Advances In Programming Languages: proceedings of the 2009 SICSA summer school.

January 27, 2011

List of Algorithms

List of Figures

List of Tables

Chapter 1

SIMD and Multi-core Parallelisation in an Imperative Language.

Paul Cockshott, Tamerlan Tajadinov

1.1 Hardware Context

One of the most significant trends that can be observed in the modern processor design is towards an increased parallelism. Single Instruction Multiple Data-stream (SIMD) computing has a long history[23, 12], but in the 1990s, a technique that had previously only been applicable in large machines began to be applied to microprocessors[25, 21]. Primarily aimed at graphics applications, SIMD instructions were introduced for the x86 in Pentium MMX processors allowing saturated arithmetic to be performed on up to eight 8-bit or two 32-bit integers at a time. The concept was further developed in SSE2 processors[5] by allowing floating point operations to be performed on up to four single precision or two double precision floats at a time, while also doubling the number of possible integer operands. The IBM/Sony Cell [18]processor used in the PlayStation 3 also operates on vectors of data of similar size. Moreover, the next generation of Intel GPGPUs codenamed Larrabee also follows the trend by allowing up to 64 8-bit integers or 16 single precision floats to be operated on with a single instruction.

While traditionally an increase in the performance of a CPU used to be associated with an increased clock speed, this historic trend is no longer observed as much. An increase in the clock speed demands a sharp increase in the power consumption and leads to a rise in the heat dissipation, thus the clock speed is limited by reasons of thermodynamics to presently around 3.5GHz as in case of Wolfdale DP generation of Xeon processors. This limitation is now increasingly commonly compensated for by planting multiple cores on a processor chip - a solution that has originated from high performance systems the performance of which frequently benefits from multiple CPUs. As the number of cores in a system increases, the CPU clock speed tends to be reduced by the processor designers, thus the 4-core Core 2 Quad processor operates on frequencies of up to 3.2GHz depending on the model of the chip compared to its single core predecessor Pentium 4 running on clock speeds of up to 3.8GHz several years prior. In mobile computing demands on reduced power consumption have retained CPU frequencies at much lower rates of around 1-2.5GHz and dual and even quad core processors are widely used.

The above developments in the topic of processor design have introduced further demands in compiler design. The majority of imperative programming languages, especially those derived from C, typically operate on a single element at a time, thus, for example, should we wish to add two vectors of integers together each pair of elements will be added one at a time, which is not in line with the hardware capabilities allowing for arithmetic operations to be performed on multiple operands at a time. Furthermore, unless multiple threads are explicitly spawned, only one of the cores is likely to be utilised, leading to the reducing processor clock speed being the main performance indicator for applications written in these legacy programming languages. The remainder of this article discusses Vector Pascal, an imperative language designed to utilise the increasing capacity of modern processors through hardware parallelism.

1.2 The Example Language

Vector Pascal[4] was developed by the computer vision group at the University of Glasgow as a language for high performance image processing. It is an extension of Pascal[17, 13] with array language concepts. Array programming languages arose from the work of Iverson. During the 1950s, when the latter was working as a doctoral student for the Nobelist Leontief on the computation of the US input output tables he developed Iverson's Notation, a concise and unambiguous mathematical notation for expressing matrix computation. The notation was taken up by IBM and released as the computer language APL[14]. What distinguished APL from other contemporary languages was both its rather abstruse character set, and its ability to concisely express calculations over whole arrays in a single assignment statement. It was an untyped interpretive language. Pascal on the other hand is famously a strongly typed language. There have been many interpretive successors to APL, the most popular ones being Matlab[19] and J[15]. Introduction of array language techniques into typed imperative languages started with versions of FORTRAN[22, 9, 2]. A number of other compiled array languages have subsequently been developed[24, 1, 11]. What has distinguished Vector Pascal from these is its particular emphasis on efficient targeting the parallel features embodied in commodity microprocessors: initially SIMD instructions, and more recently multi-core facilities. In what follows we describe only those extended features of the language that relate to parallelism. Extensions relating to the type system, mathematical character sets etc, are ignored.

