Automatic Parallelization

Jakob Østergaard
c958612

IMM — DTU
Lyngby

9th of August 1999

Abstract

With the availability of cheap networking hardware and fast commodity computer systems, *clusters* have become an extremely cost-effective way of gaining massive processing power. The alternative, to buy single specialized high-performance computers is becoming less interesting all the time, because of their much higher prices for equivalent performance.
However, there are still reasons for not using clusters. The good reason is, that some problems cannot be parallelized well. The other reason is, that it takes time and effort to successfully parallelize old sequential programs into parallel programs that will run efficiently on the cluster.
This work addresses the latter reason. This is a description of an attempt made, to fully automate the process of not only *parallelizing*, but also distributing or *scheduling* an otherwise serial program.

The work described herein was conducted for the Department of Mathematical Modelling, at the Technical University of Denmark. Supervisors on the project were *Per Christian Hansen* and *Jens Clausen*.
The original idea for the TONS computing environment that some of the ideas in this work builds upon, came from *Kenneth Geisshirt*.

Contents

Chapter 1

Introduction

If you have the right attitude,
interesting problems will find you
— Eric S. Raymond

This chapter seeks to give a short introduction to the ideas that lie behind this project. Since the approach taken in the system described here is (I think) radically different from the majority of development happening in scientific computing systems, this chapter will also serve as an attempt to justify these ideas.

1.1 The need for parallelization

As CPU speeds increase, the size of the problems we wish to solve using those CPUs increase likewise. Therefore, there will never come a time where we do not need more computing power than we have.

There are two ways of getting more power: Buy *faster* CPUs, or buy *more* CPUs. From a programming point of view, buying a faster CPU is by far the most feasible choice, because it does not involve re-writing code to take advantage of parallelism in the problem being solved. But from an economical point of view, buying more CPUs is a significantly cheaper way of achieving more raw power, as CPU prices do not correspond linearly with their speed.

I think it is pretty clear, that there will always be a need for exploiting parallelism in computing problems, so that those problems can be executed on parallel architectures. However, there is also a need to free the programmers from thinking about this parallelism.

There are compilers out there, that try to free the programmer from thinking about parallelism. High Performance Fortran is a language that, among other things, tries to make some of this abstraction from parallel programming a reality.

But even if these compilers were perfect (which of course they aren't), they still would not free the user from being bothered with parallel issues. Which CPUs are available for the program to use ? What resources are available ? What combination of CPUs yields the best resource usage ? The list of questions goes on...

What we need, is a computing *platform*, not just a new language or another compiler. There are too many of those already.

1.2 A computing platform

A friend of mine came up with the idea to build a new computing environment, one that would do the things right, we felt other environments did wrong. We have been discussing this openly with other people as well, and therefore the number of things we find is wrong in existing tools is large, and I am just going to mention a few key issues here:

- We need a good front-end language (the language the end-user uses), with no unnecessary historical luggage (odd or ugly syntax due to back-compatibility concerns). Personally I find Fortran to be an excellent example of truly ugly syntax (consider the indentation requirements for example – why would anyone want that in a language ?).
- The computing "engine", the *backend*, must be efficient enough, so that real production code can be run on this system with acceptable performance (this is what MatLab does wrong).
- Because we think cluster computing is going to be widely used, and because we hate to parallelize code manually, the computing environment should also be able to automatically parallelize and distribute computations on a cluster of machines. The C programming language is especially hard to parallelize automatically, due to the freedom it gives the programmer to reference memory as she pleases. This is an unnecessary freedom in numerical computing (it's great if you write operating system kernels, but we don't care about that in a numerical computing language).
- The software must be open source[1]. This is necessary if we ever want to build anything that can compete with existing tools. We cannot build a replacement for MatLab, RLab, and the others, if we keep the development process closed and the code a secret.

After having decided upon these goals, we found that the project and the various sub-projects needed a name.

The name of the computing environment as a whole, became **TONS**. In the beginning it was short for "**T**oolkit for **O**pen **N**umerics". Later we changed it to be "**T**ons **o**f **N**umerics". Note that the latter is a recursive acronym, I think that was the main reason for the change. However that story is not all that important for the project.

The front-end language has been dubbed **TAL**, for **T**ONS **A**lgorithmic **L**anguage. But since this language still only exists in our minds and not in the form of real code, I will not discuss this language any further here.

This project touches aspects of several of the goals mentioned above. An efficient virtual machine was developed, and it currently has the capability to automatically parallelize and distribute code to a cluster of machines, with some restrictions and limitations of course.

The TONS system today, consists only of the code that I have written in order to examine the possibilities to perform automatic parallelization and distribution of calculations. Therefore, when I refer to the TONS project in any of the following, I refer to the code that I wrote for this project and this report.

[1]See `http://www.opensource.org/` and `http://www.fsf.org/` for the philosophy behind Free Software and Open Source

1.3 The TONS idea

The TONS system is an attempt to create for scientific computing, an abstraction that frees the user from being bothered by parallelization issues. The two main tasks the system will take care of, that are interesting to us in this report, are:

- Automatic parallelization of code
- Scheduling, distribution and execution of that code

One could argue, that this is no different from using a parallelizing compiler and a batch queuing system. I disagree, of course. A parallelizing compiler does not parallelize the code with regard to the current state of the parallel machine. TONS does that to some extent already[2]. A queuing system won't start a parallel program requiring N CPUs, before N CPUs are available, which in turn means that $N-1$ CPUs have been idle for some time. A system like TONS has the potential to perform much smarter scheduling, although this is not currently implemented.

I cannot see how such functionality could reasonably be implemented using compilers and queuing systems as we know them today. Since the parallelization involves code changes, depending on the current circumstances, traditional tools would probably require a timely compile of the user code, every time it was to run. TONS does not build upon compilers and queuing systems, rather it bends the rules by *not* compiling the code, which in turn makes smarter scheduling possible.

TONS is a system that executes *interpreted* code. It is a distributed virtual machine, that will execute code on a cluster of machines.

1.4 "Interpreted" code

A language interpreter can be a great many number of things, and in order not to confuse things too much I here defined what *I* mean when I write *interpreted* in this report.

A language interpreter is a piece of software that allows the end-user to write a program in some human-readable language, and have this program executed *directly* by the interpreter.

This is in contrast to language *compilers*, that translate the human-readable code into machine-readable code, so that the end-user can execute the machine-readable code at a later time.

Early interpreters *parsed* the human-readable code line by line, executing each line as it had been parsed. Loops, for example, would require the interpreter to parse the code lines within the loop in *every* iteration of the loop. Needless to say, this is both extremely slow (because it involves scanning and parsing of text all the time), and inconvenient to the programmer (because syntax errors are not discovered until the erroneous line is reached in the program).

Modern interpreters contain a built-in "translator", that initially parses the entire user program, translating the user-written code into some internal representation which is more easily used by the interpreter. This has the benefits of finding all syntax errors before execution begins, and it is usually orders of magnitude faster than the older approach.

[2]Simple limiting in the parallelizer is implemented as of this writing. With N available nodes, a maximum of N parallel tasks will be generated

When I refer to "interpreters" in this report, I do mean a "modern" interpreter. I know that the Perl language interpreter performs this translation before execution, I also implemented the TONS virtual machine that way, and from discussions with my supervisor Per Christian Hansen, we reasoned that MatLab does the same thing.

1.5 Interpreted code and performance

High performance computing and interpreted languages are usually phrases that tend not to show up in the same sentences. That is a pity I think. The functionality I hope to build into the TONS system would not be possible, or at least close to impossible, if we had to compile the user code (in the normal meaning of the word "compile").

There is only one problem with interpreted languages, that keep us from using them in scientific computing; *performance*. That, however, should be a non-issue in most cases, if the language we interpret is designed carefully enough.

The absolute majority of scientific computing code deals with repetitive tasks. It can be anything from traversing a set of molecules in a molecular dynamics (MD) program, to approximating the solution to a system of non-linear equations. We usually work with sets of numbers, not single numbers. And we usually work with sets that are either large (matrix multiplication or LU factorization comes to mind), or has a simple relationship (eg. adjacent molecules in an MD simulation).

If the interpreted language provides ways to deal with these set operations effectively, the number of interpreted instructions will be very small, compared to the number of compiled and highly optimized instructions the TONS system will be executing to actually solve the problem.

For example, if the interpreted language provides a `mult` instruction, that performs a matrix multiplication, the difference between the performance of the interpreted language, and any compiled language (C, Fortran, etc.) will *only* be the overhead in the virtual machine for recognizing the `mult` instruction. The real code that executes the $\mathcal{O}(n^3)$ operations is highly optimized machine code in either case.

Thus, a carefully designed interpreted computing language should be able to execute at a speed close to that of a compiled language, for a very large class of problems.

Chapter 2

Designing the system

Never eat more than you can lift
— Miss Piggy

In this chapter, the design of the TONS distributed computing system is outlined, and some issues regarding the design decisions that where made are discussed.

2.1 System overview

Three major components are supposed to make up the system. These are: node-server software, the central scheduler, and some client software for the users. This is sketched in Figure 2.1. For now, only the node-server and the scheduler have been implemented. Those are the back-end components, and those are the really interesting ones. There is no need to build a front-end before we have the back-end working.

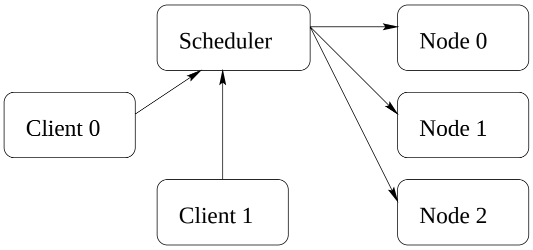

Figure 2.1: Overview of the TONS distributed computing system.

