one another. Figure 5.24 demonstrates a code snippet where everything belongs to one masterloop, masterloop 1, which is comprised of loops i and j and contains a single statement as its body.

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        A[i][j]=0;
```

Figure 5.24. A Perfect Loop Construct Which Comprises a Single Masterloop.

      Figure 5.25 displays the classic matrix multiplication code that has been used before. Loops i and j are perfectly nested and behave as a single structure while loop k is independent from the previous ones and performs its own calculations. If loop k was missing, then the original nesting would still make sense: For each iteration (i, j) the variable sum would take the value of 0 and each element of C[i][j] would take the value of sum. Following this logic, C2μTC/SL separates that code into two masterloops: Masterloop 1 is created by loops i and j, and its loop body contains the statement "sum=0", another masterloop and the statement "C[i][j]=sum". Masterloop 2 is comprised only of loop k and its body is the same as the loop body of k. Each masterloop is analyzed independently. In the end, once all transformations are done and each piece of the final code comes into place the result will be a proper transformed parallel matrix multiplication code.

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
    {
        sum=0;
        for (k=0; k<n; k++)
            sum=sum+A[i][k]*B[k][j];
        C[i][j]=sum;
    }
```

Figure 5.25. A Typical Matrix Multiplication Code which Contains Two Masterloops.

5.4.2. Dependence Analysis in a Masterloop

Each Masterloop is analyzed for the existence of loop carried dependences. This takes place as a two-fold process. Firstly scalar variables are examined. Those who carry data from one loop iteration to the next signify dependence and hence those variables will become shared ones in the transformed code. Detecting these kinds of variables is relatively trivial and straightforward: Each variable inside the loop body is examined. If during an iteration that variable is read before it is written then it has to become a shared one, otherwise it is a temporary variable only viable for the current iteration and thusly does not impose any particular ordering in the loop. Moreover, for each variable under examination, the source code is further analyzed. If the current variable is accessed (for reading) again at some later point in the code then this means that this variable is carrying the result of some computation and should, again, be marked as a shared one. The difference is slight and can only appear in certain situations. Consider the code in Figure 5.26. It calculates the n_{th} Fibonacci number.

```
for (i = 2; i < n; i++)
{
        c = a + b;
        a = b;
        b = c;
}
```

Figure 5.26. A Loop that Calculates the n_{th} Fibonacci Number (n > 2, a and b are Initialized to 0 and 1 Respectively, c Carries the End Result).

When transforming this code, the variable C is first written and then read. This means that the compiler can detect it as a thread local one. Only by accessing c after the end of the loop (e.g. by printing it) can the compiler see that its value is needed and thusly mark it as shared so that it can be accessed after syncing.

The second part of the process deals with arrays and their subscripts. C2µTC/SL looks for expressions in the form of array[index] =array[index ± constant]. Through expressions like that it is able to deduce the various dependences that may exist and so build the dependence vector. Any other form of expression when it comes to array access is currently not recognized and the loop is marked as one not to be transformed. If no shared scalar variable is found or no array access that will result in

a wavefront solution and if the current masterloop is not marked to be left untransformed then it is assumed that it can be executed fully in parallel.

5.4.3. Transformation of a Masterloop

During code transformation, each statement is copied into the output until a Masterloop is met. At that point, the transformed Masterloop takes the place of the original in the code and this continues until the end of the program. Each masterloop is transformed according to the results of the analysis that transpired in the previous step:

(i) If no dependences are located and the loop is not marked to remain untransformed then it is converted into a fully parallel construct. Each loop in the masterloop, from the innermost to the outermost, is first implemented as a thread function and then its corresponding invocation (through a pair of sl_create / sl_sync calls) is added in the appropriate place in the code. There are some compiler options that can dictate which of the outermost loops will be forced to run sequentially in the case of a deep nesting. In such a case, by having all loops run in parallel, the SVP will soon run out of resources and revert back into a sequential mode. In this situation it is prudent to have the outermost loops run sequentially in order to exploit more parallelism in the lower levels of the hierarchy.

(ii) If one (or more) shared variables have been detected then in a similar manner to the previous method, each loop is implemented and invoked, only this time the arguments of the sl_create method incorporate some shared variables, whose values are read right after the sl_sync (through the use of sl_geta).

(iii) If a dependence vector was detected, then the loop is transformed radically and the self-adaptive algorithm takes its place.

(iv) Finally if neither of the previous options applied to the particular masterloop, then it is copied in its entirety into the output without any transformation.

Returning to the matrix multiplication example described in Figure 5.25, C2μTC/SL makes the following deductions: The first Masterloop can be run completely in parallel since the variable "sum" is initialized in its iteration and the statement "C[i][j]=sum;" relies only on that variable which will have a place in the thread local storage. The second Masterloop cannot be fully parallelized. "sum" is

first read and then written in each iteration, thusly it is marked as a shared variable. Putting these deductions together results in the code illustrated by Figure 5.28. Several things are worth noting about the code in that Figure:

 1) In order for the SL macro definitions to work properly all variable types must be simplified. That is, each variable can have a name and optionally a '*' symbol indicating a pointer to that type of variable. Multidimensional arrays cannot simply be used on the macro definitions, hence types are defined (using C's typedef) which are essentially some array of a basic type. Those new typedefs can then be easily pass through the macro definitions. In that particular example, the arrays are considered of size [10][x].

 2) Each create / sync is encompassed in a block of code (denoted by the { and } symbols). This is needed as some SVP macros declared during the creation / syncing might interfere with variables of the actual code. By having the whole process in its own block helps to easily avert confusing the compiler and producing error messages, interrupting the process altogether.

 3) There is no parallel calculation of the partial sums a[i][k]*b[k][j] in that code. This happens due to the fact that C2μTC/SL pushes the reading of the shared variable down in the code, just before the statement that needs it.

 However in this case the statement is calculating the partial sum and updating sum in one statement and C2μTC/SL currently lacks the capability to break the statement in order to interject the sl_getp statement (such a mechanism is to be implemented at a later stage). In order for this code to work as intended, loop k's body should be like the one displayed in Figure 5.27 which also illustrates how the resulting code would change.

5.4.4. Code Generation

 The final step in the code transformation is the actual code generation. Initially all the *typedefs* are listed, followed by thread definitions for each masterloop that can be transformed, in such an order that any thread definition always precedes that thread's invocation. Each thread definition is designed to be self contained. All related variables are passed as arguments and initialized at the beginning of the thread code via the sl_getp() directive. The code body itself is the code of the masterloop. All

basic blocks are listed in the same order they appear in the original IR, so as not to change the functionality of the code in question. Each basic block before code emission is examined for ownership (masterloop basis). If it belongs in another masterloop that means that instead of listing that code, invocation for that masterloop is created instead in its place (assuming always that the masterloop can be transformed). Invocations vary according to the type of transformation incurred on the masterloop. After the sync, all shared variables related to that masterloop retrieve their values (via the sl_geta() directive) and the code listing continues. When all the masterloop's code has been emitted, all shared variables are written back to their respective shared channels (sl_setp()) and the thread definition is finalised with the sl_enddef keyword.
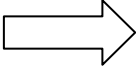
Once all threads have been defined, the main thread is defined. All variables are declared inside of the definition as local and then code generation begins in exactly the same manner as before. Basic blocks are listed in turn until one is found that belongs to a transformable masterloop (masterloops who were deemed untransformable do not exist in the masterloop list so they just get emitted verbatim). In this case the necessary invocation is placed and the code continues with the next available basic block that does not belong to any loop.

Thread invocation code can vary depending on the kind of transformation applied to a masterloop and can range from simple invocations (a simple fully parallel loop for example) to the most complicated ones (a nested loop with a dependence vector where the Self-Adaptive algorithm is employed). There exists a templated code for each transformation case that gets emitted every time with certain variables taking code-specific values to ensure proper code execution.

```
…
for (k=0; k<n; k++)
{
        int tmp=a[i][k]*b[k][j];
        sum=sum+tmp;
}
…
```

⟹

```
…
int tmp=a[i][k]*b[k][j];
int sum=sl_getp(_sum);
sum=sum+tmp;
sl_step(_sum, sum);
…
```

Figure 5.27. The Necessary Change in the Original Matrix Multiplication Code Needed for the Partial sums to be Calculated in Parallel.

```
typedef int a10[10];
sl_def(masterloop_2_k, void, sl_glparm(a10 *, _a), sl_glparm(a10 *, _b), sl_shparm(int,
_sum), sl_glparm(int, _i), sl_glparm(int, _j))
{
        sl_index(k);
        int i=sl_getp(_i);
        int j=sl_getp(_j);
        a10 *a=sl_getp(_a);
        a10 *b=sl_getp(_b);

        int sum=sl_getp(_sum);
        sum=sum+a[i][k]*b[k][j];
        sl_setp(_sum,sum);
}
sl_enddef

sl_def(masterloop_1_j, void, sl_glparm(a10 *, _a), sl_glparm(a10 *, _b), sl_glparm(a10*, _c),
sl_glparm(int, _i))
{
        sl_index(j);
        int i=sl_getp(_i);
        a10* a=sl_getp(_a);
        a10* b=sl_getp(_b);
        a10* c=sl_getp(_c);
        int sum=0;
        {
                sl_create(,,0,n,1,,,masterloop_2_k, sl_glarg(a10*, _a, a), sl_glarg(a10*, _b,
b), sl_sharg(int, _sum, sum), sl_glarg(int, _i, i), sl_glarg(int, _j, j);
                sl_sync();
                sum=sl_geta(_sum);
        }
        c[i][j]=sum;
}
sl_enddef

sl_def(masterloop_1_i, void, sl_glparm(a10 *, _a), sl_glparm(a10 *, _b), sl_glparm(a10 *,
_c))
{
        sl_index(i);
        a10* a=sl_getp(_a);
        a10* b=sl_getp(_b);
        a10* c=sl_getp(_c);

        {
                sl_create(,,0,n,1,,,masterloop_1_j, sl_glarg(a10 *, _a, a), sl_glarg(a10 *, _b,
b), sl_glarg(a10 *, _c, c), sl_glarg(int, _i, i));
                sl_sync();
        }
}
sl_enddef

{
        sl_create(,,0,n,1,,,masterloop_1_i, sl_glarg(a10 *, _a, a), sl_glarg(a10 *, _b, b),
sl_glarg(a10 *, _c, c));
        sl_sync();
}
```

Figure 5.28. The parallel result of the code in Figure 5.25.

Special mention goes to while loops. They are treated as for loops, however there is one difference. The invocation code is wrapped inside a while loop. This way there is actual loop transformation but each create takes a predefined number of threads as an argument. The guard condition of the while loop is emitted at the beginning of the thread code so that once it stops being valid, the thread invokes sl_break and code execution resumes back in the invocation part. In order for the invoker to know that the while loop issued a break, a certain boolean variable exists which is associated with that particular masterloop that is set to TRUE when the break is called. This tells the invoker code to stop its own while loop via C's break and continue execution after that. As an example let's consider the code of Figure 5.29.

```
int main(void)
{
        int a[10],i,f,c;

        while(1)
        {
                f=0;
                for (i=0;i<9;i++)
                        if (a[i]>a[i+1])
                        {
                                c=a[i];
                                a[i]=a[i+1];
                                a[i+1]=c;
                                f=1;
                        }

                if (f==0) break;
        }

        return (0);
}
```

Figure 5.29. Original code that performs bubble sort.

Figure 5.30 demonstrates the loop transformation. The innermost loop is properly transformed into a sequentially executed loop (the shared variable f makes sure of that) where each element of the array is checked with its subsequent and swap places if necessary. The interest lies with the umloop_2 loop. Firstly it's made sequential with the introduction of the shared variable _serialize since the analyzer was unable to detect if it can be run in parallel or not. the if (TRUE) statement is the transformation of the while(1) from the original code. If it was any other expression it would have been copied as well. Every time a break statement is introduced in the original code, an sl_break one is emitted in the result, with the addition that the array

_result[#interal_loop_number (2 in this example)] becomes 1 to signify that the loop has finished execution.

```
sl_def(mloop_1_inner,void,sl_glparm(int4*,_a),sl_glparm(int4,_c),sl_shparm(int4,_f))
{
        sl_index(i);
        int4*   a = sl_getp( _a );
        int4    c = sl_getp( _c );
        int4    f = sl_getp( _f );

        if  (a[i] > a[(i+1)]) goto bb9; else goto bb10;
bb9:;
        c = a[i] ;
        a[i] = a[(i+1)] ;
        a[i+1] = c ;
        f = 1 ;
bb10:;
bb11:;
        sl_setp(_f, f);
}
sl_enddef

sl_def(umloop_2,void,sl_glparm(int4,_f),sl_glparm(int4,_i),sl_glparm(int4*,_a),sl_gl
parm(int4,_c),sl_shparm(int,_serialize))
{
        int4    f = sl_getp( _f );
        int4    i = sl_getp( _i );
        int4*   a = sl_getp( _a );
        int4    c = sl_getp( _c );
        int serialize= sl_getp(_serialize);

        if  (   TRUE  ) ; else {_result[2]=1;sl_break();};

        f = 0 ;
        i = 0 ;

        {
                sl_create(,,0,9,1,,,mloop_1_inner,sl_glarg(int4*,_a,a),sl_glarg(int4
,_c,c),sl_sharg(int4,_f,f));
                sl_sync();

                f = sl_geta(_f);
        }

        if (f == 0)  {_result[2]=1;sl_break();} else goto bb14 ;

bb14:;
bb15:;
        sl_setp(_serialize, serialize+1);
}
sl_enddef

while(1)
{
        {

sl_create(,,0,_MAX_THREADS,1,,,umloop_2,sl_glarg(int4,_f,f),sl_glarg(int4,_i,i),sl_g
larg(int4*,_a,a),sl_glarg(int4,_c,c),sl_sharg(int,_serialize,0));
                sl_sync();
        }

        if (_result[2]==1) break;
}
```

Figure 5.30. The entire transformation (including invokation at the bottom) of the bubble sort while-loop of Figure 5.29.

Finally in the invocation, we can see that a while(1) is emitted that runs the loop sequentially for MAX_THREADS number of iterations and then the _result[2] is checked. If it has the value of 1 then the loop is considered to have finished and contol breaks out of the while.

CHAPTER 6. EVALUATION OF THE C2μTC/SL COMPILER

Introduction

Single-Dimensional Loops

Multi-Dimensional Loops

Livermore Loops

6.1. Introduction

Evaluating C2μTC/SL is a more complex process than just simply running and timing the transformed programs. Since its output is the SL language, the only way to execute the parallel applications is to utilize the SVP pipeline. This effectively means running the simulator system bundled with the SL compiler. However, moving into a simulated environment means that a simple timing methodology would not provide any meaningful results.

Selecting the metric we'd use for the evaluation meat turning to the simulator itself. Once the program runs, its internal Master CPU cycles counter starts counting from 0. Then the simulator sets up a series the whole execution environment and once everything is complete then the SL application starts executing. The Master CPU cycles counts the number of parallel cycles all cores executed. It can be used to determine the number of cycles (throughout all the cores) that were needed for any program to execute. A faster running application needs fewer cycles. "CPU cycles" is a constant metric unaffected by the host's CPU clock or anything else and is directly proportionate to the overall speed of an application. It also allows for percentile comparisons to take place between different applications.

For any program to be measured and compared, this would effectively mean that the program would have to be compiled and executed inside the SVP simulator environment even if that meant rewriting portions of the original code into SL form. All measurements presented in this chapter were obtained by simulating an environment of 8 cores (unless stated otherwise) and all results, as stated above, are in CPU cycles and the measurements were taken from the actual computation part of each program ignoring system and program and system initializations. SL offers two macros that can be inserted between two places in a code. The output of the simulator then can display the number of master cpu cycles that were spent inside that piece of code. We marked only the part of code that performs the actual computation. Using the master cpu cycles we could measure speedups gained with this formula:

$$Speedup = \frac{Sequential\ Cycles}{SL\ Cycles}$$

Essentially, an SL code that completes in half the time of the original sequential version will have a Speedup of 2 while codes that are slower than the original version will have speedups less than 1. In a manner similar to Chapter 5, evaluation is split into two general categories: Single-Dimensional loops and Multi-Dimensional loops. A well-known benchmark suite was also used to test the C2μTC/SL's general parallelizing abilities, the Livermore Loops.

6.2. Single-Dimensional Loops

The first and simplest example measured was that of a single dimensional for loop with no loop carried dependencies. The loop body consisted only of the statement "A[i]=i+1;". Table 6.1 demonstrates the results of measuring the two codes (sequential and parallel) and the speedup achieved, while Figure 6.1 illustrates these results graphically. It is clear that the transformed parallel code is much faster than a pure untransformed loop (as was anticipated).

Table 6.1. The Results of the Execution Times (in Cycles) of a Simple Sequential and Parallel Application.

Problem Size (N)	Sequential For	Fully Parallel SL	Speedup
100	9120	3616	2,522
200	16808	4348	3,866
300	23408	5068	4,619
400	31160	6380	4,884
500	37888	6812	5,562
600	44972	7304	6,157
700	52204	8288	6,299
800	59400	9184	6,468
900	66656	9692	6,877
1000	73852	11060	6,677
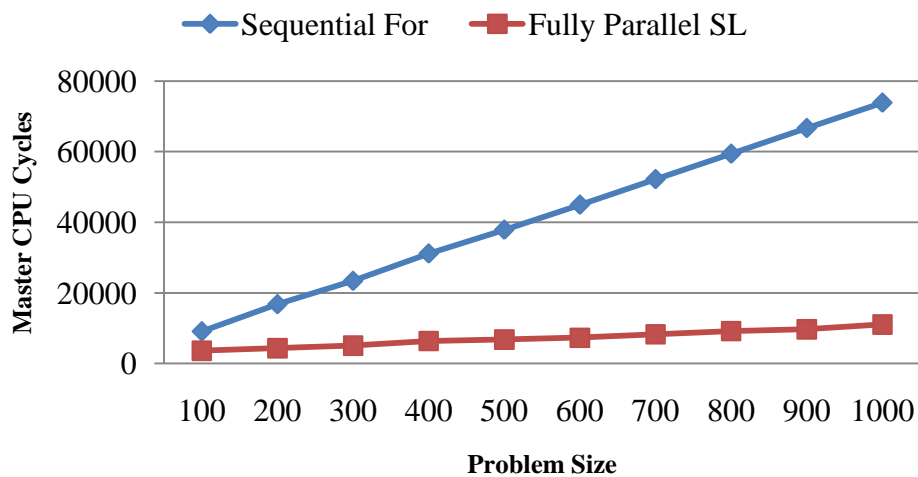
Figure 6.1. Comparing the Data of Sequential and Parallel Code in Graph Form.