1.2.1 Extend array semantics

Standard Pascal allows assignment of whole arrays. Vector Pascal[4] extends this to allow consistent use of mixed rank expressions on the right hand side of an assignment. For example, given:

```
r1:array[0..7] of real;
r2:array[0..7,0..7] of real
```

then we can write:

```
1. r1:= 1/2;
2. r2:= r1*3;
3. r1:= r1+r2[1];
```

1.2.2 Equivalent loops

Line 1 assigns `0.5` to each element of `r1`.
Line 2 assigns `1.5` to every element of `r2`.
In line 3, `r1` is incremented with the corresponding elements of row 1 of `r2`.
These are defined to be equivalent to the following standard Pascal loops:

```
1'. for ι0:=0 to 7 do
      r1[ι0]:=1/2;
2'. for ι0:=0 to 7 do
      for ι1:=0 to 7 do
        r2[ι0,ι1]:=r1[ι1]*3;
3'.  for ι0:=0 to 7 do
      r1[ι0]:=r1[ι0]+r2[1,ι0];
```

The compiler has to generate an implicit loop. In the above ι_0, ι_1, t are temporary variables created by the compiler. The implicit indices ι_0, ι_1 etc are accessible to a coder using the syntax `iota[0], iota[1]` etc.

1.2.3 Data reformatting

Given two conforming matrices a, b
the statement

```
a:= trans b;
```

will transpose the matrix b into a.
For more general reorganisations you can permute the implicit indices thus

```
a:=perm[1,0] b ;
{ equivalent to a:= trans b }
z:=perm[1,2,0] y;
```

In the second case z and y must be 3 d arrays and the result is such that z[i,j,k]=y[j,k,i]. It is clearly desirable to do this in a way which prevents the creation of temporary arrays. We will discuss below the general strategy to minimise the creation of temporaries.

Given `a:array[0..10,0..15] of t`; then

```
a[1]          array [0..15] of t
a[1..2]       array [0..1,0..15] of t
a[][1]        array[0..10,0..0] of t
a[1..2,4..6]  array[0..1,0..3] of t
```

1.2.4 Implicit mapping

Maps are implicitly defined on both operators and functions.

If `f` is a function or unary operator mapping from type T_1 to type T_2 and `x` : `array of` T_1 then `a:=f(x)` assigns an array of T_2 such that `a[i]=f(x[i])`. Similarly if we have `g(p,q:`T_1`):` T_2,then `a:=g(x,y)` for `x,y:array of` T_1 gives `a[i]=g(x[i],y[i])`

1.2.5 Pixel Arithmetic

Pixels are a predefined data type, represented as 8 bit signed fixed point binary fractions. The numerical range of pixels is from -1 to +1. Arithmetic on pixels is inherently saturating. The existence of types supporting saturated arithmetic is an advantage in image processing.

1.3 Parsing and Automatic Mapping of Operators and Functions

Translation takes place using a modification of the classic single pass compiler approach used by Wirth[17], and described in [8]. In this approach a recursive recognising function is declared for each non-terminal in the grammar of the source language. On success the recognising function outputs, as a side effect, a stream of byte codes for a machine independent intermediate code. We have modified the approach in two ways in both order to parse array expressions and also to allow the efficient translation of the array expressions to parallel code.

	compiler		code generator	
		tree-form		machine
pascal source	→	intermediate	→	specific
		code		assembler

Since the work of Budd[3] it has been known that efficient compilation of an array language requires that one eliminate the formation of temporary arrays in the course of array expressions. It turns out that a simple extension of recursive descent parsing techniques allows this.

Consider a simplified version of the Vector Pascal grammar for assignments :

```
<ass>    ::= <variable> ':=' <exp>
<exp>    ::= <term><addop><term>
<term>   ::= <factor><multop><factor>
<factor>::= '(' <exp> ')'
          | <variable>
          | <reduction> <factor>
          | 'trans' <factor>
```

Consider the following example produced by the grammar:

```
z:= y+ \+(A * B );
```

where `y,z` are vectors and `A, B` matrices. `\` is the reduction functional which reduces an expression by an operator. Thus `\+` forms the sum along the rows of the matrix formed by the Hadamard product of `A` and `B`. A naive compilation would generate 3 temporaries for this, a matrix for `A*B`, a vector for the sum of `A*B` then a final vector as a result of adding y.