2.2 The user's view

The end-user of the system is of course not supposed to think in terms of byte-code, assemblers or node-servers. We want to hide these uninteresting (to the end-user at least) technicalities. The way I would like to see this system evolve, we will be able to present users with a friendly front-end and a well designed language for expressing numerical problems. How the parallelization and scheduling is actually happening, should not be a concern of the user. The system as I envision it, will in time evolve into something that – to the end user – will look much like Figure 2.2.

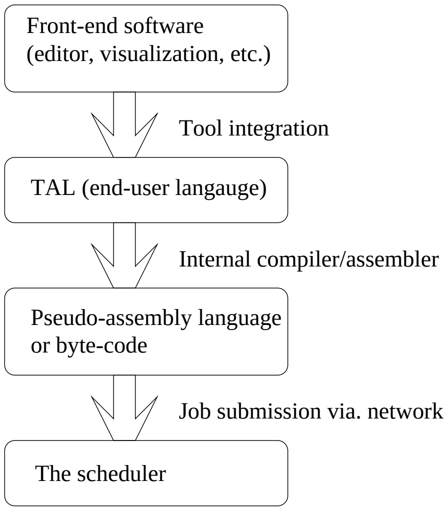

Figure 2.2: The user's view of the system.

Actually most of what the user sees, will be integrated into one client tool, that will provide source-code editing, visualization and job submission capabilities. The user will not need to bother whether the code is being run on 1 or 100 nodes.

A well functioning back-end is – as I see it – the most vital part of this system, and it is what this report is all about. The second most important thing will likely be a good language for the end-user. The current VM-oriented assembly language is a reflection of the inner workings of the virtual machine, *not* a reflection of the way users will want to easily express computing problems.

The development of the TAL (TONS Algorithmic Language) language, the compiler, and the front-end application, is a huge task in itself, and way out of the scope of this report. There is some work in progress with defining the language, but it will take years before we see a really polished well integrated end-user interface to the TONS system. The current TAL development is headed in the direction of a language that combines ideas from Pascal, MatLab and Cobol. I am not taking part in this development currently, it is done by people I know, who share the interest in developing a better computing environment. So far, I have been the sole developer of the code discussed in this report.

2.3 The scheduler

"Scheduler" is not an entirely covering name for this central part of the system. In fact, the scheduler is supposed to take care of several tasks, that make this system look as one large machine:

- Receive jobs (pieces of serial code) from clients
- Parallelize code wrt. the current state of the cluster
- Schedule and distribute code and data to selected nodes

Right now, the scheduler does not take requests from clients (since there are no clients). The scheduler is run with a program name as a command-line argument, and will then load that program into the TONS system, and perform parallelization and scheduling on that program. When the nodes return the result, the scheduler terminates. This is something that will change, but the current behavior is sufficient for development.

2.3.1 Sub-clustering

In the design chosen, there will only be one scheduler, independent of the size of the cluster. For this design to be viable, we must make sure that the scheduler does not become a bottleneck. Therefore we can not allow the scheduler to do any heavyweight computation, nor can we allow it to do a lot of communication.

Parallelization and scheduling are both light tasks. Both topics will be stressed in sections following.

In order to make sure the scheduler does not get involved in too much communication, it will distribute an entire parallel job to one single node, and pass control (or "responsibility") of that job to that node, along with information about which other nodes are available for that job. This "master node" will then distribute tasks to the other nodes, the sub-cluster, and take care of the communication needed, without involving the scheduler. This is sketched in Figure 2.3.

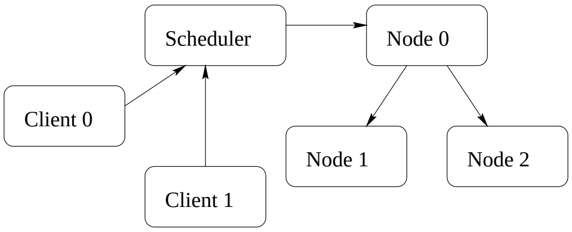

Figure 2.3: Subclustering delegates most of the communication work to a master node.

2.3.2 The parallelization engine

Since the parallelization is currently taking place on one central machine, it is absolutely vital that the complexity of the parallelization algorithms is good. Preferably the complexity should be better than $\mathcal{O}(n^2)$, for n instructions. It can, of course, never be faster than $\mathcal{O}(n)$.

The algorithms that lie behind the parallelization are described thoroughly in Chapter 4.

Since this system is intended to be used for numerical computation, it is vital that we do not compromise the numerical properties the algorithms we execute rely on. That is, we cannot allow the parallelizer to change the *algorithm* or *method* itself in a way that would significantly change the outcome of the computation.

This is something that I have not treated thoroughly in this report, since it is still not completely clear to me what we can do and what we cannot.

For now, the parallelizer plays safe. It will never change the order in which the calculations that make up a result are executed. It therefore will never change the properties of the numerical algorithm it parallelizes.

There are other technicalities that come into play, if the system is run on a heterogenous cluster. The nodes that participate in the cluster may have different machine precision (!) However, since the user is still guaranteed that the precision with which some algorithm is computed is never less than the lowest machine precision present in the cluster, this does not seem to be much of a problem.

If code is run on such a heterogenous cluster, the apparent machine precision can actually change between different runs of the program. I don't think this is a problem, as long as the user's are aware of this. It is definitely not worth seeking to guarantee a uniform machine precision throughout the cluster. This would impose a computing overhead on some of the nodes, and since numerical computations per definition are inaccurate anyway, this seems like a completely bad idea.

2.3.3 The scheduler logic

Once we have a set of parallel tasks, and a set of nodes that are available to us, we should distribute those tasks to the nodes in a way that minimizes the execution time, since in the end all that matters is to get the job done as fast as possible.

This is something that is not well implemented yet. We do almost no real scheduling, as the system is today. This was a lower priority, and didn't get the attention it needs.

2.3.4 Keeping everyone busy

This is a loose discussion of some performance issues with regards to scheduling of jobs. None of this is implemented into the system, but since a system such as the TONS system actually allows these considerations to be made, I feel that I should at least present some of my thoughts on these issues.

We ought to distribute information among the nodes and the scheduler on a regular basis, that would allow the scheduler to make smarter decisions about which groups of nodes should be assigned which jobs.

This information would be a set of data that describes which nodes are working, for how long they may be working, which ones are idle, and so on. This information is inaccurate (it is out-dated the second it is generated, because of the ever changing state of the cluster), so there will probably be a need to look into heuristics and the theory of planning under uncertainty. A node will not know what other nodes are idle or busy, but it may have a rough idea of the probabilities of the different nodes being idle or busy.

It is important that we do not *ask* the nodes about their status before submitting jobs from the scheduler. If this was done, we would take a performance hit from the network latency, especially this would be bad for large clusters if we asked 100 or more nodes whether they were idle or busy, before we decided what to do with a job.

A possible solution would be to submit the job to the nodes that have the highest probability of being idle. Somehow the number of jobs each node holds must be propagated, over time, to the entire cluster. If a node becomes idle, it could request the job from a node that accidentally got two jobs (because of the uncertainty when the scheduler distributed the jobs).

Another approach could be that idle nodes register with the scheduler *immediately* when they become idle. Then the scheduler would know exactly what nodes where idle, and it could make up a sub-cluster with the currently idle nodes, and delegate the new job to this sub-cluster.

If there are tasks enough, we could also just make sure that all nodes at all times have at least one, but preferably two runnable tasks. This would ensure, that while a node is waiting for input, or returning it's reply, it still has computation work to do. This is a simple way of making the node idle-time approach 0, and to make the network latency and bandwidth much less important. It requires that the data sets in the tasks held by the nodes fit in the memory of the nodes of course.

It will require a lot of testing and benchmarking, to come up with a scheme that will fit our needs both with respect to speed, and to information detail and accuracy. Maybe even different approaches will be needed for smaller and larger clusters.

Currently the scheduler knows what nodes exist in the cluster. It will delegate a job to an (arbitrarily selected) master node, and tell the master node to utilize all remaining nodes in the cluster, if possible, for the job. This is not the way things should be done, but it's a reasonable way to do things for now, since we can't schedule more than one job at a time anyway (see Section 2.3).

2.4 The node server

The node server does a fairly simple job: It receives a chunk of code, and the data this code needs for its computations, executes the code, and returns the result.

In the node server, the focus is on the virtual machine. The virtual machine must be efficient. Even though the language provides instructions that can do fairly complex (i.e. computationally expensive) operations, like matrix multiplication, there will be cases where we need to execute a larger number of instructions to complete some simple task. There is a constant overhead in the virtual machine for executing an instruction, no matter which instruction. This overhead is mainly

- Calling the correct instruction executing code (all instructions have a corresponding C++ implementation in the virtual machine)
- Setting up pointers to register values (needed every time we change stack frames)
- Moving the instruction pointer, and various other virtual machine state maintenance

The Fibonacci number calculation program which will be introduced in Section 5.2 is an extreme example that almost does no real work but has it's execution speed limited only by the number of instructions that are executed (not the work they do).

Chapter 3

The language

Language designers are arrogant
It comes with the territory
— Larry Wall

When the user submits jobs to the scheduler and virtual machine, this happens by means of *byte code*, a machine representation of the user's job. The byte code is built with an assembler that translates human-readable simple instructions into this machine representation. This chapter will describe the ideas behind the language that gives us the byte code. The language is a 1:1 mapping to the actual byte-code, so by dissecting the language, we will be exploring the inner workings of the virtual machine itself. First, however, I will present some of my personal views on two existing languages currently in widespread use.

3.1 Existing languages

A lot of time and effort has been spent by compiler vendors and others, to produce automatically parallelizing compilers. Still, the compilers available today suffer from several shortcomings:

- The user must supply "hints" in the source-code, to tell the compiler when and where to parallelize.
- *Many* compilers only work on shared memory architectures, not on distributed memory.
- The compilers do not understand, or work with, the job queuing system, and vice versa.