A small variation was also implemented (manually): The SL parallel code was changed into a fully sequential one (This was achieved by adding a shared channel that transferred dummy data between threads and kept the sequential ordering. The entire thread body was turned into a critical section). The aim of that change was to test the SVP model and how it fares when a fully sequential loop running without using any of the amenities provided by the system against a classic for-loop. Table 6.2 displays the results while Figure 6.2 visualizes the data.

Table 6.2. Comparing a Sequential For-loop with a Sequential Family of Threads Running the Same Code.

Problem Size	For Loop	Sequential SL	Speedup
100	9120	9016	1,012
200	16808	16012	1,050
300	23408	22048	1,062
400	31160	29000	1,074
500	37888	35016	1,082
600	44972	41512	1,083
700	52204	47844	1,091
800	59400	54644	1,087
900	66656	60912	1,094
1000	73852	67744	1,090
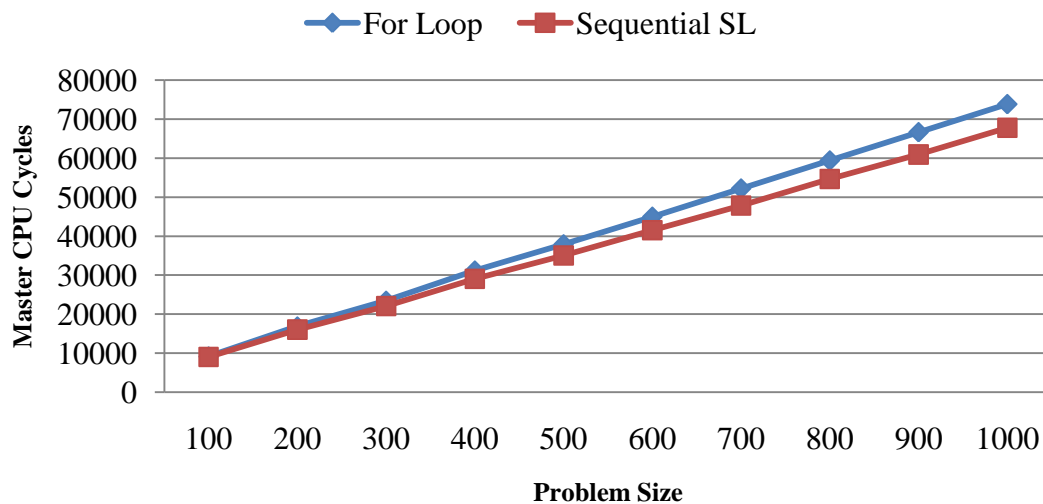
Figure 6.2. Comparing a Sequential For-loop with a Sequential Family of Threads.

There is an increase in speedup stems from two factors: (i) hardware controlled iterations: there is no actual software increment of the index variable or test to see if it exceeds its bounds and (ii) minimal synchronization overhead. This example makes it clear that even if a loop cannot be transformed in any meaningful way, (either by taking advantage the shared memory system or by exposing some hidden parallelism) re-writing it into SL form will offer a small increase in the overall speed of the program.

The next category of problems contains loops with a single dependency. Dependency of length 1 is examined firstly (Figure 6.3). The comparison data is displayed on Table 6.3 and its visualization is given on Figure 6.4. Even if the simulator setup contains 8 cores, such an example will be constrained in one core both in its original version and its transformed one and so the expected speedup should not be around 8.

There is only one data chain hence there can be no parallelism in its execution. However there can be instruction level parallelism by exploiting SVP's high memory latency tolerance: Memory related operations can be overlapped with other instructions and thusly speedups higher than 1 can appear. In addition to memory tolerance, by utilizing the synchronizing channel as a data carrier, each thread can do away with looking up the global memory for information, an action that also increases efficiency by a remarkable degree.

```
for (i=1; i<n; i++)
    a[i]=a[i-1]+1;
```

Figure 6.3. Loop with a Single Dependency of Length 1.

Table 6.3. Comparison Between the Sequential for and the Transformed SL Code.

Problem Size (N)	Sequential For	SL code	Speedup
100	10920	8552	1,277
200	20360	14932	1,364
300	28380	19356	1,466
400	37760	25292	1,493
500	45800	29668	1,544
600	54540	34912	1,562
700	63196	39644	1,594
800	72016	45096	1,597
900	80588	49952	1,613
1000	89548	55252	1,621
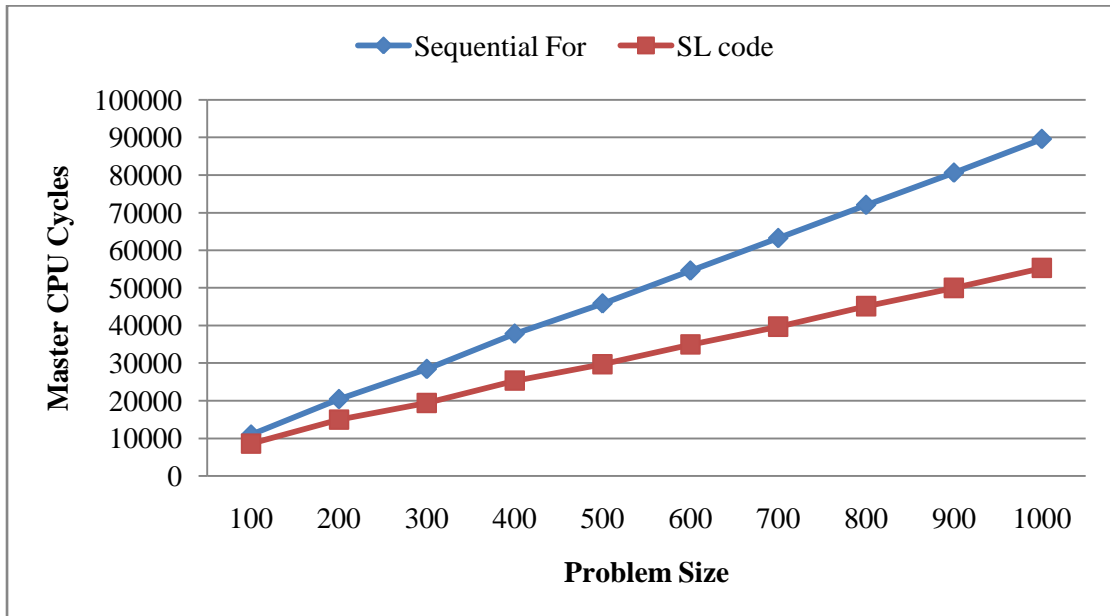
Figure 6.4. Comparing Sequential and SL Codes With a Dependency of Length=1.

The problem of a single dependency of length 2 was subsequently transformed, executed and evaluated. This time the existence of 2 independent data chains means that 2 cores would be utilized. Figure 6.5 illustrates the original code, Table 6.4 contains the results of the executions and Figure 6.6 visualizes that data.

```
for (i=2; i<n; i++)
    a[i]=a[i-2]+1;
```

Figure 6.5. A Loop with a Dependency of Length 2.

Table 6.4. Results of the Transformed Loop with a Dependency of Length 2.

Problem Size	Original Loop	SL code	Speedup
100	10836	7604	1,425
200	20116	11372	1,769
300	28140	16092	1,749
400	37432	20792	1,800
500	45476	25200	1,805
600	54048	29552	1,829
700	62616	32948	1,900
800	71364	37740	1,891
900	80004	41572	1,924
1000	88808	45960	1,932
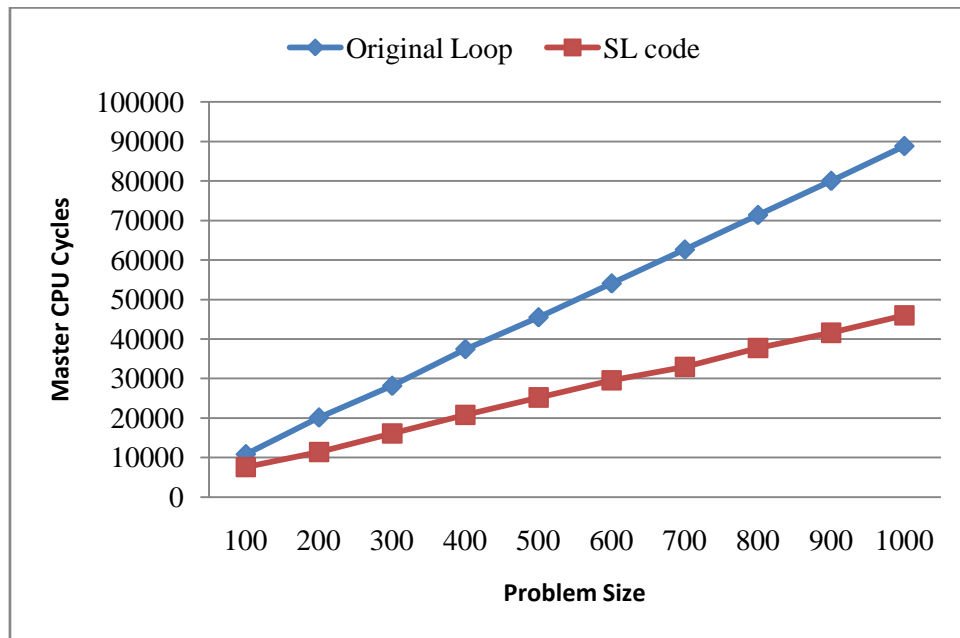
Figure 6.6. Comparing Sequential and SL Codes with a Dependency of Length=2.

In a similar manner, the problem of a single dependency but of length 5 was transformed and evaluated (5 cores utilized). Figure 6.7. illustrates the original source code while Table 6.5 and Figure 6.8 display the resulting data.

```
for (i=5; i<n; i++)
    a[i]=a[i-5]+1;
```

Figure 6.7. A Loop With a Single Dependency of Length 5.

Table 6.5. Results of the Transformed Loop with a Dependency of Length 5.

Problem Size	Original Loop	SL code	Speedup
100	9960	7252	1,373
200	18748	10540	1,779
300	26228	14472	1,812
400	35116	17632	1,992
500	42724	21444	1,992
600	50800	25264	2,011
700	59016	28384	2,079
800	67244	32400	2,075
900	75484	35780	2,110
1000	83680	39312	2,129
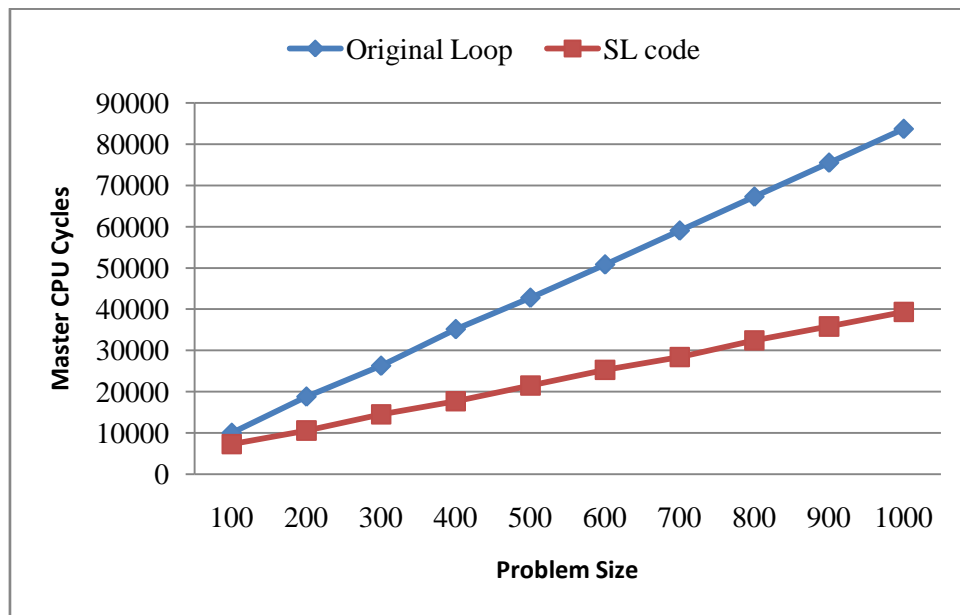
Figure 6.8. Comparing Sequential and SL Codes with a Dependency of Length=5.

For a problem size of N=1000, a dependency of length 2 (which produces 2 parallel data-chains running) offers a speedup of about 1,9 while a dependency of length 5 (which creates 5 parallel data chains) offers a speedup of about 2,13. This means that the increase in efficiency is not proportional to the increase of the number of parallel chains in existence. This result deviates from the expected speedup of 2 for a dependency of length 2 and 5 from one of length 5. This deviation can be attributed to the overhead introduced by SVP's housekeeping: 5 parallel chains require more time spent context switching and a lot more resources since each chain also creates one synchronizing channel. If the SVP runs out of resources then it gracefully reverts back into a sequential execution mode in order to serve the rest of the requests. Adding to the overall overhead is the fact that more families equal to more communication between parent and descendant threads.

Loops with multiple dependences were examined next. The general form is the one illustrated in Figure 6.9. We tested loops with 2, 3, 4, and 5 multiple dependences and the results are shown in Tables 6.6 to 6.9 respectively and visualized in Figures 6.10 to 6.13. It is becoming apparent that the more shared variables (channels) are involved in the process, the slower the execution becomes.

Finally, the anti-dependency example (Figure 6.14) is evaluated. Table 6.10 illustrates the results and Figure 6.15 the visualization of that data. Even though the transformed code entails a two-step process, the fact that everything takes place in parallel in an 8 core environment, with the help of cache utilization when it comes to the second step of copying back, provides a very good speedup of about 4.45. The only drawback is the allocation of extra space for a temporary array.

```
for (i=2, 3, 4, 5; i<n; i++)
        a[i]=a[i-1]+a[i-2](+a[i-3](+a[i-4](+a[i-5])));
```

Figure 6.9. A General Form of a Loop with Multiple Dependences (2 to 5).

Table 6.6. Comparing Sequential and SL Codes with 2 Dependences.

Problem Size	Original Loop	SL code	Speedup
100	11600	8912	1,302
200	21896	15660	1,398
300	30700	21100	1,455
400	40848	27980	1,460
500	49748	33452	1,487
600	59328	39760	1,492
700	68676	45560	1,507
800	78356	51884	1,510
900	87696	57760	1,518
1000	97512	64172	1,520
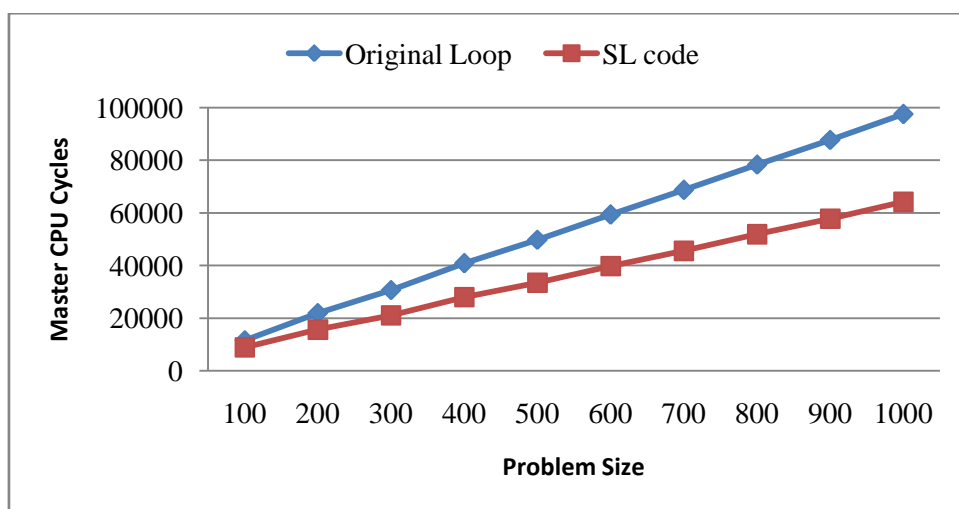
Figure 6.10. Comparing Sequential and SL Codes with 2 Dependences.

Table 6.7. Comparing Sequential and SL Codes with 3 Dependences

Problem Size	Original Loop	SL code	Speedup
100	12036	9524	1,264
200	22744	15552	1,462
300	32088	22436	1,430
400	42632	28928	1,474
500	52068	35932	1,449
600	61996	43040	1,440
700	71896	49176	1,462
800	81960	56180	1,459
900	91848	62676	1,465
1000	102076	69124	1,477

Table 6.8. Comparing Sequential and SL Codes with 4 Dependences.

Problem Size	Original Code	SL code	Speedup
100	11936	9224	1,294
200	23140	16468	1,405
300	32408	24152	1,342
400	43508	31112	1,398
500	52904	38648	1,369
600	63232	45292	1,396
700	73516	52832	1,392
800	83676	59704	1,402
900	93824	67124	1,398
1000	104472	74844	1,396

Table 6.9. Comparing Sequential and SL Codes with 5 Dependences.