A standard recursive descent parser would have functions `ass, exp, term, factor : typedescriptor` which would read the compiler input stream, return the type of the expression or sub expression they recognised, and as a side-effect output a stream of intermediate byte-codes. In order to handle array expressions we modify these so that they are now of the form:

```
function exp(Iotalocations:vector):IlcgTree;
```

Each parsing function now takes as a parameter a vector of addresses of memory locations to be used as loop index variables, and instead of returning a type, it returns a intermediate code tree. The assignment function inspects its first symbol - a variable and determines the rank of the variable. It then creates a vector of index locations with respect to the current frame pointer and passes this vector in to the call it makes on exp. The length of the vector of index locations is determined by the rank of the variable being assigned to. The following pseudo code explains the technique:

```
function ass:IlcgTree;
  musthave(Id_symbol);
  id:=currentsymbol;r:=rankof(id);
  v:= createIotaVector(r);
  e:= exp(v);
  dest:= locationof(id);
  for i := 1 to r do
    dest:= subscript(dest,v[i]);
  return createnestedforloops(dest,v,e);
```

To understand what is produced refer back to section 1.2.2. The exp function in turn passes the vector of index locations to its terms:

```
function exp(Iotalocations:vector):IlcgTree;
  t1:= term(Iotalocations);
  if have ( addop ) then
     op:=currentsymbol;
     t2:=term(Iotalocations)
     return dyad(t1,op,t2)
  else return t1;
```

At the bottom level of the parse, the function to recognise a variable uses the index vector to reduce any array variable to a scalar.

```
function variable(Iotalocations:vector):IlcgTree;
  musthave(Id_symbol);
  id:=currentsymbol;
  r:=getrank(id);
  u:=upbound(Iotalocations);
  v:= locationof(id);
  for i := 1 to r do
    v:= subscript(v,iotalocation[i+u-r]);
  return v;
```

The `subscript` function constructs and returns an intermediate code tree to perform the subscription of the variable by the contents of the memory location described by `Iotalocations`. The effect is that the exp function will always return intermediate code that will evaluate to a scalar rather than an array. This scalar result is then embedded in a for loop as specified in section 1.2.2. All intermediate array values are eliminated allowing the code generator to use conventional register optimisation techniques for array valued expressions. There are a couple of special cases worth noting:

1. Reduction factors. On encountering a reduction functional the `factor` function creates a new vector of indices. If the initial vector of `Iotalocations` was of length n this will be of length n+1 with the last element being the address of a new index variable to be used in a reduction loop. This longer vector of indices is then passed into a recursive call of `factor`. The code generated for the reduction is always for a scalar operation.

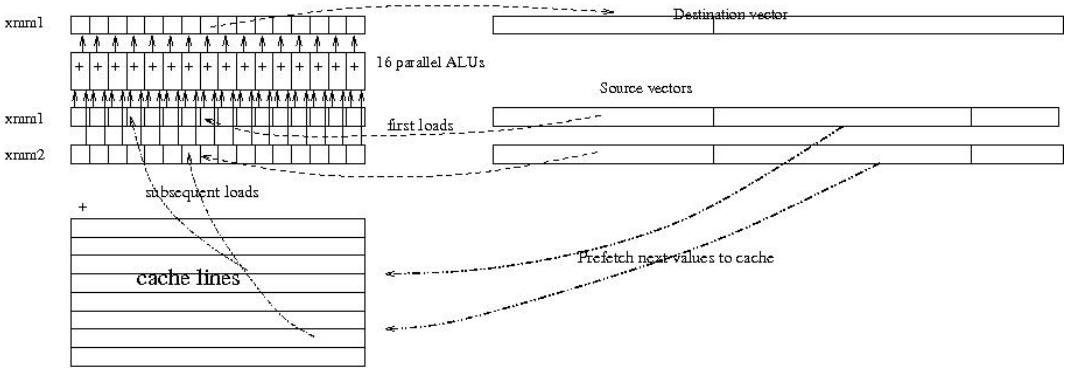

Figure 1.1: The Streaming SIMD model of processing.

2. Transpositions and permutations. These are performed on the index vector at compile time before passing it in to a recursive call of the `factor` parsing function. There is no additional run time cost associated with the transpose operation.