The third observation is a fundamental design-oriented flaw in today's systems. It has little to do with the virtual machine language, but a lot to do with the design of this system as a whole.

From the first two observations, it could seem as an impossible task to create a fully automatically parallelizing computing system, that even runs on distributed memory systems.

I believe that it is not so. The compilers that seek to automatically parallelize code, are working with languages that by nature are not parallel. These languages are usually Fortran or C.

Since the TONS system is designed from scratch, it is not at all necessary to base the system on an existing language. A new language was designed, one that lends itself well to parallelism, yet has no way for the user to represent parallelism. This may sound corny, but keep in mind that we want the machines to do the work. If a feature was put in the language, that would allow the user to "hint" the system, we would soon end up with something as messy as the existing parallelizing compilers.

Note that this new language is the language which translates directly into the code understood by the virtual machine. It is *not* the language we intend our end-users to see.

The end-user language, the language referred to as "TAL" in Section 1.2, will undoubtedly borrow a lot of ideas from existing languages. In order to make the new computing environment useful to anyone, we must of course have a language that is both intuitive and looks somewhat familiar to our prospected users (the people using C, Fortran and MatLab now). For now we will look at the kind of language that is nice to parallelize, not necessarily nice to look at.

3.1.1 Some problems with C and Fortran

Both C and Fortran are *imperative* languages. A typical program will consist of

- A main routine
- A set of sub-routines
- A set of global variables

Furthermore, variables can be passed between routines, as they call each other.

In order to automatically parallelize a program, it is necessary for us to know exactly:

- Which instructions depend on which other instructions ?
- Who modifies what, and when ?
- When does it pay off to parallelize ?

If we know the dependencies, we know what can run in parallel, and what cannot. It becomes necessary to know who modifies what, when we parallelize. The order in which the parallel tasks are evaluated, must not influence the result of the computation.

The fact that both C and Fortran can have global variables, makes it messy (but not impossible) to find out exactly who modifies what variables. When large programs are split into separate compilation units, it will require the compiler to look thru all compilation units (source files), to see what routines touch the global variables.

In C, aliasing is possible. That is, two seemingly distinct variables actually share some of the same memory. So, code that changes one variable, may actually also change another, although it is not clear from the code. It is not necessarily possible to predict aliasing at compile time. This makes it effectively impossible to automatically parallelize C code, without the user supplying hints for the compiler.

However, the main problems with existing parallelizing systems (as I see it) is, that they are parallelizing at *compile-time*. Dimensions of variables are most usually unknown at compile time, which means that parallelization must rely on either:

The user telling the compiler and/or libraries about the likely dimension of the variables, or the code and libraries performing a number of dimension checks at run-time.

If one chooses to hand-parallelize the code, using MPI, PVM, or similar libraries, the problem of changes becomes even worse. The criteria for when parallelization pays off and when it doesn't, is often reflected in the design of the program, not just in a set of easily configured parameters. (I know, because I have been working with a lot of code suffering from this).

3.2 The new language

Very unlike most existing languages in use for scientific and high-performance computing, our virtual machine (VM) language is a *functional* language.

To be more specific, this language does lack a lot of features found in "real" functional languages, such as ML[1]. Our language is extremely primitive, actually it looks not entirely unlike traditional machine code assembly language.

In the following, we refer to the basic building blocks of the language, the "instructions", as *opcodes*. This is mainly for convenience (the opcode notion is used a lot in the implementation). The instructions do for the virtual machine, what real opcodes do for a real processor.

3.2.1 Data types

Clearly, in order to do any useful work, we must define some set of data types we wish to work with. Again, the virtual machine does not yet implement everything one could wish for, but it does implement the most basic types. The types are outlined in Table 3.2.1.

Type	Description
integer	A 32 bit signed integer, used especially as loop variable and for indexing in other types
floating	A 64 bit IEEE floating point type
tuple	A tuple can hold any number of elements. The elements can be of any type. This is mostly used for argument passing, where it is only possible to pass one argument to a function. One can pass a tuple holding any number of elements (arguments), since the tuple is just one argument.
vector	This is a vector that holds elements of type floating.
matrix	This is a matrix holding elements of type floating.

Table 3.1: List of data types currently implemented in the virtual machine.

I felt these were the most basic and important types. The tuple type is important because one can only pass *one* argument to a sub-routine call, and sub-routines

[1]The Meta Language. See `http://cm.bell-labs.com/cm/cs/what/smlnj/sml.html` for details

can only return *one* argument. The tuple type can hold any number of elements of any type (even tuples of tuples of ...), thereby allowing us to pass several variables back and forth between sub-routine calls.

Some languages have multi-dimensional types. I have not yet implemented n-dimensional arrays, but in a sense the tuple type is already n-dimensional, so the basic functionality for handling n-dimensional variables is already present in the system. In the meantime one can use a tuple of vectors or matrices, to simulate n-dimensional arrays.

3.2.2 Side effects

The main difference between this language and the more conventional imperative languages is, that *side effects* do not exist.

A piece of C code that calculates an element-wise vector product could look like:

```
double A[1000];
double B[1000];
double C[1000];

void vmult(double* a, double* b, double* c, int length) {
{
  c += length;
  while(length--)
    *a++ = *b++ * *--c;
}

int main(int argv, char** argv)
{
  /* some code here... */

  /* A(1:1000) = B(1:1000) .* C(1000:-1:1) */
  vmult(A, B, C, 1000);

  /* some more code here... */

  /* C(1:500) = B(1:500) .* C(1000:-1:501)    */
  /* and                                      */
  /* C(501:1000) = B(501:1000) .* C(500:-1:1) */
  vmult(C, B, C, 1000);

}
```

The above code calculates the element-wise product of two vectors, and stores the result in a third vector. For two vectors (a_1, a_2, a_3) and (b_1, b_2, b_3) the routine will calculate the result (a_1b_3, a_2b_2, a_3b_1). Note that the ordering of the second vector is reversed.

I put the equivalent MatLab code in the comments above the two calls to the `vmult()` routine.

The first call `vmult(A, B, C, 1000)` does exactly what we would expect it to. It stores the special element-wise vector product result in the `A` array. This could fairly easily be parallelized. We could execute the 100 multiplications on each node in a 10 node cluster for example.

The second call however, has a two-line MatLab equivalent. It produces a radically different result. The first 500 multiplication results are stored in the `C` vector, which then is used as *input* to the last 500 multiplications. This does not parallelize as well as the previous example, since we require knowledge of the first results in order to compute the last results.

Because we do not pass the argument values (eg. the elements of `A`, `B` and `C`) to the function, but rather point at the memory location to where the first elements reside, we are completely unable to decide whether or not we can parallelize the `vmult()` routine just by looking at the routine. We will need complete knowledge about the location of the addresses passed to the routine. This is bad, since this information often can not be predicted at all.

One could then argue, that the C example should have been written differently. Well, if one needs an element-wise vector multiplication routine that reverses the order of the right-hand-side vector, then the above C code is a compact (and simple to the trained C programmer – however obscure to the untrained eye) and very efficient implementation.

Besides from being difficult to parallelize without more knowledge about the code than a compiler usually has, the C example is also a pristine example of code that works as expected, until someone uses it in a way that was not expected by the original implementer (for example, someone wants to store the result in one of the argument vectors, as in the second call to the routine in the example above, but expects the result to be the product of the two arguments).

In the language designed here, similar code would look like:

```
entry "example"
  decl  vector,   ; first vm register r0 is a vector
        vector,   ; second vm register r1 is a vector
        vector    ; third vm register r2 is too

  ; some code here...

  ; multiply
  move r0, r1             ; r0 <- r1
  mult r0, r2[999:-1:0]   ; r0 <- r0 * r2[999:-1:0]

  ; some more code here...

  ; multiply again

  ; first we compute  r2[0:499] <- r1[0:499] * r2[999:-1:500]
  move r2[0:499], r1[0:499]
  mult r2[0:499], r2[999:-1:500]
  ; then we compute  r2[500:999] <- r2[0:499] * r2[499:-1:0]
  move r2[500:999], r2[0:499]
  mult r2[500:999], r2[499:-1:0]

end
```

There is a perfectly clear relationship between the instruction, and the data it depends upon or *uses*, and the data it modifies or *touches*.

Also it is clear from the code what actually happens. It is easy to see (at least for the virtual machine), that the results from the second `mult` is re-used in the last `mult`.

In the virtual machine we do not have the equivalent of C pointers. The user doesn't know where the data are located in memory, and shouldn't need to know either. The fact that data may be located on different machines also make the whole idea of pointers somewhat ambiguous or inadequate. As a user, one only cares about the data, not where the data physically resides.

3.2.3 Introduction to the VM langauge

Since this is the first example code of the language, I think it deserves an explanation.

First of all, all routines are contained within an `entry` and an `end` opcode. The `entry` opcode tells the virtual machine about the name of the routine, and the `end` opcode quite intuitively ends the current routine.

The first opcode within a routine must be a `decl` opcode. It *declares* the types of the variables we use in this routine. The variables are referred to as *registers*. The virtual machine can operate with any number of registers, but for efficiency reasons the variables (registers) are numbered $r_{0...n}$ instead of being described with more intuitive names.

Of course, the TAL language should use named variables. It will then be up to the TAL translator to store these variables into numbered registers.

The `decl` opcode takes one or more arguments. The arguments are the type names from Table 3.2.1. First argument describes the type of register 0 (`r0`), second argument describes register 1, and so forth.

Note that the dimensions are not specified in the `decl` opcode. The dimension of a register can change dynamically at run-time.