Problem Size	Original Code	SL code	Speedup
100	13172	14680	0,897
200	23612	23824	0,991
300	34992	33936	1,031
400	45816	43276	1,059
500	57056	53296	1,071
600	68884	63612	1,083
700	79284	72556	1,093
800	90960	83052	1,095
900	101500	92020	1,103
1000	112732	102152	1,104
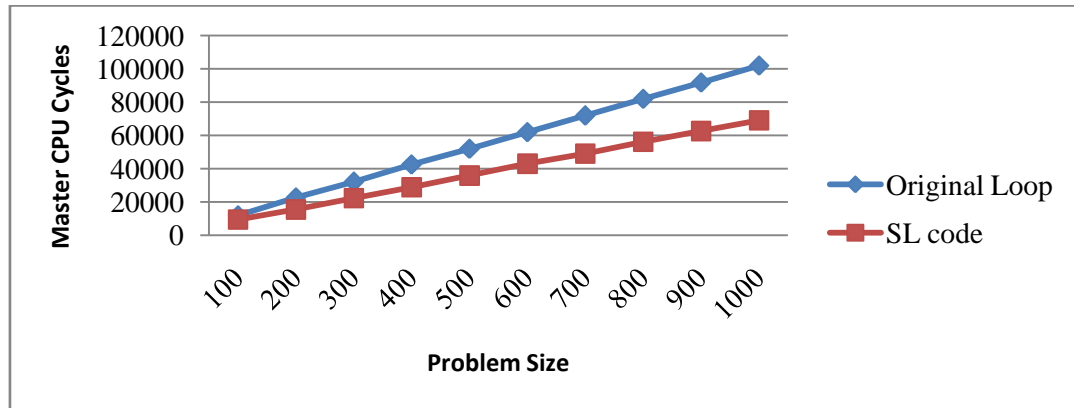
Figure 6.11. Comparing Sequential and SL Codes with 3 Dependences.
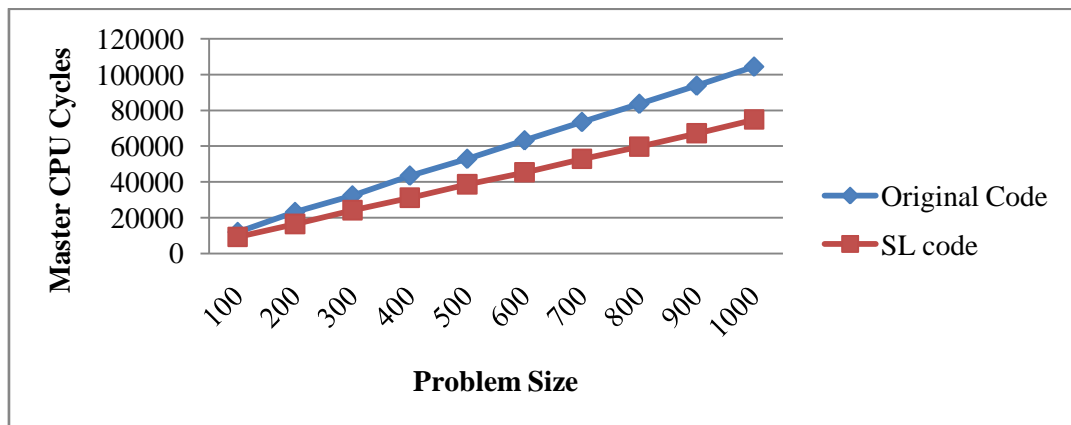
Figure 6.12. Comparing Sequential and SL Codes with 4 Dependences.
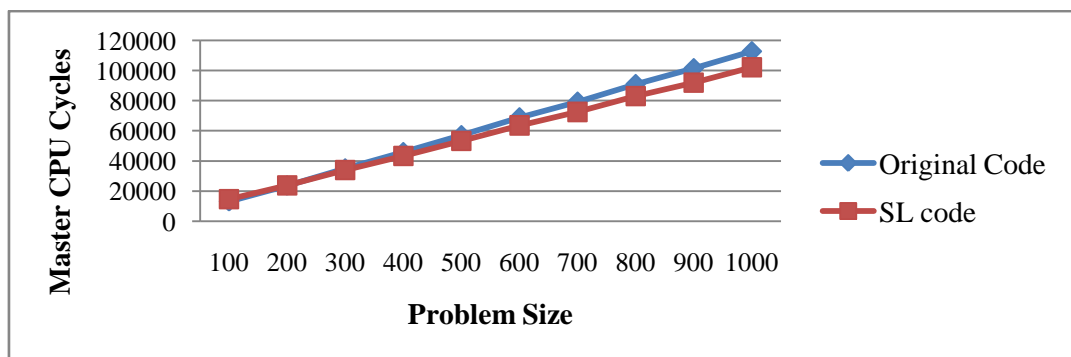
Figure 6.13. Comparing Sequential and SL Codes with 5 Dependences.

```
for (i=0;i<n-1;i++)
    a[i]=a[i+1];
```

Figure 6.14. A Typical Anti-Dependence Example.

Table 6.10. Comparing Sequential and SL Codes with an Anti-Dependence.

Problem Size	Original Loop	SL code	Speedup
100	11636	6420	1,812
200	21860	8036	2,720
300	30596	8992	3,403
400	40680	10756	3,782
500	49516	12336	4,014
600	59028	14592	4,045
700	68384	15572	4,391
800	77928	17192	4,533
900	87276	19348	4,511
1000	97020	21812	4,448
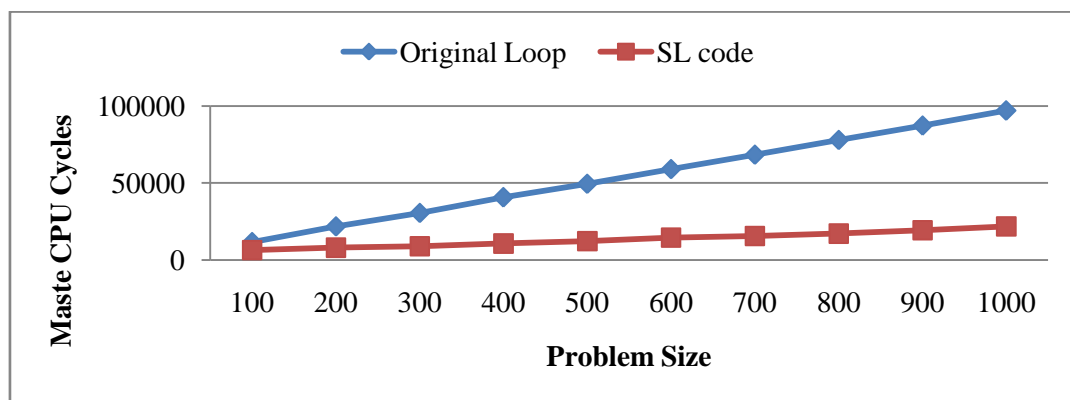
Figure 6.15. Comparing Sequential and SL Codes with an Anti-dependence.

6.3. Multi-Dimensional Loops

Evaluating Multi-Dimensional Loops (i.e. loop nestings) is a process which is further sub-categorized into two general cases: (i) Loops with no dependences and (ii) Loops with a dependence vector. These two sub-categories are treated completely differently by C2μTC/SL. The former is automatically translated as-is into a nesting of fully parallel families and relies on SVP to provide most of the efficiency-improving mechanisms. The latter is transformed into a self-adaptive algorithm trying to apply the dependence vector on the index space in order to discover the underlying hyperplane.

6.3.1. No Dependences

     In this sub-category, three real life applications were evaluated: (i) Conway's Game of Life, (ii) 2-Dimensional Matrix Multiplication and (iii) computation of the Mandelbrot Fractal. The overall evaluation of the data gained from these three examples will be presented at the end of this sub-chapter.

     A single pass of Conway's Game of Life [77] was implemented, transformed and evaluated. Tables 6.11 and 6.12 present the results of this simulation in CPU cycles while tables 6.13 and 6.14 do so in terms of speedup achieved. In both situations the size of the board is given as the length of one of its sizes (for every N, the board is a NxN array). Figures 6.16 and 6.17 visualize the data.

Table 6.11. The Results of the Game of Life in Absolute CPU Cycles.

Board Size	Original Code	SL code (1 core)	SL code (2 cores)	SL code (4 cores)
10	166488	93564	50152	32308
20	660108	366084	182004	94924
30	1467900	825020	426492	245984
40	2639280	1538716	744372	392088
50	4076948	2431700	1185480	726584
60	5872288	3465252	1668440	836252
70	7993908	4733720	2354340	1362200
80	10566976	6268056	3083648	1552796
90	12669380	7625316	3823188	2079436
100	16319956	9496204	4711040	2377932

Table 6.12. Continuation of the Results in Table 6.11.

Board Size	Original Code	SL code (8 cores)	SL code (16 cores)	SL code (32 cores)	SL code (64 cores)
10	166488	24820	17272	18636	21264
20	660108	61544	45460	30660	33200
30	1467900	152880	108776	89936	91524
40	2639280	300084	184092	163256	116716
50	4076948	498760	278788	192824	142540
60	5872288	581140	312780	222092	166564
70	7993908	797616	504940	339972	259640
80	10566976	990736	573208	400620	289568
90	12669380	1235200	686572	428492	319044
100	16319956	1472436	837760	564340	350828

Table 6.13. Speedups for the Game of Life Derived from Table 6.11.

Board Size	Speedup (1 core)	Speedup (2 cores)	Speedup (4 cores)
10	1,779	3,320	5,153
20	1,803	3,627	6,954
30	1,779	3,442	5,967
40	1,715	3,546	6,731
50	1,677	3,439	5,611
60	1,695	3,520	7,022
70	1,689	3,395	5,868
80	1,686	3,427	6,805
90	1,661	3,314	6,093
100	1,719	3,464	6,863

Table 6.14. Speedups Derived from Table 6.12.

Board Size	Speedup (8 cores)	Speedup (16 cores)	Speedup (32 cores)	Speedup (64 cores)
10	6,708	9,639	8,934	7,830
20	10,726	14,521	21,530	19,883
30	9,602	13,495	16,322	16,038
40	8,795	14,337	16,167	22,613
50	8,174	14,624	21,143	28,602
60	10,105	18,774	26,441	35,255
70	10,022	15,831	23,513	30,788
80	10,666	18,435	26,377	36,492
90	10,257	18,453	29,567	39,710
100	11,084	19,480	28,919	46,518
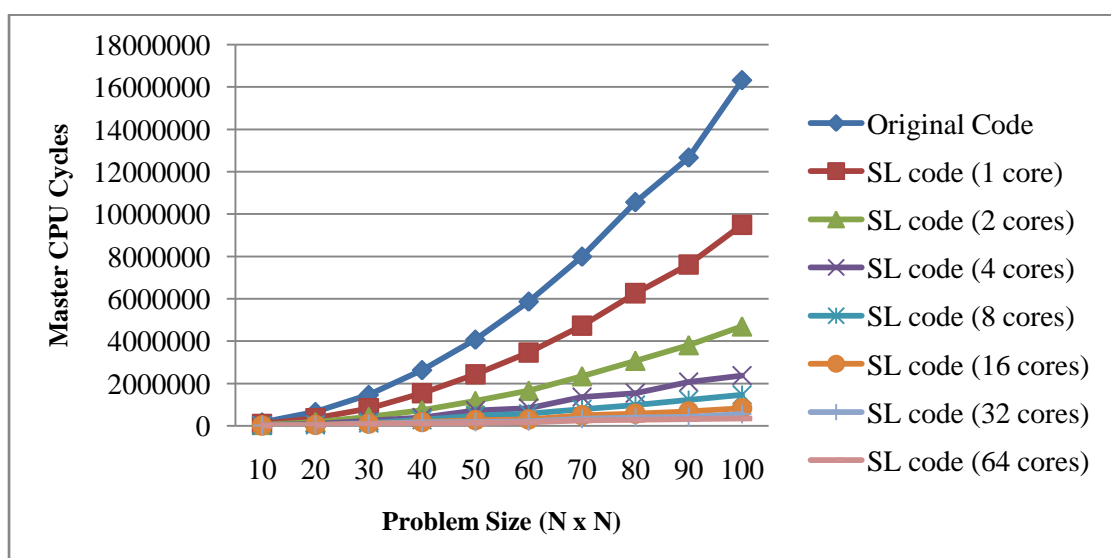
Figure 6.16. Comparing the Sequential and SL Codes for the Game of Life (Cycles).
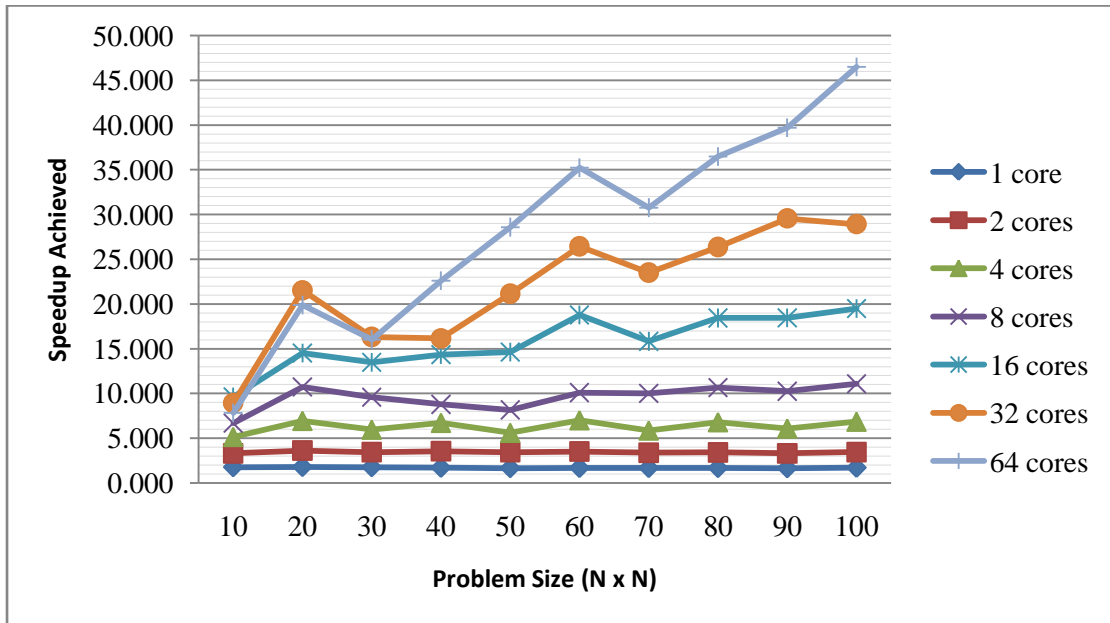
Figure 6.17. Comparing the Sequential and SL Codes for the Game of Life (Speedup).

There are plenty of algorithms which compute a variety of fractals. The Mandelbrot set is one of the most well known. For each pixel inside an area, its color is computed based on whether a repeating complex number remains bounded or not. A small variation of the one displayed in [78] was implemented, transformed and evaluated. Tables 6.15 and 6.16 present the results in absolute CPU cycles while Tables 6.17 and 6.18 present the relative speedups as a percentage. Figures 6.18 and 6.19 illustrate the corresponding visualization of the data.

Table 6.15. The Resulting Data of the Mandelbrot Calculation (1 to 4 cores).

Problem Size	Original Code	SL code (1 core)	SL code (2 cores)	SL code (4 cores)
10	16821836	6767880	3295252	3089844
20	67276200	26932780	12882604	7223820
30	151365640	69262156	34191264	22343312
40	269089788	131208400	65962844	37225076
50	420448960	210622756	105590812	64502364
60	605442832	303288432	159472340	75819036
70	824071716	412803456	213093056	119519364
80	1076336380	539117484	279323540	139472836
90	1362235208	682315352	351445964	218051464
100	1681769536	842362200	423456808	235388732

Table 6.16. The Resulting Data of the Mandelbrot Calculation (8 to 64 cores).

Problem Size	Original Code	SL code (8 cores)	SL code (16 cores)	SL code (32 cores)	SL code (64 cores)
10	16821836	2408284	2087464	2028192	2029608
20	67276200	4415792	2973392	1811400	1812764
30	151365640	14963196	11754664	10099072	10100592
40	269089788	22437260	19189784	19280972	13016172
50	420448960	39762104	21276524	21051632	19188528
60	605442832	47393724	27150204	24645572	23621364
70	824071716	69015716	37675340	24850712	22800804
80	1076336380	80823444	41800372	29247892	29238572
90	1362235208	108647180	54736592	28935624	37195796
100	1681769536	127280356	69986580	40990652	34929436

Table 6.17. Corresponding Speedups of the Mandelbrot calculation.

Problem Size	Speedup (1 core)	Speedup (2 cores)	Speedup (4 cores)
10	2,486	5,105	5,444
20	2,498	5,222	9,313
30	2,185	4,427	6,775
40	2,051	4,079	7,229
50	1,996	3,982	6,518
60	1,996	3,797	7,985
70	1,996	3,867	6,895
80	1,996	3,853	7,717
90	1,996	3,876	6,247
100	1,996	3,972	7,145

Table 6.18. Corresponding Speedups of the Mandelbrot Calculation (cont.).

Problem Size	Speedup (8 cores)	Speedup (16 cores)	Speedup (32 cores)	Speedup (64 cores)
10	6,985	8,059	8,294	8,288
20	15,235	22,626	37,140	37,112
30	10,116	12,877	14,988	14,986
40	11,993	14,023	13,956	20,673
50	10,574	19,761	19,972	21,911
60	12,775	22,300	24,566	25,631
70	11,940	21,873	33,161	36,142
80	13,317	25,749	36,800	36,812
90	12,538	24,887	47,078	36,623
100	13,213	24,030	41,028	48,148
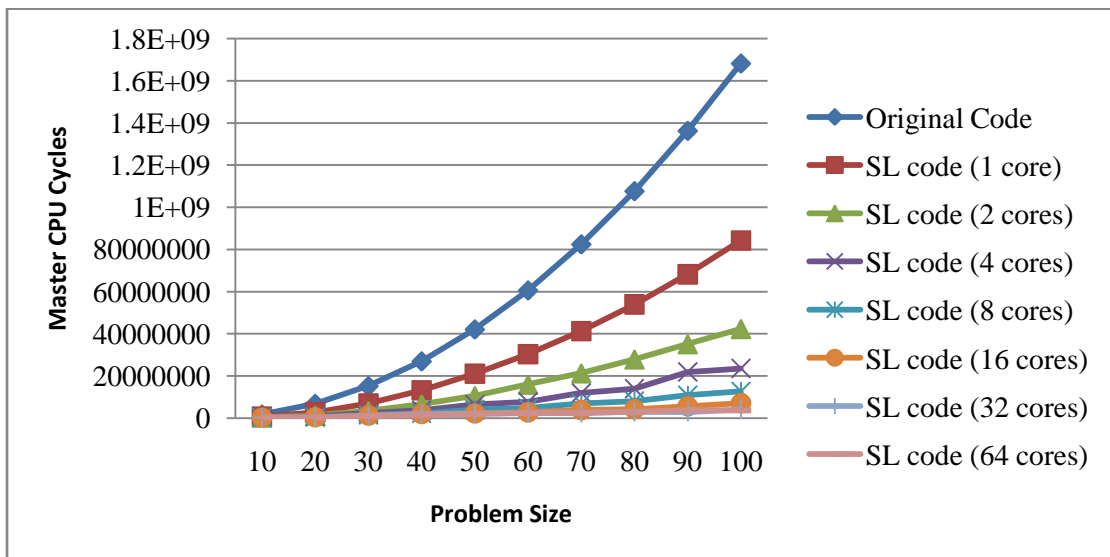
Figure 6.18. The Resulting Data of the Mandelbrot Calculation (CPU cycles).
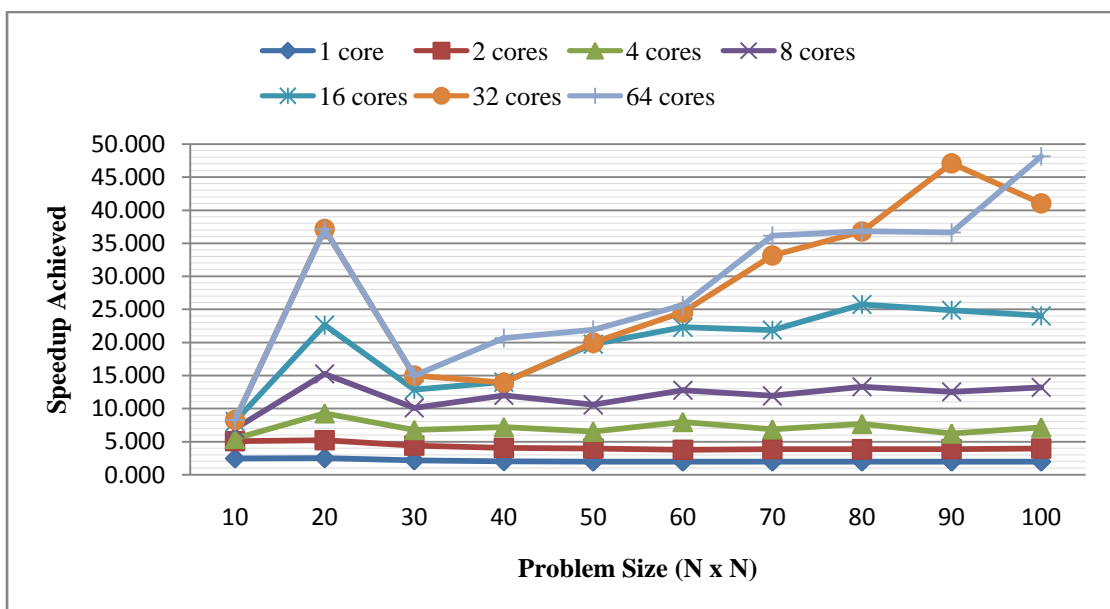
Figure 6.19.The Corresponding Speedups of the Mandelbrot Calculation.