The vector of index locations is similar in principle to the technique used in Single Assignment C [11] for array de-referencing, but whereas in SAC an index vector is explicitly present in the indexing semantics of the language, in our case it is an implicit vector existing only at compile time. Vector Pascal allows the index vector `iota` to be used in sub-scripted form within a factor thus

```
p:= q[iota[0]+1]*0.5+q[iota[0]-1]*0.5;
```

would be legal where

```
p:array[1..n] of real;q: array[0..n+1] of real;
```

but it do not allow `iota` to be passed as a parameter to a function, or allow it to be indexed by a run time expression. Although, in simple cases, `iota` corresponds to a sequence of ascending adjacent addresses, and could thus be mapped to a conventional array, in more complex expressions involving matrix transpositions or reduction operations this is no longer true.

1.4 Opportunities to Parallelise Array Assignments

Figure 1.1 shows the Streaming SIMD (SSE) model of processing supported by Intel and AMD machines. Data is loaded, up to 16 operands at a time into XMM registers. These are then operated on using up to 16 parallel ALUs in register to register mode. The results are then written to memory up to 16 operands at a time. Streaming is achieved by prefetching subsequent operands into cache so that memory fetches occur in parallel with processing. It is clear that this architecture is optimised for vector processing, both because vectors are operated on and because the prefetching mechanism assumes sequential organisation of operands in memory.

Using the SSE model, it is possible to achieve startling speedups on one dimensional array operations.

An additional level of complexity is introduced by multi-cores. Each one of these cores works best when running along a vector, or along the rows of a matrix. The separate cores can however independently work on different rows of a two dimensional array problem. A natural form of parallelism, suggested by the hardware architecture is thus to use SIMD for vector processing along rows, and to spread computations on different rows between cores.

1.5 Transformation for SIMD Parallelism

The intermediate code generated by the parser described in section 1.3 produces scalar serialised intermediate code. This can be run on any processor, but if the processor has vector registers the scalar code will be sub-optimal. After the parser has generated code for an assignment it queries the code generator to see if vector registers are available for the data type produced by the expression. If yes, then the scalar intermediate code is a candidate to be transformed to vectorised intermediate code. We will illustrate the process of

Algorithm 1.1 C code to add two images, along with timing on a 3Ghz Opteron. Compiled with gcc.

```
#define LEN 6400
#define CNT 100000
main()
{
  unsigned char v1[LEN],v2[LEN],v3[LEN];
  int i,j,t;
 /* repeat many times for timing */
  for(i=0;i<CNT;i++)
     for (j=0;j<LEN;j++) [
      t=v2[j]+v1[j];
      if( t>255)t=255;
      v3[j]=t;
     }
}
[wpc@maui tests]$ time C/a.out
real    0m2.854s
user    0m2.813s
sys     0m0.004s
```

Algorithm 1.2 Vector Pascal code equivalent to the C code in Algorithm 1.1. Timing on the same machine as in Algorithm 1.1.

```
program vecadd;
type byte=0..255;
var v1,v2,v3:array[0..6399]of byte;
    i:integer;
begin
     for i:= 1 to 100000 do
         v3:=v1 +: v2;
        { +: is the saturated add operation }
end.
[wpc@maui tests]$ time vecadd
real    0m0.094s
user    0m0.091s
sys     0m0.005s
```

translating code for array expressions with an ultra-simple example which adds two images together. When operating with 8 bit pixels one has the problem that arithmetic operations can wrap round. Thus adding two bright pixels can lead to a result that is dark. So one has to put in guards against this. The problem of adding two arrays of pixels and making sure that we never get any pixels wrapping round can be solved in C a shown in Algorithm 1.1.

The equivalent Vector Pascal code is shown in Algorithm 1.2. The very much improved performance comes from effective utilisation of the available SIMD instructions on the Opteron. The original statement is translated into ILCG as shown:

Pascal

```
v3:=v1 +: v2;
```

ILCG

```
mem(ref uint8 vector ( 6400 ),  +(PmainBase, -25600) ):=
+:(^(mem(ref uint8 vector ( 6400 ),  +(PmainBase, -12800))),
   ^(mem(ref uint8 vector ( 6400 ),  +(PmainBase, -19200)))))
```

Note that all operation are annotated with type information, and all variables are resolved to explicit address calculations in ILCG. Hence it is close to the machine, but it still allows expression of parallel operations. The symbol ^ is the ILCG dereference operation, following the Pascal convention. Efficient translation depends on there being a close match available between the semantics of available machine instructions and the ILCG operations specified by the first level of translation. We specify the machine instruction-set in ILCG. As an example, here are some key instruction specifications taken from the machine specification file `gnuPentium.ilc` which specifies the semantics of the Gnu assembler language for the Pentium:

Saturated Add Bytes

```
instruction pattern PADDUSB(mreg m, mrmaddrmode ma)
 means[(ref uint8 vector(8))m :=
      (uint8 vector(8))+:((uint8 vector(8))^(m),
                          (uint8 vector(8))^(ma))]
 assembles ['paddusb 'ma ',' m];
```

Vector Load and Store

```
instruction pattern MOVQL(maddrmode rm, mreg m)
  means[m := (doubleword)^(rm)]
  assembles['movq ' rm ',' m'\n prefetchnta 128+'rm];
instruction pattern MOVQS(maddrmode rm, mreg m)
  means[(ref doubleword)rm:= ^(m)]
  assembles['movq 'm ','rm];
```

We automatically build a family of optimising code generators that translate from ILCG trees to linear assembly code by pattern matching.

	ILCG Compiler		Java Compiler	
Pentium.ilc	→	Pentium.java	→	Pentium.class
Opteron.ilc	→	Opteron.java	→	Opteron.class

To port to new machines one has to write a machine description of that CPU in ILCG. We currently have production versions for the Intel and AMD machines[16] since the 386 plus Beta versions for the PlayStation 2[6] and PlayStation 3.

Basic array operations are broken down into strides equal to the machine vector length. Then these are then matched to machine instructions following the general approach of Graham [10] to generate code.

ILCG input to Opteron.class

```
mem(ref uint8 vector ( 6400 ),  +(PmainBase, -25600) ):=
+:(^(mem(ref uint8 vector ( 6400 ),  +(PmainBase, -12800))),
   ^(mem(ref uint8 vector ( 6400 ),  +(PmainBase, -19200)))))
```

Assembler output by Opteron.class

```
   leaq         0,%rdx             ; init loop counter
l1:cmpq         $ 6399, %rdx
   jg           l3
   movq         PmainBase-12800(%rdx),%MM4
   prefetchnta 128+PmainBase-12800(%rdx) ; get data 16 iterations
                                         ;  ahead into cache
   paddusb     PmainBase-19200(%rdx),%MM4
   movq        %MM4,PmainBase-25600(%rdx)
   addq        $ 8,%rdx
   jmp         l1
l3:
```

To maintain comparability with the C example we show the code generated at optimisation level 0. The timing shown in Algorithm 1.2 was obtained at optimisation level 0. At higher optimisation levels vectorisation is followed by loop unrolling and other well known techniques. These result considerably more opaque assembler.

1.5.1 Preconditions for SIMD parallelisation

For SIMD parallelisation to be viable a number of conditions must be checked.

1. The expression on the right hand side must not contain any function calls, since functions return scalar results.

2. The operands in the expression must be of the same data length. For instance, one can not efficiently perform SIMD addition between two arrays if one contains 32 floating point numbers and the other contains 64 bit floating point.

3. The least significant indices of all arrays in the expression must either be the same, or must differ by an additive term. Thus the ILCG code equivalent to `a[i,j]+b[i,j]` is SIMD parallelisable as is `a[i,j+10]+b[i,j]` but `a[i,j]+b[j,i]` would not be parallelisable since in the case of `b` we are stepping down columns as `j` is incremented. Since column elements are not stored adjacently in memory, one can not perform a SIMD load of them. Nor would `a[i,j]+b[i,j*3]` be parallelisable, again because the successive elements of `b` being used are not adjacent. On the other hand `a[i,j]+c[i]` for some column vector `c` can be parallelised since it is usually possible to replicate a scalar `c[i]` across all elements of a SIMD register.

4. If the array lengths being operated on are not an integer multiple of the SIMD register lengths, SIMD and scalar code must be combined. Suppose the array length is 10 and the register length is 4. Then SIMD code is used to operate on the first 8 elements of the array, and scalar code for the remaining two elements. For dynamic arrays, this test has to be delayed until run time. In this case the 'remainder code' is performed by an optional scalar loop appended to the SIMD loop.