Most opcodes take two arguments. The `move` opcode will copy the contents of the second argument into the first argument. So, the code for the first multiplication:

```
move r0, r1
mult r0, r2[999:-1:0]
```

actually copies `r1` into `r0`, then multiplies the contents of `r0` and the reversed `r2`, and stores this result in `r0`. It is equivalent to $\mathtt{r0} \leftarrow \mathtt{r1}_{0...999} \cdot \mathtt{r2}_{999...0}$

A `mult` opcode that took three arguments, a destination register and two source registers, would have spared us for the `move` operation. This would probably be a good idea to implement in the system.

3.2.4 High-level opcodes

In order for an interpreted language to become almost as efficient as a language that is natively compiled for the target platform processor, we can not rely on implementing all our basic functionality (such as matrix multiplication) by means of simple scalar operations provided by the language.

If we look a bit at the extremes, we can take a look at how the `vmult()` function from the previous example look when it is translated for the CPU to execute. I passed the C code thru an optimizing C compiler, and fetched the assembly output from the compiler.

```
vmult:
        pushl %ebp
        movl %esp,%ebp
        pushl %ebx
        movl 20(%ebp),%eax
        leal 0(,%eax,8),%edx
        movl 8(%ebp),%ebx
        movl 12(%ebp),%ecx
        addl 16(%ebp),%edx
        subl $1,%eax
        jc .L4
        .p2align 4,,7
.L5:
        fldl (%ecx)
        addl $-8,%edx
        fmull (%edx)
        addl $8,%ecx
        fstpl (%ebx)
        addl $8,%ebx
        subl $1,%eax
        jnc .L5
.L4:
        popl %ebx
        movl %ebp,%esp
        popl %ebp
        ret
```

Table 3.2: The `vmult()` routine translated into Intel i686 machine code.

Translated into real CPU instructions for my PC at home, the code becomes what is seen in Table 3.2.

Note that the tight loop that performs the multiplication takes up eight instructions (it's the code between the two labels `.L5` and `.L4` above). These eight instructions are executed 1000 times each, when the `vmult(A,B,C,1000)` call is made.

This is efficient because it runs directly on the CPU. But if we used a language such as Intel 80x86 assembly (like the code above) as the representation language in our virtual machine, we would end up with an *extremely* slow system. This is because we have an overhead for executing an instruction, on top of what the instruction itself requires (also see section 2.4).

Remember, the virtual machine must first recognize the opcode, then move to the relevant routine in the virtual machine that handles that particular recognized opcode, and *then* the CPU gets to actually execute the machine-code instructions that performs the action dictated by the recognized virtual machine opcode.

Not only for the performance reasons would it be infeasible to feed the virtual machine code on the form of the assembly language in the previous example. Think about parallelizing such code. The C example given in the previous section may look bad, but the lower the level we see the code at, the worse it gets.

The solution is to define a fairly large set of instructions, that can take care of these trivial tasks, efficiently. For example, the `mult` instruction used for multiplying vectors, will also perform matrix multiplication. The actual code that does the real work is written in C++ and doesn't look all that much different from the C example presented earlier.

Again the virtual machine is different from a "real" (hardware) machine here. In hardware, it is extremely costly to implement a large set of instructions. That is why we see the high performance hardware of today moving away from CISC (complex/complete instruction set computing) towards RISC (reduced instruction set computing) architectures instead. In the virtual machine however, a new instruction has *no* impact on the efficiency of the virtual machine (with exception from possible cache pollution due to the increased code size). Instructions are decoded using a technique similar to hash-table lookups which are $\mathcal{O}(1)$ independently of the number of instructions. Another megabyte of code in the virtual machine doesn't really cost us anything with regards to efficiency, but another million of transistors is a whole other story for the hardware producers.

The current set of opcodes implemented in the virtual machine are listed in table 3.2.4.

Opcode	Description
`move dst, src`	Move data from a source to a destination
`decr dst`	Decreases the value of the argument by one
`cmpgt dst,` $\texttt{src}_1$`,` $\texttt{src}_2$	Compares the two source values and stores a “1” in the destination if $\texttt{src}_1 > \texttt{src}_2$. Otherwise it stores a “0”.
`mult dst, src`	Stores the product $\texttt{dst}\times\texttt{src}$ in `dst`.
`add dst, src`	Stores the sum $\texttt{dst}+\texttt{src}$ into `dst`.
`zero dst, src`	Stores zeros into `dst`. If `src` is a scalar, it will make sure `dst` can hold either `src` elements in the case where `dst` is a vector, or $\texttt{src}^2$ elements in the case where `dst` is a matrix. This can conveniently be used to allocate vectors or matrices.
`div dst, src`	Divides `dst` with `src` and stores the result in `dst`.
`call dst,` $\texttt{src}_1$`,` $\texttt{src}_2$	Calls the function named $\texttt{src}_1$ with the argument $\texttt{src}_2$, and stores the returned value in `dst`.

Table 3.3: List of opcodes in the virtual machine

The arguments passed to the instructions are named `dst` (short for “destination”) when the argument is something the instruction will *touch* (modify). Arguments are named `src` (short for “source”) when they are only *use*d.

We will need a much larger set of opcodes, but the current set has been sufficient to implement a number of small test programs that were used for developing the automatic parallelization.

Depending on the opcodes, the arguments can be scalars, vectors, matrices or tuples. For example, the `mult` opcode implements both scalar integer and floating-point multiplication, element-wise vector multiplication, element-wise tuple multiplication, and matrix multiplication.

3.2.5 Function calling

Perhaps the functional-language heritage shows itself the most in the way we do function (or “subroutine”) calling and argument / return-value passing.

A `call` opcode is implemented for subroutine or "function" calling. It takes three arguments, a return register, the name of the function to call, and the argument for that function.

Functions that need more than one argument passed to them, can get this by having the one argument that can be passed to be a tuple. Likewise with the return values.

An example that uses the `call` opcode is shown in the next section. It is the recursive factorial calculator.

3.2.6 Language constructs

Any language intended to do real work, should of course provide a way of *iterating*, and a way of *branching*. Both are provided in this language by means of the `loop` and the `branch` opcodes.

The `loop` opcode takes one argument. If the argument evaluates to something non-zero, the loop opcode will set the instruction pointer of the virtual machine to point to the actual loop code (the code that is inside the loop). If the argument evaluates to zero, the instruction pointer is instead moved to the instruction immediately following the loop construct.

A program approximating the machine precision could look like:

```
;;
;; Calculate machine precision
;;
entry "prec"
        decl floating,    ; r0 is floating point
             floating,    ; r1 is floating point
             integer,     ; r2 is a condition variable (integer)
             floating     ; the previous older value is floating too
        move r0, 1.       ; move a floating-point 1 into r0
        move r2, 1        ; move an integer 1 into r2
        loop r2           ; loop while r2 is nonzero
         move r3, r1         ; older <- old
         move r1, r0         ; old <- current
         div r0, 2.          ; current <- current / 2
         add r0, 1.
         add r0, -1.
         cmpgt r2, r1, r0    ; is old > current ?
        end                  ; if not, then older = precision
        return r3
end
```

The comments in the code should make it pretty clear what is going on. The algorithm is close to Malcolm's algorithm (see [3]), but not equivalent. Especially Malcolm took into account whether the machine rounded or truncated ("chopped") values. His algorithm would add a few extra lines to the example above.

Note, we add and subtract to/from the value after we have divided it by two. This is needed for the value not to evaluate to something around 10^{-308}, the smallest possible value in 64-bit IEEE floating-point. The code isn't that important, but it demonstrates a simple use of the `loop` construct.

The second construct is the `branch`. This is similar to the if/then/else constructs of other languages. The `branch` opcode takes one argument, just like the `loop`, and chooses one branch if the argument evaluates to something non-zero, and the other branch if the argument evaluates to zero. After executing any of the two branches, the instruction pointer of the virtual machine is set to point at the instruction immediately following the branch construct.

An example of a program using the `branch` construct, is the following recursive factorial calculator. Yes, the following code may not be a stunning example of expressive beauty, but it demonstrates the use of the `branch` construct rather well.

```
;;
;; This snippet calculates the factorial
;; of the argument given.
;;
entry "fact"
        decl integer,
             integer
        move r0, a0
        decr r0                    ; r0 = n-1
        cmpgt r1, a0, 1            ; r1 = a0 > 1
        branch r1
         call r0, "fact", r0       ; if a0 > 1 then r0 = f(r0)
        else
         return 1                  ; if a0 <= 1 then return 1
        end
        mult r0,a0                 ; r0 = a0 * f(r0)
        return r0
end
```

The code is given an argument in the $\mathtt{a}_0$ register, this is the number n of which want to calculate the factorial. If the number is greater than 1, the routine makes a recursive call to calculate the factorial of $n-1$, to multiply that result with n, and then return. If the number is not greater than 1, we just return 1. So we do not handle negative arguments, but both 0! and 1! should be covered.

One thing that should be noted about these constructs is, that we know *exactly* which variables they depend upon, and which they have the potential of changing. If a loop uses s as loop variable, the instructions in the loop uses $T = \{t_1, t_2, \ldots\}$, the entire loop construct uses $T \cup \{s\}$. The loop construct will touch (or have the potential to touch) the union of the variables each of the instructions inside the loop touches.

Branches can be looked at similarly. It is at all times extremely easy to find out what variables may be used or touched by any `loop` or `branch`, just as it is with single opcodes.

In fact, since we use balanced trees to actually represent the *uses* and *touches* sets of instructions and constructs, we can calculate the entire set of dependencies of a loop in almost linear time wrt. the number of instructions in the loop. This will be more thoroughly treated in chapter 4.

3.2.7 Indexing

Instructions that allow us to perform operations on entire vectors and matrices are not going to be enough for real-world problems. (In fact, it isn't even enough for the small testing problems I wanted to work with). It is very often necessary to work on a block of a vector or in a matrix.