Finally, matrix multiplication was implemented and evaluated. Tables 6.19 and 6.20 illustrate the results in CPU cycles while Tables 6.21 and 6.22 feature the speedups gained. Figures 6.20 and 6.21 visualize the results.

Table 6.19. CPU Cycles for the Sequential and Parallel Executions of Matrix Multiplication.

Problem Size	Original Code	SL code (1 core)	SL code (2 cores)	SL code (4 cores)
10	93960	63756	31784	30380
20	701324	329852	182024	98508
30	2393432	1413860	948320	616752
40	6084044	2871192	1918756	1520672
50	11734632	4779272	4293760	2936856
60	22983224	8535988	6758848	4722192
70	35755880	12419688	9468416	6822220
80	56068460	19500748	12543716	9817144
90	80046760	27364724	16010600	14203676
100	111533084	35371188	17714700	16862152

Table 6.20. Continuation of the Results from Table 6.19.

Problem Size	Original Code	SL code (8 cores)	SL code (16 cores)	SL code (32 cores)	SL code (64 cores)
10	93960	27096	24800	25392	26668
20	701324	60148	46604	35036	36152
30	2393432	382776	267008	164556	166160
40	6084044	1472944	745260	909824	763284
50	11734632	1654540	1682060	1535856	1286524
60	22983224	3135788	2452056	2620348	2181996
70	35755880	5108300	2850672	2997772	3358756
80	56068460	5781756	3957952	3732792	3488408
90	80046760	9946904	5046460	5418156	4042052
100	111533084	11741328	6581304	5167396	5848620

Table 6.21. Corresponding Speedups Gained from Parallel Matrix Multiplication.

Problem Size	Speedup (1 core)	Speedup (2 cores)	Speedup (4 cores)
10	1,474	2,956	3,093
20	2,126	3,853	7,119
30	1,693	2,524	3,881
40	2,119	3,171	4,001
50	2,455	2,733	3,996
60	2,693	3,400	4,867
70	2,879	3,776	5,241
80	2,875	4,470	5,711
90	2,925	5,000	5,636
100	3,153	6,296	6,614

Table 6.22. Corresponding Speedups from Matrix Multiplication (cont.).

Problem Size	Speedup (8 cores)	Speedup (16 cores)	Speedup (32 cores)	Speedup (64 cores)
10	3,468	3,789	3,700	3,523
20	11,660	15,049	20,017	19,399
30	6,253	8,964	14,545	14,404
40	4,131	8,164	6,687	7,971
50	7,092	6,976	7,640	9,121
60	7,329	9,373	8,771	10,533
70	7,000	12,543	11,927	10,646
80	9,697	14,166	15,021	16,073
90	8,047	15,862	14,774	19,803
100	9,499	16,947	21,584	19,070
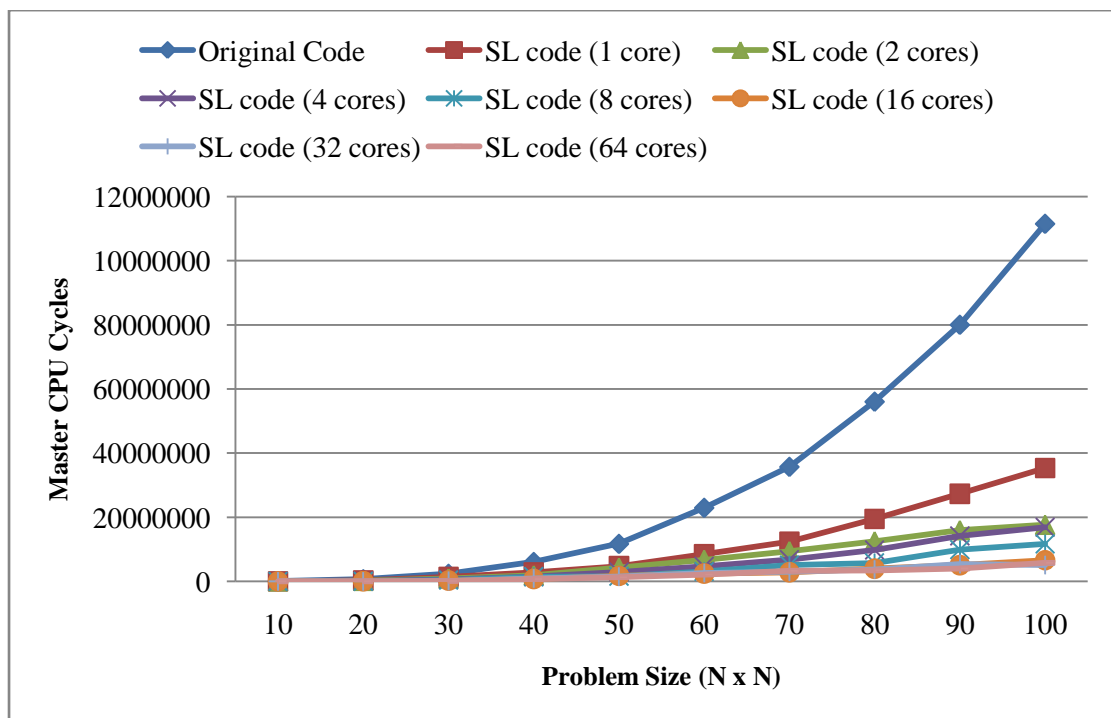
Figure 6.20. Comparing Sequential and Parallel Matrix Multiplications (Cycles).
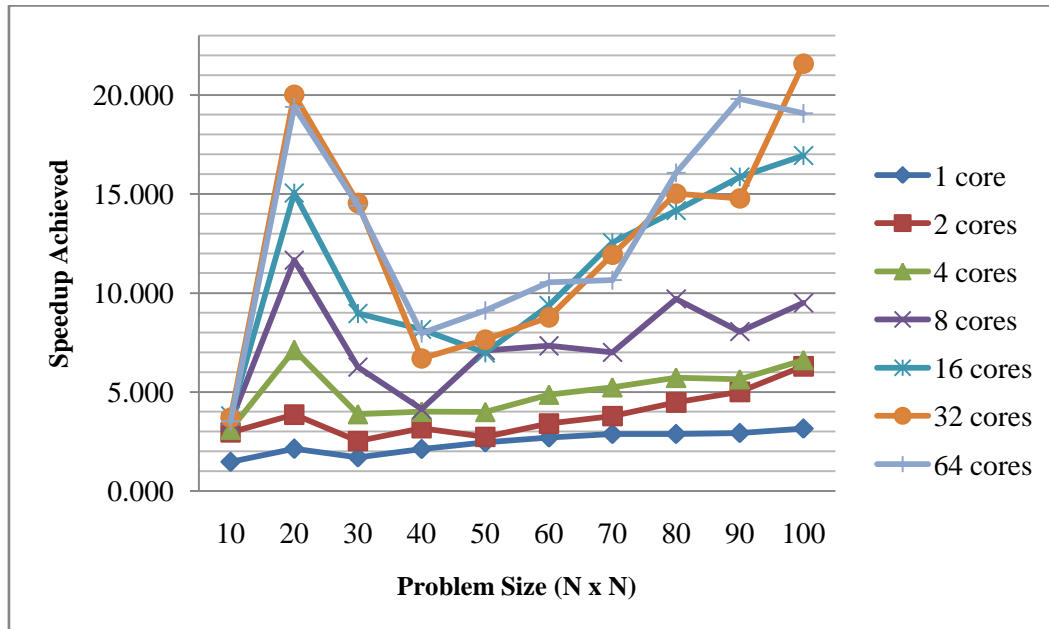
Figure 6.21. Comparing Parallel Matrix Multiplications (Speedups).

Finally we tested the Game of Life for various larger problem sizes but tiled with various tile sizes (1,2,4,8,16,32,64 and 128). The idea behind this was to test SVP's throughput while being oversaturated and how applying the tiling method helps alleviate it. Table 6.23 presents the results. Increasing the tile size certainly reduces the overall cycles needed however this effect works until a point. After that size the overall parallelism exposed becomes smaller due to the very large tile sizes.

Table 6.23. MasterCPU Cycles for the Game of Life for Various Problem and Tile Sizes.

		Problem Size				
		1000	2000	3000	4000	5000
Tile Size	1	183105732	728808580	1624787896	2924394636	4721679540
	2	138375388	545503468	1205679672	2198231048	3674882208
	4	129602680	522952608	1144788964	2078213420	3492057336
	8	136373840	507765180	1102716796	2006374452	3275702220
	16	159121700	560260192	1143944816	2069512704	3267163724
	32	149057836	639955024	1275310076	2224032312	3426920124
	64	116219276	594489528	1504648036	2544091388	3905988888

There are several conclusions that can be reached by the data obtained from the three aforementioned applications:

(i) Even one SVP core can increase efficiency substantially in a fully parallel program. This indicates the SVP's ability to speed up a loop even in a sequential environment (One core does not offer actual parallel execution). This ability is the result of a combination of SVP's characteristics: (a) High memory latency tolerance: memory access instructions are overlapped with the rest of the operation in order to eliminate idle time and (b) hardware control of thread iterations. There is no need for the software to check and branch depending on the index value per iteration.

(ii) The greater the number of cores in a system, the better the results. However the system becomes oversaturated when there is an excessively high number of cores in existence since much time is lost in communication overheads in the memory network, especially when it comes to memory store instructions just prior to synchronization. There are no other operations to overlap with these memory instructions and so there is no latency tolerance to take advantage of. This effect can be alleviated by tiling the index space and exposing parallelism on an inter-tile basis.

(iii) The overall speedup increases with the problem size for any number of cores in the system. Since the family creation overheads remain the same, increasing the problem size results in those overheads offering less and less percentage in the whole execution time. Reducing the overhead of family management in addition to having more threads and hence greater memory latencies tolerance leads to improved efficiency altogether.

6.3.2. The Run-Time Algorithm

Perfectly nested loops with a dependence vector belong in this category. In order to evaluate the efficiency of the Self-Adaptive algorithm employed by C2μTC/SL, just the speed-up gained was not enough. There remained two questions: (i) how close to the optimal result the Self-Adaptive method can get and (ii) How does it fare compared to a compile-time method. The optimal goal is the highest speedup that can be achieved by a tile-based run-time method utilizing a scheduler thread. To answer these questions, the optimal result of the fixed-size algorithm was calculated. The reasoning behind this choice is two-fold:

(i) Generally it is trivial, albeit time consuming, to find the optimal result. The algorithm is executed multiple times with varying tile sizes and the best result is considered optimal. There are many local optima in such a case and that's why it's not enough to just stop once the first peak is reached. Figure 6.22. demonstrates the speedups gained by the fixed-size algorithm for a two-dimensional loop of size 4000 x 4000 with a dependence vector of D={(1,0), (0,1)}. This figure illustrates that even though the speedups follow the trend line in the middle, they alternate above and below that line constantly in a rather jaggy manner.

(ii) The fixed-size algorithm bears a great resemblance to the Self-Adaptive one, while being a bit simpler both conceptually and programmatically. Thusly it serves as a target for the results that the Self-Adaptive algorithm can offer.

Due to these two reasons, the target goal for the Self-Adaptive algorithm is roughly the optimal result of the Fixed-Size algorithm, gained by repeated execution of different tile sizes. That optimal result per problem size is subsequently compared to a compile time algorithm. The method used is the skewed loop described in Figures 3.20 and 3.21.
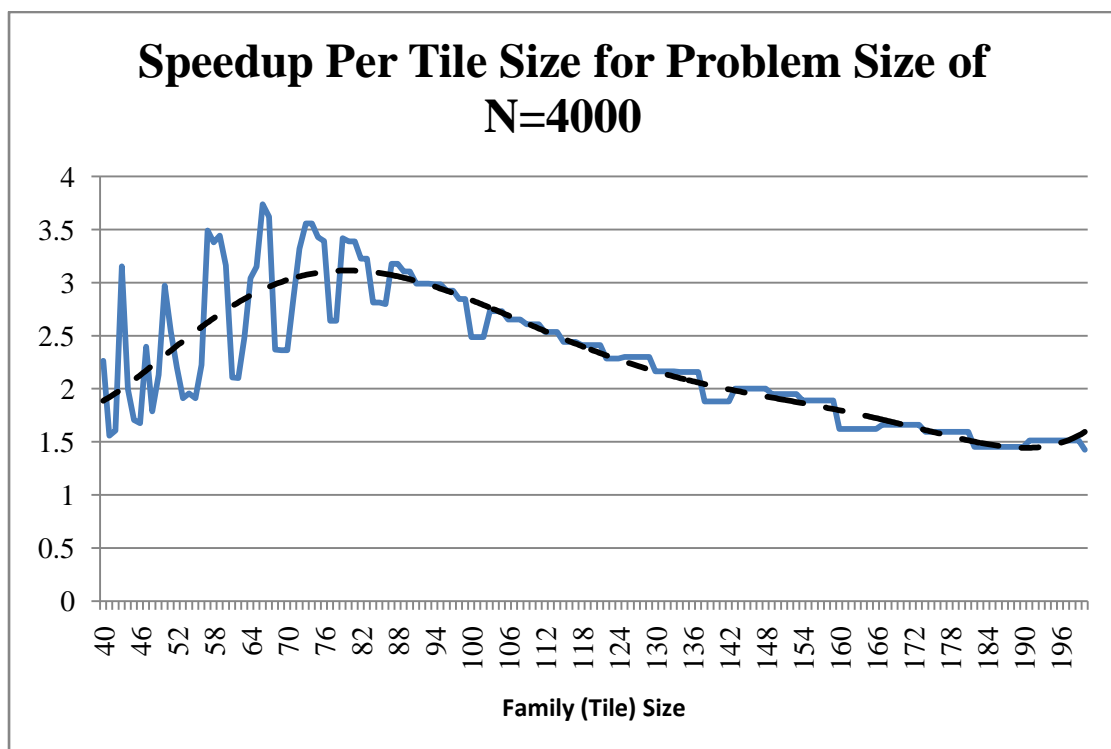
Figure 6.22. Speedups gained for the problem of D={(1,0),(0,1)} with a grid size of 4000x4000 and various tile sizes. The dashed line indicates the inferred trend.

The innermost loop in such a case can be fully parallelized. In order for the comparison to be proper, the hyperplane method (compile-time) was implemented manually in SL and was simulated over SVP. Comparison results are displayed in Table 6.24 and visualized in Figure 6.23. Table 6.25 presents the different speedups gained by the two different methods. Finally, Table 6.26 demonstrates the optimal tile size picked per problem.

It is worth noting that even though the Hyperplane method was implemented in the finest of granularities possible (1 thread per iteration), it offers a speedup of 5,392 which is much higher than the 1,737 gained by the fixed size algorithm. This difference, however, is alleviated as the problem size increases. Table 6.27 shows the results obtained from executing the algorithms with a problem size of 4000x4000 (N=4000). At this size, the fine-grain hyperplane method becomes oversaturated and its speedup is worse than the fixed-size method. Of course, as has already been demonstrated the compile-time method can be improved by applying tiling on it. However what kind of tile size to be used is unknown. Table 6.28 demonstrates the speedup gained from the tiled version of the compile-time version of the hyperplane for various tile sizes.

It is clear that the compile-time method outperforms the self-adaptive run-time algorithm by various degrees depending on the tile size. However choosing a proper tile size is an impossible task since, as in the run-time method, too small or too large a size has an adverse effect on the efficiency. In addition, the self-adaptive method offers a series of other advantages: (i) the run time algorithm does away with the need to solve any NP-Complete problem, (ii) it can work with index spaces of irregular shapes (e.g. triangular spaces) and (iii) the size of the tile is not necessary to be decided before execution, usually by estimations (or by extensive repetitions in the fixed-size algorithm's case). Finally there is no standard way to calculate a proper compile-time transformation due to the complexity of the NP-Complete problem. This compile-time method presented here is an idealized method just for comparisons and interpretations.
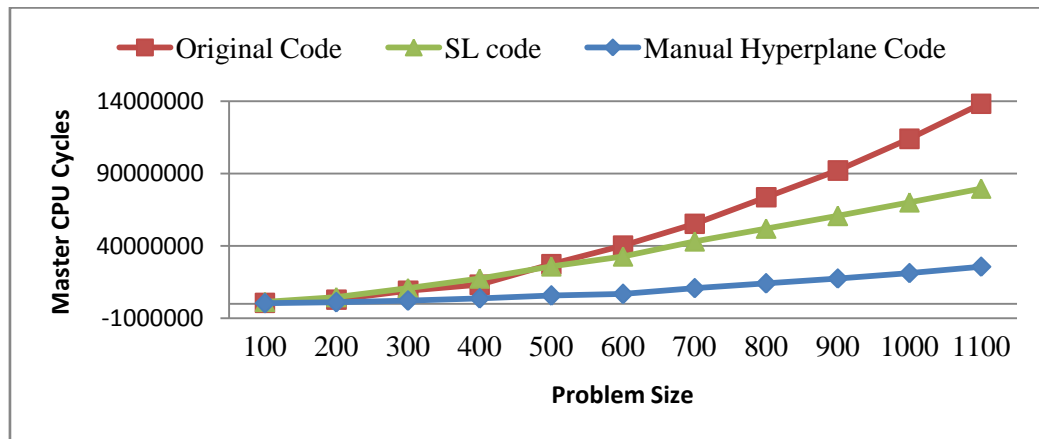
Figure 6.23. Comparing cycles between original, SL and manual hyperplane codes.

Table 6.24. Comparing execution times between sequential, transformed and manually written parallel code.

Problem Size	Original Code	SL code	Manual Code
100	731040	1314584	344800
200	2927336	4642112	1044456
300	9062496	10801192	2131344
400	13201516	17474616	3664328
500	27320340	25940796	5632024
600	40170528	32808312	6752076
700	55277512	43149216	10869036
800	73727150	51961456	14040020
900	92176788	60744780	17445424
1000	114073668	70149652	21268804
1100	138258288	79585768	25639544

Table 6.25. Speedups Gained from the two methods for various problem sizes.