1.6 Transformation for Multi-core Parallelism

SIMD vectorisation works for one dimensional data, or on the last dimension for arrays stored in row major order, because the hardware has to work on adjacent words. SIMD gives considerable acceleration on image data, and worthwhile accelerations on floating point and integer data. Future machines like the Larrabee will have considerably wider SIMD registers, increasing the benefits of SIMD code. But newer chips also have multiple cores. For these, the recent versions of the Vector Pascal compiler will parallelise across multiple cores if the arrays being worked on are of rank 2. The Pascal source code of the program remains the same independently of whether it is being targeted at a simple sequential machine, a SIMD machine or a multi-core SIMD machine. Targeting is done by flags passed to the compiler:

```
vpc sub2dex -cpu486
```

would compile sub2dex.pas using purely sequential 32 bit instructions.

```
vpc sub2dex -cpuOpteron
```

would compile the same file targeted to a 64 bit Opteron with 1 core using the SIMD instructions in the Opteron instructionset.

```
vpc sub2dex -cpuOpteron -cores2
```

would compile the file to a 64 bit Opteron with 2 cores and SIMD instructions. The compiler is implemented in Java so the selection of code generators and compilation strategies is achieved by dynamically loading appropriate compiler and code generator classes.

Let us now look at the transformations required to achieve this using a trivial rank 2 array example in Algorithm 1.3. The example is not intended to be realistic or useful, only illustrative. We assume that the code has been compiled for a dual core Opteron.

Two threads are dispatched to process the work using a fork - rejoin paradigm. The run time library is built on top of pthreads[20]. For a two core machine, two server threads are initiated at program start up. These wait on a semaphore until post_job passes them the address of a procedure and a stack frame context within which the procedure is to be executed.

The statement x:=y-z is translated into a procedure that can run as a separate task, the ILCG has been simplified for comprehensibility in Algorithm 1.4.

The basic structure of the task procedure is two nested for loops, one for each dimension of the arrays.

The outer loop or row index steps by 2 to ensure that each task will process every 2nd row, starting at the row given by the task number. Thus task 0 will process rows 0,2,4,6,... Task 1 will proocess rows 1,3,5,7,... If there are 4 cores available each task will process every 4th row, etc.

The inner loop, for the column indices, advances by 4 since the Opteron has SIMD registers capable of handling 4 floating point numbers at a time.

Algorithm 1.3 Example of combined mult-core and SIMD parallelism.

```
procedure sub2d;
type range=0..127;
var x,y,z:array[range,range] of real;
begin
        x:=y-z;
end;
```

This translates into ILCG as follows when compiled for a dual core Opteron

```
procedure(sub2d,
 procedure (label12 ... see Algorithm 1.4 )
 post_job[label12,^(%rbp),1];   /* send to core 1 */
 /* Note that %rbp is the Opteron stack frame pointer */
 post_job[label12,^(%rbp),0];   /* send to core 0 */
 wait_on_done[0];
 wait_on_done[1];
)
```

Algorithm 1.4 The function performing nested loops.

```
procedure (label12 /* internal label*/ ,
 for(mem(+(^(%rbp),-24)),^(mem(+(^(%rbp),16))),127  ,   2,
     /*iota [0]             task number        limit  step*/
  var(mem(+(^(%rbp),-32))),/* iota[1] */
  for(mem(+(^(%rbp),-32)), 0    ,127,     4 ,
     /*iota [1]           start limit  step*/
   mem(ref ieee32 vector ( 4 ), /* x[iota[0],iota[1]] */
       +(+(*(^(mem(+(^(%rbp),-24))),512),
         +(*(^(mem(+(^(%rbp),-32))),  4),-131072)),
           ^(mem(+(^(%rbp),-8))))):=
     -(^(mem(ref ieee32 vector ( 4 ),/* y[iota[0],iota[1]] */
       +(+(*(^(mem(+(^(%rbp),-24))),512),
         +(*(^(mem(+(^(%rbp),-32))),  4),-196608)),
           ^(mem(+(^(%rbp),-8))))))),
       ^(mem(ref ieee32 vector ( 4 ),/* z[iota[0],iota[1]] */
         +(+(*(^(mem(+(^(%rbp), -24))),512),
           +(*(^(mem(+(^(%rbp), -32))),  4),-262144)),
             ^(mem(+(^(%rbp),-8)))))))))),

 )
```