In a language such as MatLab, this is handled elegantly in the language. One can write `A[:,2:3] = 1`, and the second and third columns of the `A` matrix will be assigned the value 1.

But what should we do in this virtual-machine-reflecting language ? It would be nice to represent everything like opcodes: An instruction, and a set of arguments. An opcode could be introduced to "cut" out the block we want to work with, assign it to some variable, and when we're done working on the variable we could call another opcode to have the results "pasted" in again.

That would not just be ugly, it would also be inefficient, since it would involve a larger number of opcodes which in turn would impose a greater overhead in the execution of the program. So that's not the way it is done.

Even though we do not allow normal mathematical expressions to be used as arguments to opcodes (e.g. you cannot enter `add r0, r1*8`), indexing has been implemented as a sort of "modifiers" to arguments. The second and third columns of a matrix register `r0` can be assigned the value 1 with the following instruction: `move r0[:][1:2], 1`. This changes the way the virtual machine sees and treats the first argument to the `move` opcode. Instead of the first argument being a register (as it would be without the indexing), it is now an *indexing*. The base variable of the indexing is the register `r0`, and the indexing consists of two intervals, one (the `[:]` index) which is a "full" interval (one that resolves to the full length possible in the given base variable), the second which is a bounded interval (a block bounded by a lower and upper index value) resolves to the integer interval $[1; 2]$. We use zero-based indexing, so `r0[:][1:2]` refers to the second and third column of `r0`. Optionally, one can also specify a step value, for example `r0[0:5:50]`, which refers to the elements at positions $\{0, 5, 10, 15 \ldots 50\}$ of the register `r0`.

A small program demonstrating this indexing could be:

```
;;
;; Indexing example
;;
entry "indexing"
        decl  matrix
        zero r0, 10
        move r0[3:6][3:6], 1
        move r0[:][0], 2
        move r0[:][9], 3
        move r0[0][:], 4
        move r0[9][:], 5
        return r0
end
```

This is also a simple example showing how dimensions of variables (registers) can change dynamically at run-time. The `decl` opcode only tells the virtual machine that `r0` is a matrix. The `zero r0, 10` opcode initializes `r0` with a 10×10 matrix holding zeros.

It returns the matrix

$$\begin{pmatrix}
4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\
2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\
2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\
2 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 3 \\
2 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 3 \\
2 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 3 \\
2 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 3 \\
2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\
2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\
5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5
\end{pmatrix} \tag{3.1}$$

3.3 CSP and job distribution

Communicating Sequential Processes is a way of describing complex parallel systems in terms of simple sequential processes that communicate with each other. A good introduction to CSP can be found in [1].

This section is a more loose theoretical discussion about interpretations or representations of parallel systems.

3.3.1 CSP and the virtual machine

Many parallel processes can easily be described in terms of sequential processes that at certain times exchange information (in our case, data and code). Since each process is sequential, and is thought of as having it's own private memory, no process can change variables used by other processes. The *only* way one process can influence another process is by sending the other process a message. This simplifies the design of a parallel system immensely.

Although it may sound like CSP is only useful on distributed memory architectures, that is not the case. A program designed as a set of communicating sequential processes can easily be implemented on a shared memory system. The only difference being, that the message passing between the processes can now be implemented more efficiently, now that processes (or "threads") can actually share some memory.

The nodes that make up our distributed virtual machine can be thought of as sequential processes that communicate either when they receive a new job (a task description containing code and data), or when they hand back a result.

The parallelization engine relies heavily on this property. Each single opcode is thought of as a message (the opcode and the data it uses), that can be sent to some other sequential process (a node in the cluster), to produce a reply message (the data the opcode touches, our result).

What the parallelization engine actually does is, to group the opcodes so that dependencies between opcodes are respected, but in a way so that the number of opcode "groups" matches the number of spare nodes in the cluster. This way we can see each opcode group (what we have called a "task" elsewhere) as a single message containing the list of opcodes and the data they depend upon. These messages are sent to the nodes, and the sender will then expect a reply message holding the result data.

3.3.2 Parallelization paradigms

The way we do parallelization in the TONS system is much different from how parallelization is done in both MPI/PVM and with threaded programming.

Distributed memory (MPI/PVM)

A typical MPI or PVM program will typically run in cycles that resemble the following:

Node_0		Node_1
Data	$\rightarrow$	
	$\leftarrow$	Data
... computation ...		
Data	$\rightarrow$	
	$\leftarrow$	Data
$\vdots$	$\vdots$	$\vdots$

The actual code that does the computation is at all times present on all nodes in the cluster. Each node is designed to do a little part of the large computation, and all nodes will sooner or later (often repeatedly during the computation) need to tell the others about their results.

The main concerns when parallelizing with MPI or PVM, is where to locate the data. The code is ubiquitous, so the only two things we care about is that all processes have the data they need when they need it, and that no dependencies between data are broken.

The latter concern is probably what leads to most bugs (at least in my personal experience) when programming with MPI. It is easy to distribute all of the needed data among all processes, and it is easy to redistribute the data once it has changed. But making sure that no process actually starts using any of the older data (which is still present on that particular machine) can be tricky, especially when one tries to optimize the communication for better performance later on.

Shared memory (threads)

A threaded program will often resemble something like:

Node_0		Node_1
.... synchronization		
computation		computation
.... synchronization		
computation		computation
$\vdots$	$\vdots$	$\vdots$

The "synchronization" can be both a real synchronization but it can also just be seen as one of the threads holding a mutual exclusion semaphore that effectively blocks all other threads that try to grab the same semaphore. Mutual exclusion semaphores are sometimes needed when accessing data in shared memory, to guarantee that no other threads use or modify the data one thread is in the process of changing.

Again, no code is distributed, since the entire program is running on one machine.

Parallelizing with threads is fairly error-prone, since there is *no* protection of data access except for the protection the programmer puts in herself. Contrary to the MPI / PVM case, where it is easy to forget to update all data, a threaded program is more likely to accidentally update data that is being used by another thread, without that thread knowing about it.

The distributed virtual machine

A typical execution of some parallel computation on the distributed virtual machine will resemble something like:

Node_0		Node_1
Code	$\rightarrow$	
Data	$\rightarrow$	
computation		computation
	$\leftarrow$	Result
Code	$\rightarrow$	
Data	$\rightarrow$	
computation		computation
	$\leftarrow$	Result
$\vdots$	$\vdots$	$\vdots$

The programmer of course doesn't even have to know that this is happening. Still the parallelization method is important since it is what guarantees us that the system is working. It also has very strong ties to the virtual machine language, as stressed earlier.

Because we bind code and data so strongly together, it is always clear exactly what code changes what data, and what code depends on what results.

We can never forget to update a variable with a result from another node, because we will be awaiting the reply message from that other node before we can continue with our next calculations (the next opcodes depend on the last result).

Also, a result can never be changed in a node without it knowing about it, since it has to read the data by itself.

So the two main sources of errors from MPI / PVM and threads do not exist when we choose to see the nodes in our cluster as communicating sequential processes, and the instructions executed and the data they use as messages passed back and forth between the processes.

An excellent toy language that demonstrates the powers of parallelization with CSP is *Concurrent ML*[2]. It is an implementation of the ML programming language with parallelization extensions. ML is a functional language, so by nature it lends itself well to parallelism using CSP.

[2] See `http://cm.bell-labs.com/cm/cs/who/jhr/sml/cml/` for more information

Chapter 4

Parallelization

Paranoid schizophrenics outnumber their enemies at least 2:1
— Unknown

In this chapter there will be a description of *how* we actually parallelize code, what we can do, and some ideas about what we might do in the future.

4.1 Properties of parallelization

This system is meant to be a tool for numerical computations. One of the most important properties in such a system is, in my humble opinion, that the system maintains the *numerical* properties of a problem.

Most algorithms in scientific computing rely on numerical properties. They are aware of the finite precision of computers, and they do what they can to minimize the noise introduced by the finite precision.

If a computing system executed code in a way that altered the ordering of the computations, or otherwise altered the way computations where done, the errors (that are always present, and which the users know about and takes into account in the numerical methods they use) would become unpredictable. If the parallelizer change a method, that method may no longer work as expected.

Therefore I define one key property the parallelizers must obey: The method must not change. That is, we will not allow any re-ordering of instructions, and we will not change the instructions.

If the user wrote $a = (b + \sqrt{b^2}) - 2b$, we *will* calculate $(b + \sqrt{b^2}) - 2b$, not $b + (\sqrt{b^2} - 2b)$ or anything else. The user's aren't stupid. If they want to test numerical inaccuracies of $(b + \sqrt{b^2}) - 2b$, then they know better than any parallelizer does.

This is also discussed in Section 2.3.2.

4.2 Communication and "tasks"

The word "task" is used to describe a block of code along with the data this code depends upon. The parallelizers will break the code into a number of tasks, and the scheduler will then distribute one task to each idle node.

Whenever a task is distributed, the code that makes up the task and the data the code depends upon is sent over the network to the destination node.

When the node completes its computations, the resulting data is sent back over the network.

Obviously this leaves room for optimization. It would be a good idea to let the scheduler be aware of the current localization of data, so that new tasks could be distributed to the nodes that already hold some of the data the new task needs, in order to minimize network traffic.

There has not been put much work into optimization of the network communications. The first concern was to get a system running with a decent virtual machine. Also, the current approach is a fairly clean implementation of a CSP specification of this system. This means, it never required much debugging, it is simple, and it works.

Because the parallelizers seek to group the instructions that depend on each other into tasks, the actual communication requirements are already minimized somewhat.

The network communication layer and the scheduler are two things that we cannot yet begin to optimize. We *must* have a virtual machine, with appropriate parallelizers, so that we can run larger programs that actually resemble real scientific computing code. Before that, any "tuning" of the communication strategies would be purely guesswork.