Problem Size	Speedup SL	Speedup Hyperplane
100	0,556	2,120
200	0,631	2,803
300	0,839	4,252
400	0,755	3,603
500	1,053	4,851
600	1,224	5,949
700	1,281	5,086
800	1,419	5,251
900	1,517	5,284
1000	1,626	5,363
1100	1,737	5,392

Table 6.26. The Optimal Tile Size for Various Problem Sizes.

Problem Size	Optimal Tile Size
100	5
200	5
300	6
400	7
500	30
600	30
700	31
800	39
900	41
1000	42
1100	43

Table 6.27. Speedups for Problem Size of (2, 3, 4)000x (2, 3, 4)000 for the Loop With Dependence Vector D={(1,0), (0,1)}

Problem Size	Speedup SL	Speedup Hyperplane
2000	2,654	5,107
3000	2,745	3,887
4000	3,739	3,187

Table 6.28. CPU Cycles and Speedup Gained for Various Tile Sizes for the Compile-time Hyperplane Method (Problem size: 4000 x 4000).

Tile Size	Master CPU Cycles	Speedup
1	592101356	3,111
2	287120992	6,416
4	277922988	6,628
8	265680920	6,934
16	276058308	6,673
32	173013796	10,64
64	139595836	13,19
128	142822016	12,89

The Self-Adaptive method is next compared with the Fixed-size one in three problems with different dependence vectors: (i) D={(1,0), (0,1)}, (ii) D={(0,1), (1,1), (1,0), (1,-1)} and (iii) D={(2,0), (0,2)}. Each of these problems has a different

characteristic. The first is a typical example, the second is augmented with two more dependences and finally the third offers more parallelism by letting two different columns execute simultaneously at any time.

As far as the first problem is concerned, Table 6.29 illustrates the results of the two methods in CPU cycles, while Table 6.30 illustrates the speedups offered by each algorithm. Figures 6.24 and 6.25 provide a graphic representation of the data. It is clear that the Self-Adaptive algorithm not only reached the Fixed-Size algorithm's efficiency levels, in some cases it slightly surpassed it. It fares a lot worse in smaller problem sizes due to two different situations: (i) as previously discussed, larger problem sizes reduce the percentage of the overall overheads in the total execution time. Consequently, small problems don't amortize the overheads enough and (ii) there simply is not enough time for the algorithm to reach a conclusion about the proper tile size and that results in the lower levels of efficiency illustrated in Table 6.30 and Figure 6.25. However, with enough time (in greater problem sizes), the algorithm not only results in finding an optimal size, it also makes up for its slow start.

The second problem (loop) has a dependence vector of D={(0,1), (1,1), (1,0), (1,-1)}. Figure 6.26 demonstrates the actual loop while Figure 6.27 visualizes the full dependence vector in an index space. Table 6.31 demonstrates the results in CPU cycles and Table 6.32 presents the speedups offered by the two run-time algorithms.

Table 6.29. Comparing the Fixed-Size algorithm with the Self-Adaptive one for the {(1,0),(0,1)} Problem.

Problem Size	Original Code	Fixed Size code	Self Adaptive Code
100	731040	1314584	1843368
200	2927336	4642112	6938972
300	9062496	10801192	13052524
400	13201516	17474616	19550652
500	27320340	25940796	24905396
600	40170528	32808312	34408864
700	55277512	43149216	40316528
800	73727150	51961456	49048212
900	92176788	60744780	55648768
1000	114073668	70149652	66257132

Table 6.30. Comparing the Speedups of the two Methods for the {(1,0),(0,1)} Problem.

Problem Size	Fixed Size Speedup	Self Adaptive Speedup
100	0,556	0,397
200	0,631	0,422
300	0,839	0,694
400	0,755	0,675
500	1,053	1,097
600	1,224	1,167
700	1,281	1,371
800	1,419	1,503
900	1,517	1,656
1000	1,626	1,722
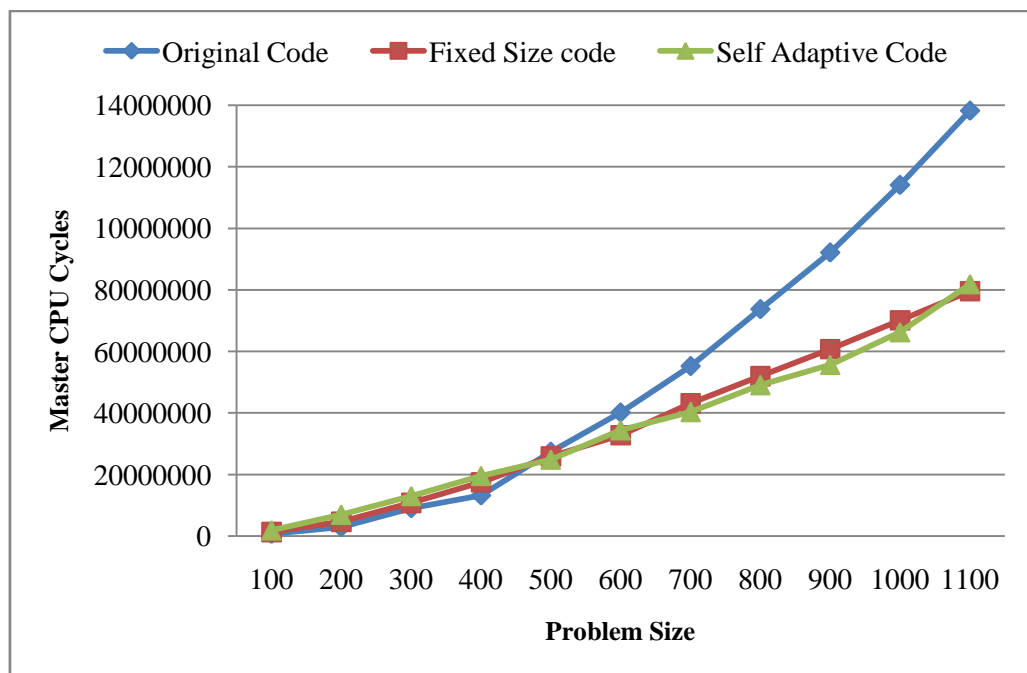
Figure 6.24. Comparing the Fixed-Size algorithm with the Self-Adaptive one for the D={(1,0),(0,1)} Problem.
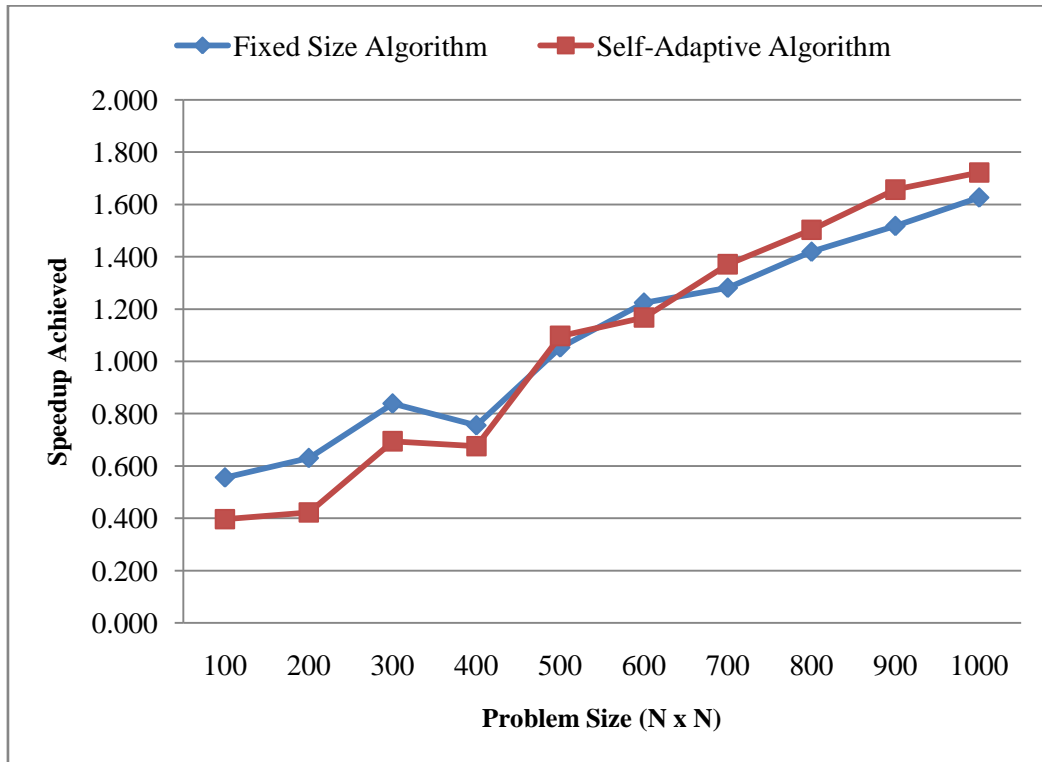
Figure 6.25. Comparing the Speedups of the two Run-time Methods for the
D={(1,0),(0,1)} Problem.

```
for (i=1;i<n-1;i++)
        for (j=1;j<n-1;j++)
                A[i][j]=A[i][j-1]+A[i-1][j-1]+A[i-1][j]+A[i-1][j+1];
```

Figure 6.26. The Second Loop Nesting Under Evaluation. The Dependence Vector is
D={(0,1), (1,1), (1,0), (1,-1)}
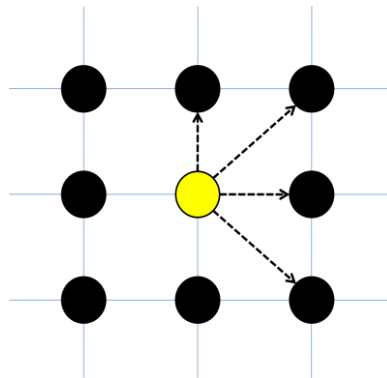
Figure 6.27. Visualization of the Dependence vector in the 2-D index space.

Table 6.31. Comparing the Resulting Data of the Two Run-time Algorithms for the
D={(0,1), (1,1), (1,0), (1,-1)} Problem.

Problem Size	Original Code	Fixed Size Algorithm	Self Adaptive Algorithm
100	1069512	1955432	2628388
200	4339096	6490840	9814264
300	10227620	14517088	22626908
400	19030496	23745432	37218680
500	30405836	38530916	46855584
600	44332336	54702360	69131064
700	60793824	72207180	100495712
800	80959556	87373276	102670936
900	101352808	110178980	131357372
1000	125461936	137689188	148974992
1100	152135436	160506780	177237064
1200	181264208	177854256	198504028
1300	213046836	189210796	183085240
1400	247357888	218266376	231859332
1500	284153852	249275952	238125204
1600	328077396	263238604	304201800
1700	365474704	319817604	316397112
1800	409992824	336841140	328592424

Table 6.32. Comparing the Speedups of the Two Run-time Algorithms for the Loop
with D={(0,1), (1,1), (1,0), (1,-1)}

Problem Size	Fixed Size Speedup	Self Adaptive Speedup	Problem Size	Fixed Size Speedup	Self Adaptive Speedup
100	0,547	0,407	1000	0,911	0,842
200	0,668	0,442	1100	0,948	0,858
300	0,705	0,452	1200	1,019	0,913
400	0,801	0,511	1300	1,126	1,164
500	0,789	0,649	1400	1,133	1,067
600	0,810	0,641	1500	1,140	1,193
700	0,842	0,605	1600	1,246	1,078
800	0,927	0,789	1700	1,143	1,155
900	0,920	0,772	1800	1,217	1,248

Figures 6.28 and 6.29 visualize the comparisons. The fact that there is less
parallelism to exploit is indicated by the inability of both algorithms' to offer any
significant speedup until the problem size of between N=1200 and N=1300. This size
is much larger than the problem of D={(1,0), (0,1)}. Once again though, the self-

adaptive variant quickly catches up and follows the fixed-size one after a while. Again, this proves the effectiveness of the adaptive algorithm.
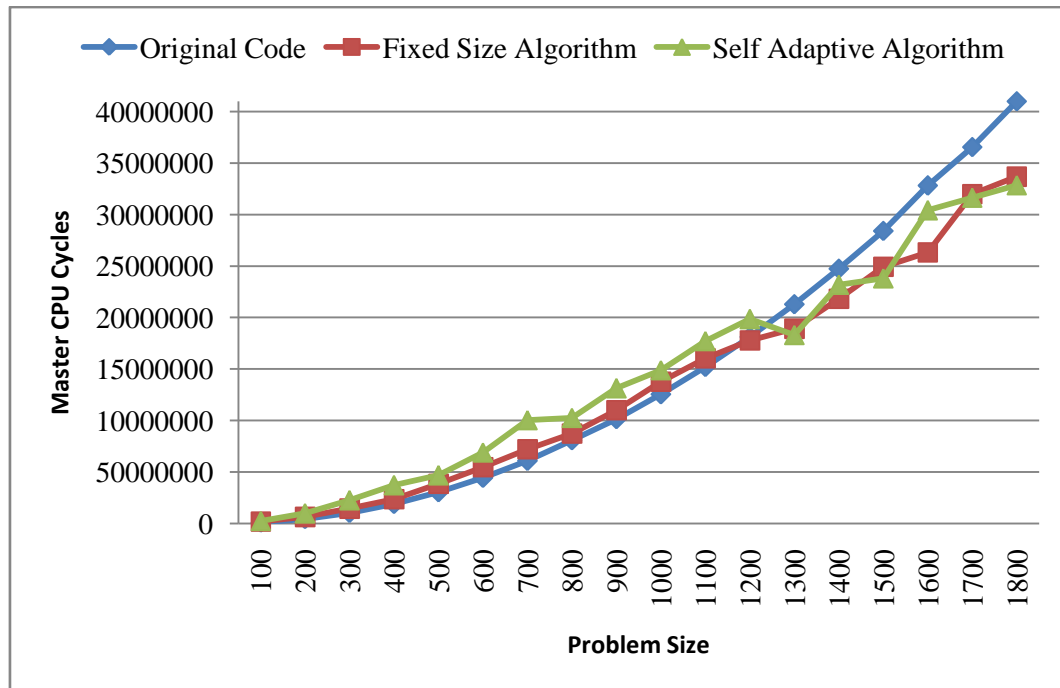
Figure 6.28. Comparing the CPU Cycles of the two Run-time Algorithms for the Loop with D={(0,1), (1,1), (1,0), (1,-1)}
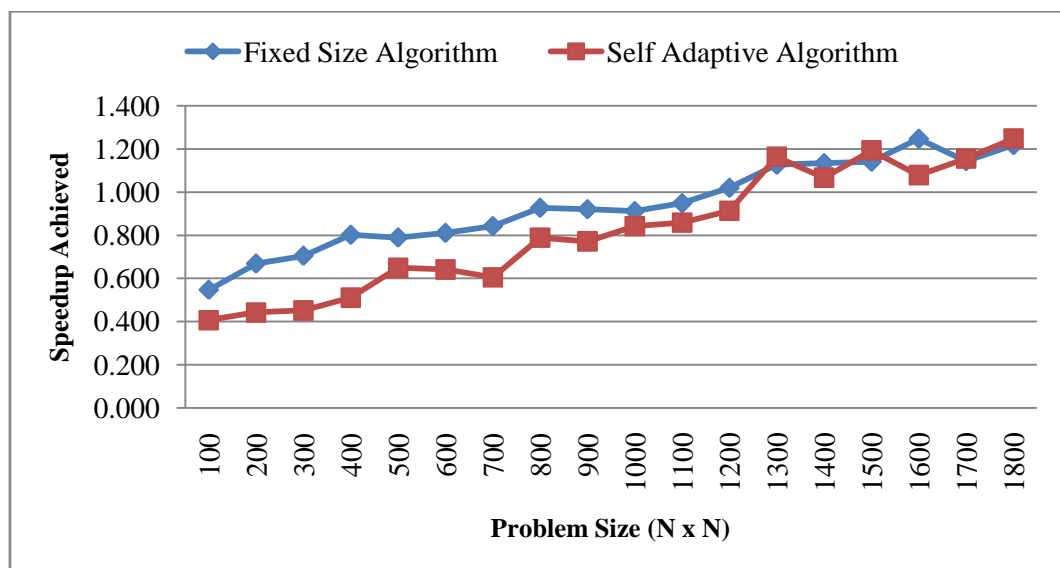
Figure 6.29. Comparing the Speedups of the two Run-time Algorithms for the Loop with D={(0,1), (1,1), (1,0), (1,-1)}

The final problem is the loop nesting with a dependence vector of D={(2,0), (0,2)}. The dependency of (0,2) is actually internally treated as (0,1) by both algorithms as has been already mentioned. The (2,0) dependency however allows two simultaneous columns to execute at any given time, effectively doubling the amount of parallelism that can be exploited. Table 6.33 displays the results in CPU cycles and Table 6.34 displays the speedups achieved for each problem size. Figures 6.30 and 6.31 help visualize the data.

Table 6.33. CPU Cycles for the {(2,0), (0,2)} Problem.

Problem Size	Original Code	Fixed Size Algorithm	Self Adaptive Algorithm
100	774108	516520	1698932
200	3771708	1735864	4127672
300	9148748	6529064	7743552
400	16099372	11497204	11801488
500	25195468	17515188	15020296
600	36275752	24898940	19725720
700	49041688	32041456	25317360
800	64890712	40364832	29403340
900	85394952	49834420	36329496
1000	111067292	58711484	40422532

Table 6.34. Speedups Achieved by the two Algorithms.

Problem Size	Fixed Size Speedup	Speedup Self Adaptive
100	1,499	0,456
200	2,173	0,914
300	1,401	1,181
400	1,400	1,364
500	1,438	1,677
600	1,457	1,839
700	1,531	1,937
800	1,608	2,207
900	1,714	2,351
1000	1,892	2,748
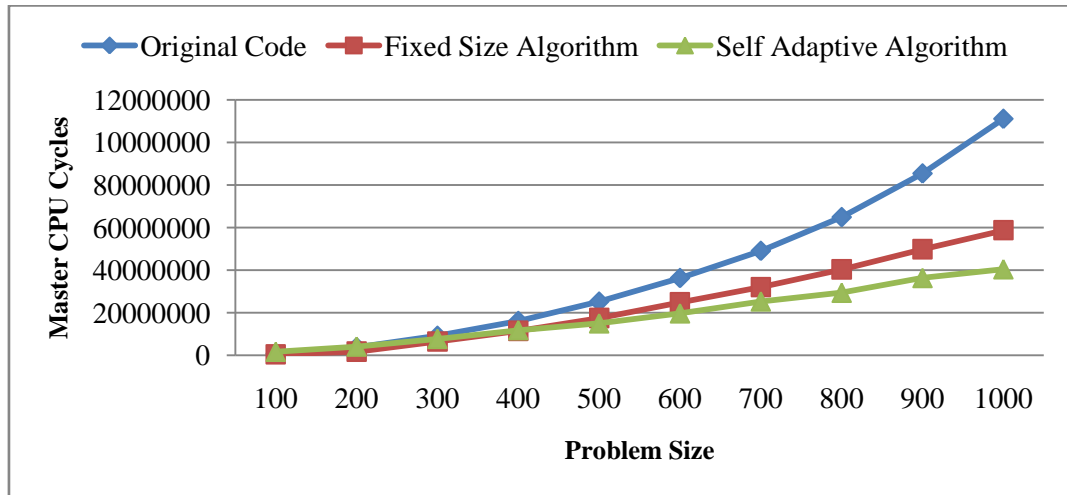
Figure 6.30. CPU Cycles for the {(2,0), (0,2)} Problem.
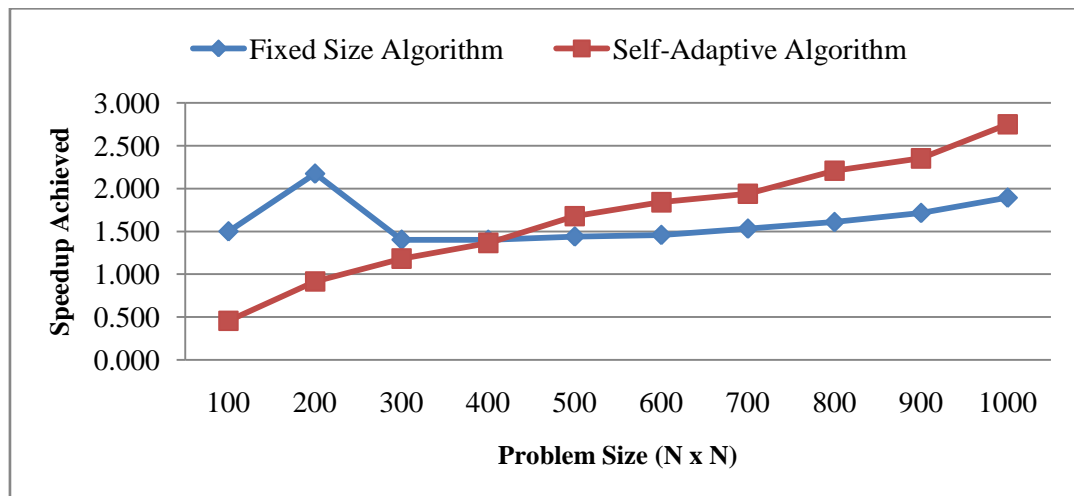
Figure 6.31. Comparing the Speedups Achieved by the two Algorithms for the
D={(2,0), (0,2)} Problem.

In this case the Self-Adaptive algorithm fared much better and closer to the ideal target of a doubling the speedup of the D={(1,0), (0,1)} problem than the fixed size method. This can be explained by the fact that the real optimal tile size for the fixed size algorithm was outside the range of the numbers that were tried (2 to 40) and thusly was lost.

6.4. The Livermore Loops

As a final evaluation test a suite of programs was needed where each test is more complex than just simple perfectly nested loops. For that reason, the Livermore loops [79] were chosen. It is a set of 24 *kernels*, each performing a particular task. The suite was originally created to test parallelizing / vectorizing compilers so it was selected to benchmark C2μTC/SL. It should be noted that, by definition, not all of the kernels can be parallelized in the first place. A list of the kernels follows detailing how C2μTC/SL fared against each of them. For each kernel there is an indication whether C2μTC/SL did a proper transformation that increases efficiency/exposes parallelism (PASS) while if it decided to err on the safe side and just transformed the kernel into a sequentially executing family of threads(which can slightly increase efficiency as well) (SAFE). There is a third result in some of the kernels called FAIL. This happens when the code contains C constructs not included in the C subset that C2μTC/SL supports (like goto). Instead of the compiler stopping at detection of those constructs and not producing any output, instead it just proceeds to create some output that is completely wrong and will not even compile properly.

• **Hydrodynamics fragment**: The loop is rather simple and can be fully executed in parallel. C2μTC/SL automatically parallelized it correctly (*PASS*).

• **Incomplete Cholesky conjugate gradient**: The code is rather convoluted and C2μTC/SL cannot distinguish any hidden parallelism to exploit (no proper meaning can be extracted from some variables and an existing dependence is not static). However, the whole loop is transformed into an infinite family of threads which contains another family of threads with shared variables. As has been demonstrated before this can increase efficiency by a small percentage (*SAFE*).

• **Inner product**: This is a similar code to the innermost loop of matrix multiplication. If the partial sums are first calculated in a temporary variable by hand and then added to the accumulator variable, efficiency can be greatly sped up by taking advantage of all the threads calculating their sum in parallel before locking down on the shared channel (*PASS*).

• **Banded linear systems solution**: C2μTC/SL cannot detect any meaningful parallelism (there is a relationship which cannot be statically identified as a

dependence or an antidependence) in that code so it transforms both loops to families of threads with shared variables (one of the loops is accumulating a value to a variable) to increase efficiency slightly (*SAFE*).

- **Tridiagonal linear systems solution**: This is a normal single-dimensional loop with a unary dependency of length 1. C2μTC/SL acts appropriately (*PASS*).

- **General linear recurrence equations**: Once again C2μTC/SL is unable to perform a meaningful transformation (the existence of non-static dependences prevents such an action) so it resorts to transform each loop into a family in order to gain some efficiency (*SAFE*).

- **Equation of state fragment**: Although it seems like a complicated loop, it is in fact rather simple and it can be computed fully in parallel. C2μTC/SL provides the correct transformation (*PASS*).

- **Alternating direction implicit integration**: This loop is too complicated for the compiler to "understand" so it fails (*FAIL*).

- **Integrate predictors**: A fully parallel loop where each iteration writes some value at the first column of the appropriate row. There are no dependences and C2μTC/SL performs the proper parallel transformation (*PASS*).

- **Difference predictors**: Another fully parallel loop which is properly transformed by C2μTC/SL (*PASS*).

- **First sum**: A single dimensional loop with a unary dependency of length 1. A synchronizing channel is utilized to provide sequential execution and better efficiency (*PASS*).

- **First difference**: An obviously fully parallel loop which is transformed in an appropriate manner by C2μTC/SL (*PASS*).

- **2-D particle in a cell**: An overly complex loop where C2μTC/SL fails to detect any parallelism. The whole loop is transformed into a sequentially executed family of threads (*SAFE*).

- **1-D particle in a cell**: This loop is comprised of 3 smaller loops. The first of them is fully parallel and is understood as such by C2μTC/SL. The remaining loops for various reasons are transformed into sequentially executing families of threads (*PASS/SAFE/SAFE*).

• **Casual Fortran**: This loop is considered too complicated by C2μTC/SL. It is transformed into a sequentially executing family of threads (*SAFE*).

• **Monte Carlo search**: The loop is so complicated (with the use of "goto" aggravating the complexity) that C2μTC/SL fails to produce any meaningful code (*FAIL*).

• **Implicit conditional computation**: Again another loop too complex for C2μTC/SL to produce correct code ("goto" is again present) (*FAIL*).

• **2-D explicit hydrodynamics fragment**: A loop comprised of 3 others but all of them are fully parallel which C2μTC/SL understands as such and acts accordingly (*PASS*).

• **General linear recurrence equations**: This loop is comprised of 2 smaller loops. Each of those two loops carries a shared variable in the code. C2μTC/SL understands this and produces two sequentially executing families with a synchronizing channel for the shared variable (*PASS*).

• **Discrete ordinates transport**: C2μTC/SL is unable to detect any parallelism (cross dependences are not handled) or variables to use as shared so it takes the safe approach and transforms the entire loop into a sequentially executing family (*SAFE*).

• **Matrix-matrix product**: A fully parallel loop nesting which C2μTC/SL correctly identifies and transforms (*PASS*).

• **Planckian distribution**: Another fully parallel loop which C2μTC/SL understands properly and produces a correct transformed output (*PASS*).

• **2-D implicit hydrodynamics fragment**: This loop contains both two-dimensional anti-dependences and dependences. By reversing the direction of the anti-dependences (and essentially turn them into dependences), the loop is transformed into a two-dimensional nesting with a dependence vector. C2μTC/SL invokes the Self-Adaptive algorithm for this loop and produces a correct transformation (*PASS*).

• **Location of a first array minimum**: This is a simple loop which cannot be parallelized. C2μTC/SL correctly identifies that the current minimum index variable used in its iteration is a shared variable and transforms the loop accordingly (*PASS*).

Table 6.35. A Summary of the Results of the Livermore Loops Transformations by C2µTC/SL.

Kernel No.	Kernel Name	Result
1	Hydrodynamics fragment	PASS
2	Incomplete Cholesky conjugate gradient	SAFE
3	Inner product	PASS
4	Banded linear systems solution	SAFE
5	Tridiagonal linear systems solution	PASS
6	General linear recurrence equations	SAFE
7	Equation of state fragment	PASS
8	Alternating direction implicit integration	FAIL
9	Integrate predictors	PASS
10	Difference predictors	PASS
11	First sum	PASS
12	First difference	PASS
13	2-D particle in a cell	SAFE
14	1-D particle in a cell	PASS/SAFE/SAFE
15	Casual Fortran	SAFE
16	Monte Carlo search	FAIL
17	Implicit conditional computation	FAIL
18	2-D explicit hydrodynamics fragment	PASS
19	General linear recurrence equations	PASS
20	Discrete ordinates transport	SAFE
21	Matrix-matrix product	PASS
22	Planckian distribution	PASS
23	2-D implicit hydrodynamics fragment	PASS
24	Location of a first array minimum.	PASS

Table 6.35 summarizes the results for all the Livermore loops. Qualitatively, more than half of the loops are transformed properly and most of the rest are transformed into some sort of family which produces correct results. Due to this, C2µTC/SL should be considered relatively successful in its task. However it is obvious that it needs a better symbolic analyzer in order to properly "understand" more complex codes (i.e. codes where index accesses take place via pointer dereferencing, codes where index accesses contain regular expressions etc.)

CHAPTER 7. FINAL THOUGHTS

This paper presented the most basic elements and ideas regarding the automatic parallelization of legacy sequential code. In addition, it described the research on what –at the time of writing- was considered novel: Using the SVP model to parallelize loops in ways that mainstream compilers could not. This research led to the creation of C2μTC/SL compiler. Heavily in beta, C2μTC/SL served more as a vessel to perform research than a commodity (or even commercial) compiler system that would be available to the public. The beta aspect reflects upon almost all aspects of the compiler in the form of a series of limitations:

(i) C2μTC/SL only compiles programs with a main() function and no other functions in the same program. That means that the all functions must be declared as external and linked against the transformed code during compilation phase. In addition all external functions must have a return value (they cannot be declared as void).

(ii) Due to some issues with the syntax analyzer, only statically declared arrays are supported, hence no dynamic arrays with malloc or any other type of pointer arithmetics are supported.

(iii) Input to the application is problematic due to some external reasons. There is no direct way to get input save for batches of data saved in a file in FIBRE format.

(iv) There is no way to "mark" which loops are going to be parallelized and which should be left alone. C2μTC/SL blindly analyses and transforms all of them. The only way to make a loop run sequentially is to utilize a variable which increases by one. Such an act forces the compiler to sequentialize the loop with that variable marked as shared.

For those reasons, it is not possible for any real life application to be compiled as-is. It will have to be re-written in order to comply with the above restrictions: Any function calls that perform actual computation which needs to be parallelized should be inlined in the main function. The rest of the functions need to be declared as external in that source file and implemented in a different file. They can be linked against the transformed file once C2μTC/SL is done with it. Since there is no way to mark loops for parallelization, loops that only perform printouts (for example loops that print the contents of a matrix) should not exist within the same main file or at the very least they should be forced to be transformed in a sequential form (through the use of shared variables). It is best for that kind of code to be factorized into an external function call. Finally, input data should be declared statically inside the code itself for any example or be batch-loaded through some helper FIBRE functions.

Despite of that though, most of the research goals that were set were achieved: Single dimensional loops, carrying dependences or not, can be transformed in a manner which improves their efficiency by a large degree even if there is no parallelism to exploit (thanks to the synchronising channels). Simple multi-dimensional loops (without dependences) can also be transformed into fully parallel SVP constructs (families) that produce the same result while providing great speedups (46 in the case of 64 cores for example). Finally, perfectly nested loops with a static dependence vector can be parallelized in a wavefront-like style. Instead of focusing on compile-time methods which try to calculate the perfect hyperplane to utilize and then only estimate or even guess at the tile size to use (since it has been proven that a fully fine grain method will oversaturate at large problem sizes so tiling is a necessity), a different approach was chosen: Utilising the information of the run-time environment to the benefit of the compiler.

This novel solution was met with many difficulties, mostly because of the rather small bibliography on parallelizing in run-time, but in the end the Fixed-Size Algorithm was born which eventually evolved to the Self-Adaptive system: An algorithm that follows the dependence vector in order to choose which indices will execute at a given cycle (similar to the hyperplane, only instead of a hyperplane there are execution cycles). Not only does that algorithm intuitively find the best

hyperplane to use, it also finds the best tile size to utilise in order to achieve performance as close to maximum as possible.

There are several aspects though that find C2μTC/SL lacking and will need addressing in some future work. Those aspects can be categorized into two sets, software engineering and research.

When it comes to software engineering aspects, all of the limitations that were listed in the first paragraph must be fixed. There is nothing inherently difficult however a rather large timespan and a great deal of work must be invested in that aspect. Other areas also need improvement. The symbolic interpreter, albeit having served its purpose perfectly, is at an infant stage and the process of dependence detection relies on very simple expression identification (only the form of "A[i] = A[i – constant]" is understood). A proper symbolic analyzer needs to be implemented which will be able to comprehend complex expressions as well as dependences that extend into multiple statements. Moreover, the compiler itself is entwined with SL and the SVP in general. However, certain ideas and transformations it employs can be applied in more general systems. A different branch of its development should focus on producing output for libraries and systems widely in use: pthreads in a lower level or OpenMP for a higher level of abstraction. This would allow not only C2μTC/SL's usage to become more widespread (which can lead to more people picking it up and upgrading it) but also for some comparison with other commodity compilers in existence today.

Research-wise, even though the Self Adaptive Algorithm has proven to work for the typical nested loop with a static dependence vector, there is still plenty of room for improvement. Firstly, it is rather slow in its convergence rate. Since at each cycle the tile size in use is altered by the value of one, it takes several cycles for it to reach an optimal state. A smarter system needs to be implemented that will be increasing or decreasing the tile size based on its distance from the target or at the very least in a faster way than the current system. Secondly, the scheduler thread can in theory be improved. The way it traverses the coordinates with the dependence vector, can be executed in parallel and hence help the scheduler end faster. This will result in higher amounts of parallelism in general and hence greater speedups. Lastly, some way to deal with non-static dependences should be researched. If a dependence can be

described as an affine combination of a series of known variables (i.e. the indices) then, in theory, it should be possible to extend the algorithm to transform these kinds of cases as well.

In conclusion, C2µTC/SL is an automatic parallelizing compiler which is capable of transforming C code into parallel SL code that can be executed by an SVP system. It provides a combination of compile time techniques (in cases of no dependence existing or in single-dimensional loops) with a run-time technique that was researched and developed especially for this compiler. Experimental results indicate that the run-time method can offer significant improvements in execution times and is definitely on the right path. Even though it can not compete with traditional compile-time methods in pure speedup gain, its versatility (i.e. handling irregular index spaces, calculating the optimal tile size, etc.) more than makes up for that. More work is needed in various areas: The compiler should be able to deal with more than one functions in a program, pointer arithmetics should be implemented in order to deal with dynamically allocated arrays, I/O needs to be improved and the symbolic analyser should also be expanded with the ability to "understand" more diverse types of expressions inside array subscripts. Finally, the main Self-Adaptive algorithm itself can also benefit from a few improvements. Convergence rate needs to be improved, the scheduler thread can benefit from some inherent parallelism and finally non static dependences need to be researched.

REFERENCES

[1] R. Allen and K. Kennedy. "Automatic Translation of FORTRAN programs to Vector Form", ACM Transactions on Programming Languages and Systems, pp:491-542, 1987.

[2] R. Allen and K. Kennedy. "Optimizing Compilers for Modern Architectures", Morgan Kaufmann Publishers, 2001.

[3] G. Almasi and A. Gottlieb. "Highly Parallel Computing", The Benjamin / Cummings Publishing Company, Inc., 1994.

[4] J. M. Anderson, S. P. Amarasinghe and M. S. Lam. "Data and Computation Transformations for Multiprocessors", Proc. of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Processing, Jul. 1995.

[5] J. M. Anderson and M.S. Lam. "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines", In Proc. Of SIGPlan '93 Conf. Programming Language Design and Implementation, ACM Press, New York, pp. 112-125, 1993.

[6] R. Asenjo, R. Castillo, F. Corbera, A. Navarro, A. Tineo and E. L. Zapata. "Parallelizing Irregular C Codes Assisted by Interprocedural Shape Analysis", in 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS'08), 2008.

[7] D. Bacon, S. Graham, O. Sharp. "Compiler Transformations for High-Performance Computing", Computing Surveys, v:26, pp:345 - 420, 1994.

[8] H. Bae, L. Bachega, C. Dave, S-I. Lee, S. Lee, S-J. Min, R. Eigenmann and S. Midkiff. "Cetus: A Source-to-Source Compiler Infrastructure for Multicores", In Proc. Of the 14th Intl. Workshop on Compilers for Parallel Computing, 2009.

[9] U. Banerjee. "Dependence Analysis for Supercomputing", Kluwer. Boston, MA, 1988.

[10] U. Banerjee. "Loop Transformations for Restructuring Compilers", Kluwer Academic, 1993.

[11] P. Banerjee, J. A. Chandy, M. Gupta, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy and E. Su. "The Paradigm Compiler for Distributed Memory Multicomputers", IEEE Computer, Oct. 1995, v. 28, pp. 37-47, 1994.

[12] U. Banerjee, R. Eigenmann, A. Nicolau and D. Padua. "Automatic Program Parallelization", Proceedings of the IEEE, 81(2)pages 211-243, February 1993.

[13] M. Baskaran, N. Vydyanathan, U. Bondhugula, J. Ramanujam, A. Rountev and P. Sadayappan. "Compiler-Assisted Dynamic Scheduling for Effective Parallelization of Loop Nests on Multicore Processors", in Proc. Of PoPP, pp:219-228, 2009.

[14] C. Bastoul. "Code Generation in the Polyhedral Model is Easier than You Think", In Proc. Of the 13th International Conference on Parallel Architectures and Compilation Techniques, IEEE Computer society, Washington DC, USA, pp: 7 - 16, 2004.

[15] V. Beletskyy and M. Poliwoda. "Parallelizing Perfectly Nested Loops with Non-Uniform Dependencies", In Proc. Of the Advanced Computer Systems, pp:83-98, 2002.