Besides the tuning at the more "strategic" level, there is also a large number of performance optimizations at a lower level that could be implemented. There is no reason why we shouldn't do asynchronous communication (eg. start receiving results from a number of nodes, while we're still waiting for others to complete their computations) for example. Also, the entire protocol encoding and decoding code is currently not very efficient.

The main efforts so far have been put into the virtual machine, and the automatic parallelization strategies in it.

4.3 Sequence parallelization

The most basic parallelization that happens is that sequences of instructions are parallelized, when possible. An example of such a parallelization could be the Fibonacci calculator. The serial and parallel pseudo-code is shown in figure 4.3.

The strategy behind the sequence parallelizer is actually quite simple. We have our sequential code as input, and we have another code sequence where we put our instructions as we finish treating them. Furthermore we have some temporary buffers where we hold the instructions as we try to arrange them in parallel tasks.

Our temporary buffers are: **main** which hold a number of instructions that are currently not parallel, but may have the potential to become a task that can run concurrently with other tasks. And we have the **tasks** array (it is a vector of instruction sequences), which is in use only when we are actually adding instructions to a set of concurrent tasks. The **tasks** structure will then hold an instruction sequence per active task.

We look at an instruction too see what data it depends upon (uses) and what data it modifies (touches). The current instruction is called C (for Current). We will search for the *most recent* instruction that touches any of the data the current instruction uses, and call this instruction T (for Toucher). If the instruction cannot be found, we assume that the data was touched somewhere outside the block of code we are

Serial	Parallel
if $r_0 > 2$ then	if $r_0 > 2$ then
$r_1 \leftarrow a_0$	$r_1 \leftarrow a_0$
$r_1 \leftarrow r_1 - 1$	$r_1 \leftarrow r_1 - 1$
$r_2 \leftarrow \text{fib}(r_1)$	parallel block start
$r_1 \leftarrow r_1 - 1$	task
$r_3 \leftarrow \text{fib}(r_1)$	$r_2 \leftarrow \text{fib}(r_1)$
$r_2 \leftarrow r_2 + r_3$	end
return r_2	task
else	$r_1 \leftarrow r_1 - 1$
return a_0	$r_3 \leftarrow \text{fib}(r_1)$
	end
	end
	$r_2 \leftarrow r_2 + r_3$
	return r_2
	else
	return a_0

Figure 4.1: Serial and parallel version of the Fibonacci calculator

currently looking at, and therefore we "fake" the toucher by setting it to be the first instruction in the **main** sequence.

Several cases can now arise, also depending on our current parallelizer state (whether we have parallel tasks already or not):

4.3.1 In case we have no parallel tasks currently

We introduce yet another marker (or pointer) into our code. We want to find the instruction I, that succeeds T if T was actually found in the code we know about, but is either before or equal to the start of our `main` sequence. In the case where we faked T to point to the beginning of the `main` sequence, we simply set I to point to the beginning of the `main` sequence.

So summarize: If T is found in code we know, then $\mathsf{T} < \mathsf{I} \leq$ beginning of `main`. Otherwise, $\mathsf{T} = \mathsf{I} =$ beginning of `main`.

If the sequence of instructions $[\mathsf{I};\mathsf{C}[$ is non-empty, it means that between the current opcode and the opcode that touches what we depend upon, there is something else which we don't care about. This is where we start two parallel tasks !

Sequence $[\mathsf{I};\mathsf{C}[$ non-empty

If $\mathsf{I} >$ beginning of `main`, then we move the instructions from the beginning of `main` up until (but not including) I into `parcode`. This code is not going to be parallelized.

If the code could be parallelized it would already have happened. The parallelizer is "greedy". It goes parallel the moment it sees a possibility for doing so.

Now we create two entries in the `tasks` array. Task 0 and task 1. We move all code in $[\mathsf{I};\mathsf{C}[$ into task 0. This code is something that succeeds the instruction that we depend upon, so we put it in a task for itself. Finally we move the current instruction C into task 1.

This is optimal. All the code in $[\mathsf{I};\mathsf{C}[$ *has sequential dependencies, so we cannot split that up further. If it did not have sequential dependencies it would already be parallel.*

Sequence $[\mathsf{I};\mathsf{C}[$ empty

There is no code between the instruction that touches the data we depend upon, and the current instruction. We move the current opcode C into the `main` sequence, and thus we do not go parallel.

Since $\mathsf{I} = \mathsf{C}$*, and we know that* I *is at most one instruction later than* T*, we know that there can't be any instructions between our toucher* T *and the current instruction* C*. The only thing we can do is to serially execute these two instructions, which means just putting them in* `main` *and hoping that they will be moved into a separate task later on.*

4.3.2 In case we already have parallel tasks

If we depend on data in an active task, that is T was found and it currently exists in an active task, or, if our number of active tasks is equal to the maximum size of our sub-cluster, then we decide not to spawn more tasks. Otherwise, we will go on seeking to exploit more parallelism.

When we do not spawn new tasks

First, we seek out *all* of our dependencies. We already know the most recent instruction T that touches something we use, but we need to know *how many* instructions in the currently active tasks that actually touch anything we use.

If the number of tasks the total set of instructions (currently in active tasks) we depend upon, is greater than one, we know that we depend on data that is touched by more than one active task. If this is true, we go serial. That is, we move all tasks to our output buffer, because this is code that generates results we need in order to carry on. The current opcode is moved into the `main`.

If the number of tasks that hold instructions that touch data we use is one, we place the current opcode in the task that already holds the toucher of the data it uses.

If the toucher of the data the current opcode uses is not in any current task, we simply add the opcode to the task that has the least opcodes.

This leaves room for enhancements. We should move the instruction into the task that either has the lowest computational cost, or in the task that will allow for the best parallelization for any following instructions. I can see how this could be done, but I haven't had the time to implement it.

When we do spawn new tasks

When we do not depend on any data in an active task, and when our number of active tasks has not yet reached the limit of our sub-cluster, then we will go on and spawn a new task.

A new task is simply added to the `tasks` array. The current opcode is inserted as the first instruction in the new task.

Again there is some room for optimization. It is not always feasible to spawn new tasks. I believe it is a good thing spawn as many tasks as we can, initially, but *when we have finished parallelizing a sequence we should look at the tasks and the estimated costs of executing them locally versus remotely. Tasks can very easily be contracted as we see fit, and a simple heuristic on this will definitely be a huge benefit over this "pervasive parallelization" strategy. Again, the hard part was to be able to detect the possibility of spawning new tasks. The easy thing will be to merge some of them together again.*

4.3.3 Overview

The code involved directly with the sequence parallelization is roughly 1300 lines of C++ code. Half of it is either generic code used elsewhere as well, or comments. The core of the sequence parallelizer is more like 600 lines of code, including comments. Most of this code is book-keeping code for handling the instruction sequences, the dependency sets etc. etc.

I have made heavy use of the C++ standard template library for implementing the sets, maps, lists of sets, and other fairly complicated but efficient data structures.

There are still optimizations that should be done in the code. There may be cases where we spoil the otherwise fine algorithmic complexity of the parallelizer. The following is a completely non-academic and unproven estimate of the complexity of the sequence parallelizer. It looks as though this estimate is possible to meet, but there may well be cases in the current implementation that blows this.

In a code sequence with p instructions, where the instructions have a total of q arguments, we should be able to parallelize the entire sequence in $\mathcal{O}(pq \log q + q \log q)$ time.

$\mathcal{O}(q \log q)$ is the cost of building up a balanced tree holding q arguments (dependencies). We use balanced trees to implement the dependency sets.

The $\mathcal{O}(pq \log q)$ is the cost of looking up q dependencies in our mapping between touched variables and the touchers, for p instructions. Again, the mapping is implemented using a balanced tree, and it is kept up to date as we move instructions between the various sequences.

The number of dependencies q is likely to be a small number compared to the number of instructions p, so effectively we may see run-time that looks more like $\mathcal{O}(pq \log q)$.

4.4 Loop parallelization

The second parallelization that was implemented identifies independent operations that take place in a loop, and move those instructions out into independent loops. Such an example can be seen in figure 4.4.

Clearly this way of parallelizing the loop is much faster than if we parallelized the independent operations inside the loop. That would require communication in every iteration thru the loop, and unless the independent operations where really computationally hard, we would end up doing a lot of communication and little real computation.

The parallelization of a loop is a step-wise process, where we gather more and more information, and continually decide whether it is possible to continue the loop parallelization. The steps are described in detail in the following.

Serial	Parallel
loop while r_0	parallel block start
$r_2 \leftarrow r_2 \times r_1$	task
$r_2 \leftarrow r_2 / r_0$	loop while $r_0 > 0$
$r_0 \leftarrow r_0 - 1$	$r_2 \leftarrow r_2 \times r_1$
$r_3 \leftarrow r_3 \times r_1$	$r_2 \leftarrow r_2 / r_0$
$r_3 \leftarrow r_3 / r_0$	$r_0 \leftarrow r_0 - 1$
$r_0 \leftarrow r_0 - 1$	$r_0 \leftarrow r_0 - 1$
end	end
$r_2 \leftarrow r_2 + r_3$	end
	task
	loop while $r_0 > 0$
	$r_0 \leftarrow r_0 - 1$
	$r_3 \leftarrow r_3 \times r_1$
	$r_3 \leftarrow r_3 / r_0$
	$r_0 \leftarrow r_0 - 1$
	end
	end
	end
	$r_2 \leftarrow r_2 + r_3$

Figure 4.2: Serial and parallel version of the multiplication/division loop

4.4.1 Identifying common instructions

In figure 4.4 some instructions are printed in blue. Those are the instructions that directly or indirectly influence the loop variable. Clearly, any instruction that influences the loop variable, must be propagated to all loops we create, otherwise the behavior of the new loops will be different from the original loop.