[16] T. Bernard, K. Bousias, L. Guang, C. R. Jesshope, M. Lankamp, M. W. van Tol and L. Zhang. "A General Model of Concurrency and its Implementation as Many-core Dynamic RISC Processors", In Proc. of Intl.Conf. on Embedded Computer Systems: Architecture, Modeling and Simulation, SAMOS-2008, ISBN: 978-1-4244-1985-2, pp. 1-9, Samos, Greece, 2008.

[17] T. A. M. Bernard, C. Jesshope, and M. Lankamp. "Evaluation of a Hardware Implementation of the SVP Concurrency model". ISCA 2010.

[18] W. Blume and R. Eigenmann. "The Range Test: A Dependence Test for Symbolic, Non-linear Expressions", Technical Report 1345, Univ. of Illinois at Urbana-Champaign, Centr. for Supercomputing Res & Dev., April 1994.

[19] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu and S. Weatherford. "Effective Automatic Parallelization with Polaris", International Journal of Parallel Programming, May 1995.

[20] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev and P. Sadayappan. "Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model", In Proc. Of the the International Conference on Compiler Construction, 2008.

[21] U. Bondhugula, A. Hartono, J. Ramanujan and P. Sadayappan. "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer", ACM SIGPLAN Programming Languages Design and Implementation (PLDI), Tucson, Arizona, June 2008.

[22] K. Bousias, L. Guang, C.R. Jesshope and M. Lankamp. "Implementation and Evaluation of a Microthread Architecture", Journal of Systems Architecture, Volume 55, Issue 3, pp 149-161, March 2009.

[23] C. Brownhill, A. Nicolau, S. Novack and C. Polychronopoulos. "Achieving Multi-Level Parallelization", High Performance Computing, Lecture Notes in Computer Science Volume 1336, pp 183-194, 1997.

[24] K. D. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon and S. K. Warren. "The ParaScope Parallel Programming Environment", In Proc. of IEEE, pp. 244-263, Feb. 1993.

[25] A. Darte and F. Vivien. "Optimal Fine and Medium Grain Parallelism Detection in Polyhedral Reduced Dependence Graphs", International Journal of Parallel Programming, v:25, pp:447-496, 1997.

[26] A. Darte, Y. Robert and F. Vivien. "Scheduling and Automatic Parallelization", Birkhäuser Boston, 2000.

[27] V.V. Dimakopoulos, E. Leontiadis and G. Tzoumas. "A portable C compiler for OpenMP V.2.0", in Proc. EWOMP 2003, 5th European Workshop on OpenMP, Aachen, Germany, Sept. 2003, pp. 5--11.

[28] K. A. Faigin, J. P. Hoeflinger, D. A. Padua, P. M. Petersen and S. A. Weatherford. "The Polaris Internal Representation", Technical report, Univ. of Illionois at Urbana-Champaign, Cntr. for Supercomputing Res. and Dev. CSRD Report No. 1317, UILU-ENG-93-8038, October 1993.

[29] M. Flynn. "Some Computer Organizations and Their Effectiveness". IEEE Trans. Comput. C–21: 948. 1972.

[30] D. C. Grunwald. "Data Dependence Analysis for Supercompilers: the lambda Test Revisited", Technical report, Boulder University of Colorado Dept. of Computer Science.

[31] M. R. Haghighat and C. D. Polychronopoulos. "Dependence Analysis", Kluwer Academic Publishers, 1995.

[32] M. W. Hall, J-A. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S-W. Liao, E. Bugnion and M. S. Lam. "Maximizing Multiprocessor Performance with the SUIF Compiler", Digital Technical Journal, (10)1:71-80, 1998.

[33] M. W. Hall, K. Kennedy and K. S. McKinley. "Interprocedural Transformations for Parallel Code Generation", Supercomputing '91, pages 423-434, 1991.

[34] A. Hayashi, Y. Wada, H. Shikano, T. Kamiayama, T. Watanabe, T. Sekiguchi and M. Mase. "OSCAR Parallelizing Compiler Cooperative Heterogeneous Multi-

Core Architecture", The Eighteenth International Conference on Parallel Architectures and Compilation Techniques (PACT2009), 2009.

[35] J. Hoeflinger and Y. Paek. "Unified Interprocedural Parallelism Detection", International Journal of Parallel Processing, 2000.

[36] K. Ishizaka, T. Miyamoto, J. Shirako, M. Obata, K. Kimura and H. Kasahara. "Performance of OSCAR Multigrain Parallelizing Compiler on SMP Servers", In Proc. of 17th International Workshop on Languages and Compilers for Parallel Computing, 2004.

[37] C. R. Jesshope. "µTC – An Intermediate Language for Programming Chip Multiprocessors", In Proc. of Advances in Computer Systems Architecture, 11th Asia-Pacific Conference, ACSAC 2006, Shanghai, China, September 6-8, pp. 147 – 160, 2006.

[38] C. R. Jesshope. "SVP and µTC - A Dynamic Model of Concurrency and its Implementation as a Compiler Target", Technical Teport, University of Amsterdam, 2007.

[39] C. Jesshope, M. Hicks, M. Lankamp, R. Poss and L. Zhang. "Making Multi-cores Mainstream – From Security to Scalability", In Parallel Computing: From Multicores and GPU's to Petascale, Vol. 19, pp. 16-31, 2010.

[40] C. R. Jesshope, J-M Philippe and M. van Tol. "An Architecture and Protocol for the Management of Resources in Ubiquitous and Heterogeneous Systems Based on the SVP Model of Concurrency", In Proc. of Intl. Workshops on Embedded Computer Systems: Architecture, Modeling and Simulation, SAMOS-2008, LNCS 5114, pp. 218-228, Samos, Greece, 2008.

[41] T. A. Johnson, S-I. Lee, L. Fei, A. Basumallik, G. Upadhyaya, R. Eignmann and S. P. Midkiff. "Experiences in Using Cetus for Source-to-Source Transformations", In Proc. Of the 17th Intl. Workshop on Languages and Compilers for Parallel Computing, 2004.

[42] B. W. Kernighan and D. M. Ritchie. "The C programming language, 2nd edition, Section A13", Prentice Hall, 1988.

[43] X. Kong, D. Klappholz and K. Psarris. "The I Test: An Improved Dependence Test for Automated Parallelization and Vectorization". IEEE Transactions on Parallel and Distributed Systems 2 (1991), 342-349.

[44] D. J. Kuck. "High Performance Computing, Challenges for Future Systems", Oxford University Press, New York, 1996.

[45] D. Kulkarni and M. Stumm. "Loop and Data Transformations: A Tutorial", University of Toronto, 1993.

[46] L. Lamport. "The Parallel Execution of DO loops", Commun. ACM, v:17, pp:83-93, 1974.

[47] S-I. Lee, T. A. Johnson and R. Eigenmann. "Cetus - An Extensible Compiler Infrastructure for Source-to-Source Transformation", In Proc. Of the 16th Intl. Workshop on Languages and Compilers for Parallel Computing, v:2958, pp:539-553, 2003.

[48] A. W. Lim, G. I. Cheong and M. S. Lam. "An Affine Partitioning Algorithm to Maximize Parallelism and Minimize Communication", Proc.of the 13th ACM SIGARCH International Conference on Supercomputing, Jun.1999.

[49] A. G. Navarro, F. Corbera, A. Tineo, R. Asenjo and E. L. Zapata. "Detecting Loop-Carried Dependences in Programs with Dynamic Data Structures", Parallel Distrib. Comput. v:67, pp: 47-62, 2007.

[50] D. A. Padua and M. J. Wolfe. "Advanced Compiler Optimizations for Supercomputers", Commun. ACM, v:29, pp:1184 – 1201.

[51] C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. Leung and D. Schouten. "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling programs on Multi-processors", International Journal of High Speed Computing, 1(1): 45-72, 1989.

[52] C. D. Polychronopoulos, M. B. Gikar, M. R. Haghighat, C. L. Lee, B. P. Leung and D. A. Schouten. "The Structure of Parafrase-2: An Advanced Parallelizing Compiler for C and Fortran", In Languages and Compilers for Parallel Computing. MIT Press, 1990.

[53] R. Poss. "On the realizability of hardware microthreading. Revisiting the general-purpose processor interface: consequences and challenges", Technical Report, University of Amsterdam, 2012. ISBN 978-94-6108-320-3.

[54] B. Pottenger. "Parallelism in Loops Containing Recurrences", Technical report, Univ. of Illinois at Urbana-Champaign, June 1996.

[55] B. Pottenger and R. Eigenmann. "Idiom Recognition in the Polaris Parallelizing Compiler", International Conference on Supercomputing, 1995.

[56] W. Pugh. "The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis", In Proc. of Super Computing '91, 1991.

[57] L. Rauchwerger, N. M. Amato and D. A. Padua. "Run-Time Methods for Parallelizing Partially Parallel Loops", Proceedings of the 9th ACM International Conference on Super computing, Barcelona, Spain, pages 137–146, Jul.1995.

[58] H. Saito, N. Stavrakos, S. Carroll, C. Polychronopoulos and A. Nicolau. "The Design of the PROMIS Compiler", Compiler Construction, Lecture Notes in Computer Science Volume 1575, pp 214-228, 1999.

[59] J. Saltz, R. Mirxhandaney and K. Crowley. "Run-time Parallelization and Scheduling of Loops", IEEE Trans. Comput., 40(5), May 1991.

[60] P. Tu and D. Padua. "Automatic Array Privatization", Proc. 6th Annual Workshop on Languages and Compilers for Parallel Computing, 1993.

[61] D. W. Wall. "Limits of Instruction-level Parallelism", In Proc. Of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV), Apr. 1991.

[62] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S-W. Liao, C-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers", ACM SIGPLAN Notices, v: 29, pp: 31-37, 1994.

[63] M. Wolfe. "High Performance Compilers for Parallel Computing", Addison-Wesley Publishing Company, 1995.

[64] M. Wolfe and C.-W. Tseng. "The Power Test for Data Dependence", IEEE Transactions on Parallel and Distributed Systems, v:3, issue 5.

[65] M. E. Wolf and M.S. Lam. "A Loop Transformation Theory and an Algorithm to Maximize Parallelism", IEEE Transactions on Parallel Distributed Systems v:2 pp:452 - 471, 1991.

[66] H. Zima. "Supercomputers for Parallel and Vector Computers", Addison-Wesley, 1991.

[67] [Online] http://openmp.org/wp/

[68] [Online] http://openmp.org/wp/openmp-specifications/

[69] [Online], http://www.mpi-forum.org/

[70] [Online], https://developer.nvidia.com/category/zone/cuda-zone

[71] [Online], https://www.cilkplus.org/

[72] [Online], http://www.paraphrase-ict.eu/

[73] [Online], http://www.sac-home.org/

[74] [Online], http://www.apple-core.info

[75] [Online], http://svp-home.org

[76] [Online], http://www.ace.nl/compiler/cosy.html

[77] [Online], http://en.wikipedia.org/wiki/Conway's_Game_of_Life

[78] [Online], http://en.wikipedia.org/wiki/Mandelbrot_set

[79] [95] [Online], http://en.wikipedia.org/wiki/Livermore_loops

APPENDIX A. THE SL LANGUAGE

The information contained in this Appendix comes mostly from [53]. As has been mentioned already, the SL language is essentially the C language expanded with a series of macro definitions that help encapsulate all the parallel constructs functionality. Due to this property, the grammar utilized by C2μTC/SL is the one listed below (The original C language specification is listed in [42]) :

```
<translation-unit> ::= <external-declaration>*

<external-declaration> ::= <function-definition>
                         | <thread-function-declaration>
                         | <thread-function-definition>
                         | <declaration>

<thread-function-definition> ::= sl_def ( <identifier> {, <attributes>? {, <thread-parametre-
list>}?}? ) <compound-statement> <sl-enddef>

<thread-function-declaration> ::= sl_decl ( <identifier> , <thread-specifiers>? {, <thread-
parametre-list>}? ) ;

<thread-parametre-list> ::= <thread-parametre-declaration>
                          | <thread-parametre-declaration> , <thread-parametre-list>

<function-definition> ::= {<declaration-specifier>}* <declarator> {<declaration>}* <compound-
statement>

<declaration-specifier> ::= <storage-class-specifier>
                          | <type-specifier>
                          | <type-qualifier>

<storage-class-specifier> ::= auto
                            | register
                            | static
                            | typedef
```

```
<type-specifier> ::= void
                   | char
                   | short
                   | int
                   | long
                   | float
                   | double
                   | signed
                   | unsigned
                   | <struct-or-union-specifier>
                   | <enum-specifier>
                   | <typedef-name>


<struct-or-union-specifier> ::= <struct-or-union> <identifier> { {<struct-declaration>}+ }
                              | <struct-or-union> { {<struct-declaration>}+ }
                              | <struct-or-union> <identifier>


<struct-or-union> ::= struct
                    | union


<struct-declaration> ::= {<specifier-qualifier>}* <struct-declarator-list>


<specifier-qualifier> ::= <type-specifier>
                        | <type-qualifier>


<struct-declarator-list> ::= <struct-declarator>
                           | <struct-declarator-list> , <struct-declarator>


<struct-declarator> ::= <declarator>
                      | <declarator> : <constant-expression>
                      | : <constant-expression>


<declarator> ::= {<pointer>}? <direct-declarator>


<pointer> ::= * {<type-qualifier>}* {<pointer>}?


<type-qualifier> ::= const
                   | volatile


<direct-declarator> ::= <identifier>
                      | ( <declarator> )
                      | <direct-declarator> [ {<constant-expression>}? ]
                      | <direct-declarator> ( <parameter-type-list> )
                      | <direct-declarator> ( {<identifier>}* )


<constant-expression> ::= <conditional-expression>


<conditional-expression> ::= <logical-or-expression>
```

```
                              | <logical-or-expression> ? <expression> : <conditional-expression>


<logical-or-expression> ::= <logical-and-expression>
                          | <logical-or-expression || <logical-and-expression>


<logical-and-expression> ::= <inclusive-or-expression>
                           | <logical-and-expression && <inclusive-or-expression>


<inclusive-or-expression> ::= <exclusive-or-expression>
                            | <inclusive-or-expression> | <exclusive-or-expression>


<exclusive-or-expression> ::= <and-expression>
                            | <exclusive-or-expression> ^ <and-expression>


<and-expression> ::= <equality-expression>
                   | <and-expression> & <equality-expression>


<equality-expression> ::= <relational-expression>
                        | <equality-expression> == <relational-expression>
                        | <equality-expression> != <relational-expression>


<relational-expression> ::= <shift-expression>
                          | <relational-expression> < <shift-expression>
                          | <relational-expression> > <shift-expression>
                          | <relational-expression> <= <shift-expression>
                          | <relational-expression> >= <shift-expression>


<shift-expression> ::= <additive-expression>
                     | <shift-expression> << <additive-expression>
                     | <shift-expression> >> <additive-expression>


<additive-expression> ::= <multiplicative-expression>
                        | <additive-expression> + <multiplicative-expression>
                        | <additive-expression> - <multiplicative-expression>


<multiplicative-expression> ::= <cast-expression>
                              | <multiplicative-expression> * <cast-expression>
                              | <multiplicative-expression> / <cast-expression>
                              | <multiplicative-expression> % <cast-expression>


<cast-expression> ::= <unary-expression>
                    | ( <type-name> ) <cast-expression>


<unary-expression> ::= <postfix-expression>
                     | ++ <unary-expression>
                     | -- <unary-expression>
                     | <unary-operator> <cast-expression>
                     | sizeof <unary-expression>
```

```
                            | sizeof <type-name>


<postfix-expression> ::= <primary-expression>
                       | <postfix-expression> [ <expression> ]
                       | <postfix-expression> ( {<assignment-expression>}* )
                       | <postfix-expression> . <identifier>
                       | <postfix-expression> -> <identifier>
                       | <postfix-expression> ++
                       | <postfix-expression> --


<primary-expression> ::= <identifier>
                       | <constant>
                       | <string>
                       | ( <expression> )
                       | sl_geta ( identifier )
                       | sl_getp ( identifier )


<constant> ::= <integer-constant>
             | <character-constant>
             | <floating-constant>
             | <enumeration-constant>


<expression> ::= <assignment-expression>
               | <expression> , <assignment-expression>


<assignment-expression> ::= <conditional-expression>
                          | <unary-expression> <assignment-operator> <assignment-expression>


<assignment-operator> ::= =
                        | *=
                        | /=
                        | %=
                        | +=
                        | -=
                        | <<=
                        | >>=
                        | &=
                        | ^=
                        | |=


<unary-operator> ::= &
                   | *
                   | +
                   | -
                   | ~
                   | !


<type-name> ::= {<specifier-qualifier>}+ {<abstract-declarator>}?
```

```
<parameter-type-list> ::= <parameter-list>
                        | <parameter-list> , ...


<parameter-list> ::= <parameter-declaration>
                   | <parameter-list> , <parameter-declaration>


<parameter-declaration> ::= {<declaration-specifier>}+ <declarator>
                          | {<declaration-specifier>}+ <abstract-declarator>
                          | {<declaration-specifier>}+


<abstract-declarator> ::= <pointer>
                        | <pointer> <direct-abstract-declarator>
                        | <direct-abstract-declarator>


<direct-abstract-declarator> ::=  ( <abstract-declarator> )
                                | {<direct-abstract-declarator>}? [ {<constant-expression>}? ]
                                | {<direct-abstract-declarator>}? ( {<parameter-type-list>|? )


<enum-specifier> ::= enum <identifier> { <enumerator-list> }
                   | enum { <enumerator-list> }
                   | enum <identifier>


<enumerator-list> ::= <enumerator>
                    | <enumerator-list> , <enumerator>


<enumerator> ::= <identifier>
               | <identifier> = <constant-expression>


<typedef-name> ::= <identifier>


<declaration> ::=  {<declaration-specifier>}+ {<init-declarator>}*
                | <thread-index-declaration>
                | <thread-function-pointer-declaration>
                | <thread-function-pointer-typedef>


<thread-function-pointer-typedef> ::= sl_typedef_fptr ( <identifier> { , <thread-specifiers>?
{ , <thread-parametre-list>}?}? ) ;


<thread-function-pointer-declaration> ::= sl_decl_fptr ( <identifier> , <thread-specifiers>? {
, thread-parametre-list}? ) ;


<thread-index-declaration> ::= sl_index ( identifier ) ;


<thread-specifiers> ::= <thread-specifier-item>
                      | ( thread-specifier-list )


<thread-specifier-list> ::= <thread-specifier-item>
```

```
                              | <thread-specifier-list> , <thread-specifier-item>


<thread-specifier-item> ::= <thread-specifier>
                         | <thread-attribute>


<thread-specifier> ::= sl__static


<thread-atribute> ::= undefined


<init-declarator> ::= <declarator>
                   | <declarator> = <initializer>


<thread-parametre-list> ::= <thread-parametre-declaration>
                         | <thread-parametre-list> , <thread-parametre-declaration>


<thread-parametre-declaration> ::= sl_glparm ( <declaration-specifiers> , <identifier> )
                                | sl_glfparm ( <declaration-specifiers> , <identifier> )
                                | sl_shparm ( <declaration-specifiers> , <identifier> )
                                | sl_shfparm ( <declaration-specifiers> , <identifier> )




<initializer> ::= <assignment-expression>
               | { <initializer-list> }
               | { <initializer-list> , }


<initializer-list> ::= <initializer>
                    | <initializer-list> , <initializer>


<compound-statement> ::= { {<declaration>}* {<statement>}* }
                      | { <create-construct> }


<statement> ::= <labeled-statement>
             | <expression-statement>
             | <compound-statement>
             | <selection-statement>
             | <iteration-statement>
             | <thread-argument-assignment>
             | <thread-parametre-assignment>


<thread-argument-assignment> ::= sl_seta ( <identifier> , <assignment-expression> ) ;


<thread-parametre-assignment> ::= sl_setp ( <identifier> , (assignment-expression> ) ;


<create-construct> ::= sl_create ( , <create-parametres> , <create-specifiers>? , <assignment-
expression> {, <thread-argument-list>}? ) ; <create-block-item-list>? sl_sync ( ) ;


<create-parametres> ::= <assignment-expression>? , <range-parametres>
```

```
<range-parametres>  ::=  <assignment-expression>?  ,  <assignment-expression>?  ,  <assignment-
expression>?  ,  <assignment-expression>?


<create-specifiers> ::= <create-specifier>
                     | ( <create-specifier-list> )


<create-specifier-list> ::= <create-specifier>
                          | <create-specifier-list> , <create-specifier>


<create-specifier> ::= <thread-attribute>


<thread-argument-list> ::= <thread-argument-definition>
                         | <thread-argument-list> , <thread-argument-definition>


<thread-argument-definition>  ::=  sl_glarg  (  <declaration-specifiers>  ,  <identifier>?  {,
<assignment-expression>}? )
                                | sl_glfarg  (  <declaration-specifiers>  ,  <identifier>?  {,
<assignment-expression>}? )
                                | sl_sharg  (  <declaration-specifiers>  ,  <identifier>?  {,
<assignment-expression>}? )
                                | sl_shfarg  (  <declaration-specifiers>  ,  <identifier>?  {,
<assignment-expression>}? )


<create-block-item-list> ::= <create-block-item>
                           | <create-block-item-list> , <create-block-item>


<create-block-item> ::= statement
                     | <create-construct>


<labeled-statement> ::= <identifier> : <statement>
                     | case <constant-expression> : <statement>
                     | default : <statement>


<expression-statement> ::= {<expression>}? ;


<selection-statement> ::= if ( <expression> ) <statement>
                        | if ( <expression> ) <statement> else <statement>
                        | switch ( <expression> ) <statement>


<iteration-statement> ::= while ( <expression> ) <statement>
                        | do <statement> while ( <expression> ) ;
                        | for ( {<expression>}? ; {<expression>}? ; {<expression>}? )
<statement>


<identifier> ::= <letter>* { <letter> | <digit> }*


<letter> ::= a | b | ... | z | A | B | ... | Z | _
```

```
<digit> ::= 0 | 1 | ... | 9
```

It is clear that this grammar is a superset of the C language, so any C legacy program can be compiled and executed under SVP with no change. However, taking advantage of the parallelism offered by the hardware requires the code to declare threads and invoke them from some other thread (main can also be considered a thread). Alongside the syntax of SL, a list of constraints and semantics follows:

Constraints:

- The identifier used in sl_geta() must be a visible thread argument name.
- The identifier used in sl_getp() must be a thread parameter name in the enclosing thread.
- The sl_geta function cannot be used in any thread function body.
- The sl_geta function can only appear inside its corresponding create context.
- The sl_setp function cannot appear outside of a thread function body.
- A thread index declaration can only appear in a thread function body.
- Argument names cannot be used in any other create construct in the same scope.
- A goto from outside a create construct cannot jump inside one and vice versa.
- Thread functions cannot have a return statement.
- The identifier inside a thread function definition must be in the same name space as C names.

Semantics:

- Each use of sl_getp generates a side effect.
- If execution reaches an expression using sl_getp after it has passed a sl_setp statement using the same thread parameter identifier, the behavior of the program becomes undefined.
- A thread function declaration declares a thread function with the specified name and prototype, with external linkage unless the attribute "sl__static" is specified.

- The thread specifier sl__static plays the same role as C's storage qualifier *static* on external declarations.
- A thread parameter definition specifies channel endpoints for the thread program. The directives sl_glparm and sl_glfparm specify global channel endpoints while the directives sl_shparm / sl_shfparm denote a shared channel.
- sl_shparm / sl_glparm denote (directly or indirectly vie typedefs) integers. sl_shfparm / sl_glfparm denote in the same way floats / doubles.
- Each execution is associated with a unique logical thread index, which can be observed via a sl_index declaration in the designated thread program.
- If execution reaches a thread argument or parameter assignment statement after it has passed another such statement designating the same channel endpoint, the behavior of the program becomes undefined.

A list of the most important directives of SL alongside a description for each follows:

sl_def(thread_name, return_type, …) {code} **sl_enddef**. sl_def defines a thread named thread_name and a return type of return_type (usually void). In the (…) part a series of arguments is listed. Arguments are passed by value exactly like the C language. Once the thread body's functionality is defined (i.e. the instruction sequence is complete) between the brackets { }, sl_enddef designates to the compiler the end of a thread definition.

sl_shparm / sl_shfparm (parameter_type, parameter_name). Inside sl_def()'s parameter list, each shared channel parameter is formally defined with this directive. Parameter_type indicates the type of the data (int, char *, etc) while parameter_name indicates the name of the particular shared channel. In the case of a floating point value, the directive sl_shfparm needs to be used instead.

sl_glparm / sl_glfparm (parameter_type, parameter, parameter_name). Similar to the previous directive, this one defines a global channel parameter inside the sl_def's parameter list. Again, in the case of a floating point type of variable, the sl_glfparm directive needs to be used in place.

sl_index(variable_name). Stores the index of the current thread to the variable designated by variable_name.

sl_getp(channel_name). Decouples the value from a channel named channel_name and returns it for use or storage inside a thread local variable. The channel can be either a global or a shared one and in the case of a shared channel, if the channel is empty, sl_getp will block the execution of the entire thread. It shouldn't be called more than once per channel so it is wise to store all such decouplings into local variables.

sl_setp(shared_channel_name, shared_value). Writes the value of shared_value back into a shared channel named shared_channel_name. It is meaningful only for shared channels and thusly it should be used only then and only once. If a thread does not write back to the shared channel a deadlock might occur.

sl_break(). Similar to C's break, which breaks execution of a loop and continues the execution past the point of the loop's end, sl_break() terminates the execution of the entire family of threads. Control of the program moves past the family's synchronization point.

sl_create. Perhaps the most important directive of SL. Its usual invocation is sl_create(,,from,to,step,,,thread_body,…). It creates a family of threads whose index will have a starting value of "from", will go up to the value of "to" and have a step of "step". This means that (to-from)/step threads will exist inside this family. The … is the argument list that assigns values to the global and the shared channels.

sl_sharg / sl_shfarg(value_type, shared_channel_name, initial_value). Part of the formal parameter list of sl_create, it creates a shared channel named shared_channel_name which carries a value of type value_type. Additionally, it can be initialized with the value of initial_value. In the case of a floating type value, sl_shfarg should be used instead. The sharg / shfarg directives set the two endpoints of the shared channel that will be applied to all threads in the family. The initial value is automatically set and the final value can be read after the synchronization point.

sl_glarg / sl_glfarg(value_type, global_channel_name, initial_value). Another part of the formal parameter list of sl_create, this set of directives creates and initializes a global channel that permeates all threads in the family. The name of the channel will be global_channel_name, its type will be of value_type and it will be initialized with the value of initial_value. Again if the value is of floating point type then the counterpart sl_glfparm needs to be used.

sl_sync(). Similar to the classic join for threads, sl_sync() will halt execution of the parent thread that created a family and wait till that family terminates to continue execution.

sl_geta(shared_channel_name). Once a family has terminated, the parent thread can read the final value of a shared channel via sl_geta. It takes as argument the name of the shared channel, decouples and returns its value for storage in a variable or direct use.

It should be stated here that all parameter types passed between threads are basic types or pointers to / arrays of them (type-defined). Any other user defined type (like compound types (i.e. structs / unions)) is not currently supported by SL. A simple example code similar to the classic "Hello world" program is depicted below:

```
sl_def (void, print)
{
        sl_index(i);
        printf("Hello from thread %d\n",i);
}
sl_enddef
```

This thread declaration defines a thread that prints "Hello from thread " and its accompanying index (its position inside the family chain). Creating a family of those threads is also straightforward:

```
sl_create(,,0,N,1,,,print);
sl_sync();
```

This code creates a family of N threads that will all execute in parallel. The indices inside the family will range from 0 to N-1 and increment by 1. A full list of constraints and semantics of SL can be found in Appendix I of [53].

APPENDIX B. SUPPORTED C SUBSET

In a similar manner to Appendix A the supported subset of the C grammar (in BNF form) is listed below:

```
<translation-unit> ::= <external-declaration>


<external-declaration> ::= <function-definition>


<function-definition> ::=  <main-type> main {<declaration>}* <compound-statement>


<main-type> ::= int
              | void


<declaration-specifier> ::= <storage-class-specifier>
                          | <type-specifier>
                          | <type-qualifier>


<storage-class-specifier> ::= auto
                            | register
                            | static
                            | typedef


<type-specifier> ::= void
                   | char
                   | short
                   | int
                   | long
                   | float
                   | double
                   | signed
                   | unsigned
                   | <struct-or-union-specifier>
                   | <enum-specifier>
                   | <typedef-name>


<struct-or-union-specifier> ::= <struct-or-union> <identifier> { {<struct-declaration>}+ }
                              | <struct-or-union> { {<struct-declaration>}+ }
                              | <struct-or-union> <identifier>
```

```
<struct-or-union> ::= struct
                    | union


<struct-declaration> ::= {<specifier-qualifier>}* <struct-declarator-list>


<specifier-qualifier> ::= <type-specifier>
                        | <type-qualifier>


<struct-declarator-list> ::= <struct-declarator>
                           | <struct-declarator-list> , <struct-declarator>


<struct-declarator> ::= <declarator>
                      | <declarator> : <constant-expression>
                      | : <constant-expression>


<declarator> ::= {<pointer>}? <direct-declarator>


<pointer> ::= * {<type-qualifier>}* {<pointer>}?


<type-qualifier> ::= const
                   | volatile


<direct-declarator> ::= <identifier>
                      | ( <declarator> )
                      | <direct-declarator> [ {<constant-expression>}? ]
                      | <direct-declarator> ( <parameter-type-list> )
                      | <direct-declarator> ( {<identifier>}* )


<constant-expression> ::= <conditional-expression>


<conditional-expression> ::= <logical-or-expression>
                           | <logical-or-expression> ? <expression> : <conditional-expression>


<logical-or-expression> ::= <logical-and-expression>
                          | <logical-or-expression || <logical-and-expression>


<logical-and-expression> ::= <inclusive-or-expression>
                           | <logical-and-expression && <inclusive-or-expression>


<inclusive-or-expression> ::= <exclusive-or-expression>
                            | <inclusive-or-expression> | <exclusive-or-expression>


<exclusive-or-expression> ::= <and-expression>
                            | <exclusive-or-expression> ^ <and-expression>


<and-expression> ::= <equality-expression>
                   | <and-expression> & <equality-expression>
```

```
<equality-expression> ::= <relational-expression>
                        | <equality-expression> == <relational-expression>
                        | <equality-expression> != <relational-expression>


<relational-expression> ::= <shift-expression>
                          | <relational-expression> < <shift-expression>
                          | <relational-expression> > <shift-expression>
                          | <relational-expression> <= <shift-expression>
                          | <relational-expression> >= <shift-expression>


<shift-expression> ::= <additive-expression>
                     | <shift-expression> << <additive-expression>
                     | <shift-expression> >> <additive-expression>


<additive-expression> ::= <multiplicative-expression>
                        | <additive-expression> + <multiplicative-expression>
                        | <additive-expression> - <multiplicative-expression>


<multiplicative-expression> ::= <cast-expression>
                              | <multiplicative-expression> * <cast-expression>
                              | <multiplicative-expression> / <cast-expression>
                              | <multiplicative-expression> % <cast-expression>


<cast-expression> ::= <unary-expression>
                    | ( <type-name> ) <cast-expression>


<unary-expression> ::= <postfix-expression>
                     | ++ <unary-expression>
                     | -- <unary-expression>
                     | <unary-operator> <cast-expression>
                     | sizeof <unary-expression>
                     | sizeof <type-name>


<postfix-expression> ::= <primary-expression>
                       | <postfix-expression> [ <expression> ]
                       | <postfix-expression> ( {<assignment-expression>}* )
                       | <postfix-expression> . <identifier>
                       | <postfix-expression> -> <identifier>
                       | <postfix-expression> ++
                       | <postfix-expression> --


<primary-expression> ::= <identifier>
                       | <constant>
                       | <string>
                       | ( <expression> )


<constant> ::= <integer-constant>
             | <character-constant>
```

```
                    | <floating-constant>
                    | <enumeration-constant>


<expression> ::= <assignment-expression>
                  | <expression> , <assignment-expression>


<assignment-expression> ::= <conditional-expression>
                            | <unary-expression> <assignment-operator> <assignment-expression>


<assignment-operator> ::= =
                          | *=
                          | /=
                          | %=
                          | +=
                          | -=
                          | <<=
                          | >>=
                          | &=
                          | ^=
                          | |=


<unary-operator> ::= &
                    | *
                    | +
                    | -
                    | ~
                    | !


<type-name> ::= {<specifier-qualifier>}+ {<abstract-declarator>}?


<parameter-type-list> ::= <parameter-list>
                          | <parameter-list> , ...


<parameter-list> ::= <parameter-declaration>
                    | <parameter-list> , <parameter-declaration>


<parameter-declaration> ::= {<declaration-specifier>}+ <declarator>
                            | {<declaration-specifier>}+ <abstract-declarator>
                            | {<declaration-specifier>}+


<abstract-declarator> ::= <pointer>
                          | <pointer> <direct-abstract-declarator>
                          | <direct-abstract-declarator>


<direct-abstract-declarator> ::=  ( <abstract-declarator> )
                                  | {<direct-abstract-declarator>}? [ {<constant-expression>}? ]
                                  | {<direct-abstract-declarator>}? ( {<parameter-type-list>|? )
```

```
<enum-specifier> ::= enum <identifier> { <enumerator-list> }
                   | enum { <enumerator-list> }
                   | enum <identifier>


<enumerator-list> ::= <enumerator>
                    | <enumerator-list> , <enumerator>


<enumerator> ::= <identifier>
               | <identifier> = <constant-expression>


<typedef-name> ::= <identifier>


<declaration> ::=  {<declaration-specifier>}+ {<init-declarator>}*


<init-declarator> ::= <declarator>
                    | <declarator> = <initializer>


<initializer> ::= <assignment-expression>
                | { <initializer-list> }
                | { <initializer-list> , }


<initializer-list> ::= <initializer>
                     | <initializer-list> , <initializer>


<compound-statement> ::= { {<declaration>}* {<statement>}* }


<statement> ::= <labeled-statement>
              | <expression-statement>
              | <compound-statement>
              | <selection-statement>
              | <iteration-statement>


<labeled-statement> ::= <identifier> : <statement>
                      | case <constant-expression> : <statement>
                      | default : <statement>


<expression-statement> ::= {<expression>}? ;


<selection-statement> ::= if ( <expression> ) <statement>
                        | if ( <expression> ) <statement> else <statement>
                        | switch ( <expression> ) <statement>


<iteration-statement> ::= while ( <expression> ) <statement>
                        | do <statement> while ( <expression> ) ;
                        |  for  (  {<expression>}?  ;  {<expression>}?  ;  {<expression>}?  )
<statement>


<identifier> ::= <letter>* { <letter> | <digit> }*
```

```
<letter> ::= a | b | ... | z | A | B | ... | Z | _

<digit> ::= 0 | 1 | ... | 9
```

Essentially, the whole translation module becomes only a single function (called main) while there are no declarations outside that main function. In addition, jump statements of any form are not supported since they break normal code flow. Everything else retains exactly the same grammar and semantics of the original language.

AUTHOR'S PUBLICATIONS

[1] D. Saougkos, G. Manis, "Self Adaptive Run Time Scheduling for the Automatic Parallelization of Loops with the C2µTC/SL Compiler", Parallel Computing 39 (2013), pp. 603-614.

[2] D. Saougkos, G. Manis, "A Parallelizing Compiler for the Microgrid: Exploiting Concurrency from Software Continuity", In: The AppleCore Project Workshop organized during High Performance and Embedded Architecture and Compilation (HiPEAC) 2012, Paris.

[3] D. Saougkos, G. Manis, "Run Time Scheduling with the C2uTC Parallelizing Compiler", In: 2nd Workshop on Parallel Programming and Run - Time Management Techniques for Many − Core Architectures, organized during 24th Conference on Computing Systems (ARCS 2011), 2011, pp. 151-157.

[4] D.Saougkos, A. Mastoras, G. Manis, "Fine Grained Parallelism in Recursive Function Calls", In: Workshop on Languaged-Based Parallel Programming Models organized during PPAM (Parallel Processing and Applied Mathematics) Conference, Torun, Poland, September 2011.

[5] D. Saougkos, D.Evgenidou, and G.Manis. "Specifying Loop Transformations for C2µTC source − to −source compiler", in 14th Workshop on Compilers for Parallel Computing (CPC '09), 2009.

[6] D. Saougkos, G. Manis, K. Blekas, A. V. Zarras, "Revisiting Java Bytecode Compression for Embedded and Mobile Computing Environments", IEEE Transactions on Software Engineering, vol. 33, no. 7, pp. 478-495, Jul., 2007.

SHORT CURRICULUM VITAE

Dimitris Saougkos was born, raised and spent his formative years in Ioannina. At the age of 17 he was accepted into the Computer Science Department of the University of Ioannina with honors (Highest entry grade). Programming was always an interest for him and, as such, the choice of major was easy. During his studies he also received awards for being the first student in the first, second and third year of studies. After graduation he was accepted with honors (highest entry grade) in the Post Graduate department of the same Computer Science Department which he finished after two years specialized in Software. Once he completed his mandatory military duty, and after having discussed about the possibility of a PhD with assistant professor George Manis, he was accepted as a PhD candidate in 2008 and also worked as a researcher for the E.U. – funded project APPLECORE where he developed a source-to-source compiler. He has also worked as IT support for the DASTA office of the University of Ioannina. Currently (2014 - 2015) he is working as a Software Design Engineer at the UK-based company Imagination Technologies. His research interests include automatic (and general) parallelization, compilers and system programming.