These instructions that influence the loop variable are called the *common* instructions. In the example it is simply the two operations that decrease the loop variable, but the case could have been less simple.

In the following, we consider our loop code (the code inside the loop) to be the only code visible to us (which is actually the case for the loop parallelizer in the current implementation).

We work with several sets here: The set C of common instructions. The set of variables $\mathcal{S}$ that influence the loop variable. The touches set $\mathcal{T}$ of the current instruction. And finally the uses set $\mathcal{U}$ of the current instruction.

Initially $\mathcal{S}$ is set to contain only the loop variable itself. We set the current instruction to be the *last* instruction in the loop code. We want to traverse the loop from the bottom and up (since the last instruction in the loop code is the one that has most recently had the chance to alter the loop variable, in the second and all following iterations).

We calculate the intersection $m = \mathcal{T} \cap \mathcal{S}$. This is the set of variables that currently influence the loop, and that the current instruction touches.

If m is non-empty, we add the current instruction to the set of common instructions. And, we assign $\mathcal{S} \leftarrow (\mathcal{S} \setminus m) \cup \mathcal{U}$. Or, in other words, we remove the variables the current instruction touches from the set of variables that influence the loop variable. Secondly, we add the variables it uses, to this same set.

The current instruction is then set to be the instruction preceding the current

"current instruction". Remember, we are traversing the loop code from the bottom and up. We go back to recalculating m.

This iteration is continued until we have been thru all instructions in the loop code.

The end result, is a set of common instructions that reflects exactly which instructions are needed to update the loop variable properly. Any remaining instructions are the ones that do the "real" work.

4.4.2 A quick heuristic

Although I have not focused much on whether it is feasible to parallelize anything, I did insert a rough heuristic in the loop parallelizer.

We know how many common instructions we found, and we know how many instructions the loop code contains. So, if the number of common instructions is more than half of the instructions in the loop, we decide not to parallelize. Instead we call the sequence parallelizer to have the loop code parallelized if at all possible.

This heuristic works for now, but we will need something more sophisticated later. If some cost estimate is developed (it is also needed in the sequence parallelizer), it would be easy to "plug in" here too.

4.4.3 Going parallel

At this point we are certain that we want to parallelize. Still it is not entirely possible that we actually can. We call the sequence parallelizer to get the loop code parallelized.

If the sequence parallelizer returns something that is not strictly *one* parallel block with a number of tasks, then something is wrong. If we receive any serial sequences, we are beginning to duplicate code into separate loops, and that is something we don't want. We exit with the parallelized loop code here, if that happens.

We may want to duplicate serial code, if it the execution of it is fast enough, and if the cost of the code we actually can parallelize justifies this. It should be simple to implement, but we need some sort of heuristic for this. We need an execution cost estimate.

4.4.4 Creating the loops

So we have a parallel block containing a number of independent tasks, and that's it.

Our next move is to create new loop constructs, one for each task returned by the sequence parallelizer, and move the code from the tasks into the new loops.

4.4.5 Fitting in the common instructions

We know that each task will either contain *all* of the common instructions (this is true for one task) or *none* of the common instructions (which will be true for the remaining tasks).

The common instructions must reside in the same task, because they can per definition not be entirely independent (remember, they all contribute to the change of the same variable). Some of the common instructions can be independent and thus *placed in separate tasks, but if this was exploited by the sequence parallelizer, we would have known earlier, since we would not have received a number of independent tasks then.*

To identify the task that contains the common instructions, we should simply search for any instruction that touches the loop variable, that is, an instruction that contains the loop variable in it's touches set. Any instruction that touches the loop variable is per definition a common instruction, and the remaining common instructions are bound to reside in the same task.

Note that we make a lot of assumptions on correctness of the sequence parallelizer and other things here. This code is not fault tolerant, and is therefore also fitted with a large number of sanity checks throughout the code. Of course code should not be faulty, but this code must be absolutely correct not to break completely, something that makes debugging interesting.

When we have identified the task that contains the common instructions, we append the common instructions to all remaining tasks.

The common instructions must be appended, not prepended, because we know that even though the code may use the data touched by the common instructions, it will never depend on the results from the common instructions before the next iteration in the loop.

If this was not true, the sequence parallelizer could not have parallelized the code the way it did. It distributed the common instructions out into one task, and all the remaining tasks will be using the values of the variables touched by the common instructions as they where before the common instructions touches them, for the current iteration.

4.4.6 Overview

This parallelizer was surprisingly easy to implement. It depends heavily on the functionality and, not least, the correctness of the sequence parallelizer. The code takes up around 250 lines including comments, probably less than this description.

Curious things happen, if one supplies code to the parallelizer that does not make sense. If the loop code touches variables that are never used, various strange results can occur. This is not a huge problem par se, but it is something that must be taken into account, at least when we get a reasonable high-level front end language and a compiler for that.

It is actually another strength of this language, that we can easily spot superfluous instructions and remove them. This is not currently done, but it's another one of those fairly easy to implement tasks, that are just waiting to happen.

4.5 Instruction expansion

Consider the operation $A[0:9] \leftarrow f(A[0:9])$. This code should be fairly easy to translate into the sequence:

$A[0:4] \leftarrow f(A[0:4])$
$A[5:9] \leftarrow f(A[5:9])$

if we know that $f(\{A_0, A_1, \ldots\})_i$ depends only on A_i. The code is the equivalent of $\forall i : A_i \leftarrow f(A_i)$.

Since $f(\cdot)$ will be some combination of the instructions provided by the virtual machine, we should be able to actually determine if $f(\cdot)$ has that desired property.

This is not currently implemented, but I fail to see why it should not be possible. We may not be able to spot every special case of instruction combinations, but it is often the simple cases that are the interesting ones, since the majority of computing code spends it's time in fairly simple loop constructs. If we can just catch enough cases to benefit in most situations, it will be good enough. But I'm not yet sure if there would be any problem in detecting all cases.

The resulting code will lend itself very well to the sequence parallelizer.

4.6 Nested loops

A more difficult, but also very interesting case to parallelize, is the nested loops.

We want to be able to transform code like

loop while $i \in [0; 9]$
  loop while $j \in [0; 9]$
    $A[i, j] \leftarrow B[i, :] \cdot C[:, j]$
    $j \leftarrow f(j)$
  end
  $i \leftarrow g(j)$
end

into

loop while $i \in [0; 4]$
  loop while $j \in [0; 9]$
    $A[i, j] \leftarrow B[i, :] \cdot C[:, j]$
    $j \leftarrow f(j)$
  end
  $i \leftarrow g(j)$
end
loop while $i \in [5; 9]$
  loop while $j \in [0; 9]$
    $A[i, j] \leftarrow B[i, :] \cdot C[:, j]$
    $j \leftarrow f(j)$
  end
  $i \leftarrow g(j)$
end

This would lend itself well to the sequence parallelizer, since we have two independent sets of nested loops.

Both loop structures use the B and C variables, but they don't touch them. The two structures touch two different subsets of A, so they really can be parallelized and run independently of each other.

This parallelization might not be as hairy to implement, as a first look on the wanted transformation could suggest.

Actually this parallelization is not much different from the core of the aforementioned instruction expansion. The only difference is the complexity of our function $f(\cdot)$, but the fact that we at all times work with sets of dependencies should make the actual complexity or size of some part of the code something we don't worry about.

Although this parallelization is not yet implemented, it may already be implemented partially. Because especially the dependency checking (which is the same dependency checking used in all parallelizers, since dependency checking is implemented in the opcode definitions, not in the parallelizers) is not yet completely implemented, we currently assume some dependencies that are actually not there. This makes the parallelizers work correctly (but not optimally) until I get the dependency checking routines completed.

4.7 Where to go next

This is a short description of what issues I consider important to address if this work is continued. It is a description of current "work in progress" and work that should be started as soon as possible.

The main issue in getting these suggested extra parallelizations working, seems to be to implement the dependency detection routines in a way so that they handle all cases of indexing that can possibly be handled.

Once we have a complete dependency (uses and touches set generation) routine implementation, the parallelization routines will do a better job without much need for change.

There may be cases that will require a change in the current parallelizers. And there definitely is a need for the "instruction expansion" functionality. But I believe that this work can be done in a fairly short amount of time, as the building blocks are falling into place.

As a side note, the loop parallelizer was designed, implemented, and tested in something like four days. I am very optimistic with regards to further expansion on the functionality of these parallelizers.

However, I don't think the parallelizers will need that much more work. The main burden is going to be the completion of the implementation of the dependency detection routines. The main problem here, is to decide whether variables "intersect". This is needed and used throughout the parallelizers, and is basically a decision of whether $A \cap B \neq \emptyset$.

Chapter 5

Performance

All computers wait at the same speed
— Unknown

This chapter will give examples of real world performance of the virtual machine.

5.1 VM performance

Although the virtual machine operates with instructions mostly at a much higher level that the usual computer processor instructions, there will be code in any program where we actually will get very little computational work done with a large number of instructions.

The need for such instruction-heavy code has been minimized by allowing most instructions to work on many different types of data, and by introducing the indexing scheme.

Still, it is absolutely vital that the virtual machine has a good performance not only on the single heavy instructions (such as the `mult` opcode on matrices), but also executes simple instructions at a high speed (such as `mult` on scalars).

A few comparisons have been done. Different sample programs are used to benchmark the performance on different computing platforms or languages. I mainly use MatLab, C, and Perl for comparison.

MatLab is used because it is in widespread use, and also executes interpreted code (according to the definition of "interpreted" in Section 1.4). C because it has virtually no instruction execution overhead. And Perl because this is also a virtual machine, but it is *extremely* fast.

I did not include Fortran tests because C should yield roughly the same performance.

The implementation of all the tests can be found in Appendix A.

5.2 The Fibonacci test

The Fibonacci calculator does only simple operations, and thus the performance on solving this problem will reflect the overhead in the virtual machine, and not much else. The code is found in Appendix A.1.

This table shows the execution times for calculating the fib(30) number, the 30th Fibonacci number.

Language	Time (seconds)
C	0.22
Perl	26.5
MatLab	183.4
TONS	40.0
$TONS^2$	29.7

The $TONS^2$ number, is for TONS running on two processors on an SMP machine.

Our virtual machine places itself somewhere between the performance of Perl and MatLab, but closest to Perl. I'm pretty happy with that. Especially because there hasn't been done any work on fine tuning the performance on the virtual machine yet. It is fairly well performing by design, but there are definitely tuning opportunities left.

The reason why we don't scale better when going to two CPUs, is that the first node (which computes fib(28) and delegates fib(29) to its slave), is spending some time waiting for the node to return it's result. This is an implementation issue mostly, since the nodes should be able to communicate to each other when someone becomes idle. It is, however, also a design issue, since this communication should not be taken lightly upon. This is discussed in chapter 2.3.4.

This problem will scale to a large number of processors. Because of the re-scheduling problem, we do not see a linear speedup, and the actual speedup will decrease as we add more nodes. But I've run this program successfully on 9 nodes, though with a more disappointing speedup. But the speedup problem will – to a large extent – be solved by implementing a smart re-scheduling strategy, so nodes can delegate jobs to other nodes, as they become idle.

5.3 Matrix multiplication

The second test is a simple repeated matrix multiplication. The code is found in Appendix A.2

The program is simply a repeated matrix multiplication of two different 100×100 matrices, with a third matrix. The multiplication series is repeated 100 times.

This time, I only tested MatLab and TONS.

Language	Time (seconds)
MatLab	9.2
TONS	20.5
$TONS^2$	10.9

That looks more like a reasonable result from MatLab. The matrix multiplication I implemented is a very fast hack, and it is far from being fast. Obviously, MatLab has efficient multiplication routines implemented, so even though their virtual machine, or interpreter, is slow as molasses, MatLab is twice as fast as TONS, on one CPU.

We scale almost linearly in performance, as the extra CPU is taken into use. This is because the amount of work done in the two independent loops (which of course the loop is transformed into as we add the second CPU) is the same. No node server waits for the other.

If we added a third CPU, it would never be taken into use. This code just does not parallelize onto three or more CPUs, with the current state of the TONS parallelizer.

I do not see an easy way of parallelizing this program any further, at the virtual machine opcode level, without changing the order in which things are happening, which we refrain from.

We could however, sometime in the future, implement parallel versions of the instructions, so that if nodes where available, the matrix multiplication could run in parallel on several nodes. But there are two things to this. It is not "automatic parallelization" in the sense that the virtual machine code is parallelized, it is simply a matter of exchanging the actual implementations of the instructions with parallel ones. Secondly, implementing parallel matrix operations in C++ is way beyond the scope of this work. It is an area in which there has been a lot of research, and it should be fairly simple just to plug in well-known efficient parallel matrix routines, once we get the actual free-node/busy-node communication done.

Chapter 6

Conclusion

If I have seen farther than others,
it is because I was standing on the shoulders of giants.
— Isaac Newton
In computer science, we stand on each other's feet.
— Brian K. Reid

When I started on this project, I thought I would spend 30% of the time developing a fast virtual machine, and the remaining 70% on the parallelization. Boy I was wrong.

It turned out, that a virtual machine with a language that lends itself well to parallelism is absolutely vital, if one wishes to do on-the-fly automatic parallelization. The virtual machine was rewritten several times during the early stages of development, simply because it turned out badly with respect to further parallelization of the code it ran.

That being said, I think the current virtual machine has a viable design. It is simple to detect dependencies, and it is not too hard (computationally) to actually generate parallel versions of the functions submitted to the scheduler, using this internal representation of programs.

The current implementation is able to detect simple cases of parallelism. The sequence and the loop parallelizers have been discussed, along with their current short comings. The important thing to remember is, that there should be nothing in the *design* of the system that makes it very hard to implement those missing features. The language is very predictable with regards to data dependencies, and that is absolutely vital for any further refinement of the parallelizers.

I believe I have come to the conclusion, that automatic parallelization is not just a matter of a single parallelizing translator or execution entity. We will not see good automatic parallelization before we build an automatically parallelizing computing environment, something that includes everything from end-user language to compute-server software.

This project is headed in the direction of an automatically parallelizing distributed computing environment. Such an environment is not built in half a year, at least not by me. But further work on this project, or a similar project, may lead the way to efficient cluster computing in the future.

Jakob Østergaard,
Lyngby, 2nd of July, 1999

Appendix A

Benchmarks

All the performance tests was run on a dual processor Intel Pentium Pro 150 MHz machine with 256 MB of RAM. The machine runs the Linux 2.2.7 kernel, and all performance tests were done in single-user mode (to assure that no daemons or network traffic etc. would influence the results).

A.1 The Fibonacci calculator

The Fibonacci test was run with C, Perl, MatLab and on the TONS system. All programs implement a simple Fibonacci number calculator. The n'th Fibonacci number is defined as:

$$f(n) = \begin{cases} n & : \quad n \leq 2 \\ f(n-1) + f(n-2) & : \quad \text{otherwise} \end{cases}$$

A.1.1 The C implementation

This code was compiled with the EGCS 1.1.2 compiler using the following compilation options: `-O6 -mcpu=i686`

```
#include <stdio.h>

int fib(int n)
{
  if(n > 2)
    return fib(n-1) + fib(n-2);
  return n;
}

int main(int argc, char** argv)
{
  int res;
  printf("Calculating fib(30)...");
  fflush(stdout);
  res = fib(30);
  printf("done.\n");
  printf("fib(30) = %i\n", res);
}
```

A.1.2 The Perl implementation

This code was run on the 5.005 version of the Perl interpreter, compiled for an Intel i386 Linux platform.

```
#!/usr/bin/perl -w

sub fib {
    my ($n) = @_;
    return $n if ($n <= 2);
    return fib($n-1) + fib($n-2);
}

my $arg = $ARGV[0];

print "fib($arg) = ".fib($arg)."\n";
```

A.1.3 The MatLab implementation

This code was run on the 5.1.0.421 version of MatLab for Linux.

```
function r = fib(n)
if( n > 2)
 r = fib(n-1) + fib(n-2);
else
 r = n;
end
```

A.1.4 The TONS implementation

This code was run on the 0.0.6 release of the TONS program available via anonymous FTP from `ftp://ftp.sslug.dk/pub/TONS/`.

```
;;
;; Fibonacci caluclator
;;
;; f(n) =    f(n-1)+f(n-2)    if n > 2
;;      =    n                otherwise
;;
entry "fib"
        decl integer,   ; r0
             integer,   ; r1
             integer,   ; r2
             integer    ; r3
        cmpgt r0, a0, 2
        branch r0
         move r1, a0
         decr r1
         call r2, "fib", r1
         decr r1
         call r3, "fib", r1
         add r2, r3
         return r2
        else
         return a0
        end
end
```

A.2 Matrix Multiplication

The matrix multiplication program does a repeated matrix multiplication of two matrices:

$$A \leftarrow AC \qquad B \leftarrow BC$$

The matrices have dimension 100×100, and the multiplication loop is iterated thru 100 times.

This test was only done on MatLab and TONS.

A.2.1 The MatLab implementation

```
d = 100;
i = 100;
A = zeros(d,d) + 0.01;
B = zeros(d,d) + 1;
C = zeros(d,d) + 10;
for(s = 1:i)
 B = B * A;
 C = C * A;
end
C = C + B;
```

A.2.2 The TONS implementation

```
;;
;; Small performance test on
;; parallel operations...
;;
entry "matrixperf"
        decl  integer, integer, matrix,
              matrix, matrix
        ; Set matrix dimension
        move r0, 100
        ; Set number of iterations
        move r1, 100
        ; Initialize matrices
        zero r2, r0
        add  r2, 0.01
        zero r3, r0
        add r3, 1
        zero r4, r0
        add r4, 10
        ; Run the loop
        loop r1
                mult r3, r2
                mult r4, r2
                decr r1
        end
        ; Add return values
        add r4, r3
        ; And exit
        return r4
end
```

Bibliography

[1] **Communicating Sequential Processes** *C.A.R. Hoare.* Prentice-Hall International Series in Computer Science, 1985.

[2] **Principles of Protocol Design** *Robin Sharp.* Prentice-Hall International Series in Computer Science, 1994.

[3] **Algorithms to reveal properties of floating point arithmetic** *Michael A. Malcolm* Association for Computing Machinery, Inc. Numerical Mathematics, November 1972.

[4] **Lex and Yacc** *John R. Levine, Tony Mason, Doug Brown.* O'Reilly & Associates Inc. 1995.

[5] **The C++ programming language** *Bjarne Stroustrup* Addison-Wesley, 1997.

Also, the following Internet resources have been a source of inspiration for this project:

`http://beowulf.gsfc.nasa.gov/`
The homepage of the Beowulf project at NASA. Especially the associated Beowulf mailing list is interesting as it is regularly a source of good cluster computing discussions.

`http://cnls.lanl.gov/avalon/`
One of the many examples of what is possible with cluster computing. This is the homepage of a 140 processor Alpha based cluster. This is 47.7 GFLOPS on a Linpack benchmark, for $ 300.000 in 1998.

The TONS project has a homepage at `http://www.sslug.dk/TONS/`. The source code for the latest released implementations is also available there for download. The homepage is kindly hosted by Skåne/Sjælland Linux User Group.