1.7 The PURE Function Extension

We define a pure function to be such that does not have any side effects, i.e. it does not update any global, or shared, states outside of its own scope. If the other functions are called from the body of the function, these also have to be pure even if not marked as such. This definition of pure functions is consistent with the definition used in FORTRAN 95. Given the absence of explicit multi-threading constructs in the given language, the above property of pure functions implies thread safety. A function can be labeled as pure by prepending the keyword `pure` in front of every declaration or definition of the function. This means that if, for example, a function is declared as pure in the interface section of the programme, it must be also declared as pure anywhere else in the code, e.g. in the subsequent definition of the function body. Any inconsistency in the declared purity of a function is spotted by the compiler and treated as a syntactic error.

```
pure function next(i : integer): integer;
begin
        next := i+1;
end;
```

Above function *next* operates only on the parameter passed to it, thus it is appropriate to declare it as pure. The keyword *pure* does not bare any semantic value, other than it serves as a hint to the compiler which may then generate multi-threaded code. Multi-threaded code will be generated if -cores*n* flag is passed to the compiler specifying more than one core, and hence the number of threads, available to the programme and if the function is then invoked as part of an assignment statement.

1.8 Task Parallelism and Block Structure

The technique of procedurising code shown in Algorithms 1.3 and 1.4 is well established when parallelising loops in Open-MP [7]. There are two significant differences. First, and least significantly, in Vector Pascal the loop is implicit rather than the explicit loops used in Open-MP. But secondly Open-MP is targeted at C and FORTRAN which are flat languages. Pascal is a block structured language which makes the access to variables by spawned tasks somewhat more complex. Consider Algorithm 1.8 which illustrates the use of nested blocks in Pascal. This has a main program `nestpar` and embedded within that a procedure `emap` which takes a matrix a as a parameter and replaces each $a_{i,j}$ with $e^{scale.a_{i,j}}$ where *scale* is a global variable. The exponential function is approximated by a Taylor series

$$e^x = 1 + \frac{1}{1!}x^1 + \frac{1}{1!}x^1 + \frac{1}{2!}x^2 + \frac{1}{3!}x^3 + ...$$

using the function `Taylor`. The Taylor series is evaluated as
$Taylor \leftarrow \sum (coefs \times x^{\iota_0})$;
where $\iota_0 = 0, 1, 2, 3...$ using the line

```
Taylor:=   \+ (coefs * x pow iota[0]);
```

The coefs vector has been initialised in the main program to contain the inverse factorial series as required.

There are a number of references from inner to outer scopes: `Taylor` uses the vector `coefs`, and `emap` uses the variable `scale`. There are three well known techniques for implementing this in normal procedural code: λlifting, static chaining or display vectors. Since Intel provide direct hardware support for display vectors in the procedure ENTER and LEAVE instructions we have chosen to use displays. Figure 1.2 illustrates how the stack would be organised during execution of Taylor when the program is compiled for a single core machine. Observe how Taylor can access variables in the enclosing stack frames using the display vector. But if the code is to run on a dual core machine there will be not one but three stacks as shown in Figure 1.3: one for the main program and one each for the child tasks. The original function `emap` will have been written to ILCG equivalent to:

```
procedure emap(var a:t);
  var coefs:coef;
  pure function Taylor( x:real):real;
  begin
    Taylor:=   \+ (coefs * x pow iota[0]);
  end;
  procedure dummy(start:int);
  var iota:array[0..1] of integer;
```

Algorithm 1.5 Taylor series example.

```
program nestpar;
type t = array[1..3,1..2] of real;
     coef=array[0..5] of real;
                { tabulate inverse factorials }
const expc:coef=(1,1,1/2,1/6,1/24,1/(5*24));
var scale:real;B:t;
procedure emap(var a:t);
  { for each a[i,j] replace with a[i,j]+exp(scale*a[i,j]) }
  var coefs:coef;
  pure function Taylor( x:real):real;
  begin
    Taylor:=   \+ (coefs * x pow iota[0]);
  end;
begin
   coefs:= expc;
   a := Taylor(a*scale);
end;
begin
     scale:=0.1;
     B:= iota[0]*iota[1];
     write(B);
     emap(B);
     write(B);
end.
```

Output Produced

```
     1.00000     2.00000
     2.00000     4.00000
     3.00000     6.00000

     1.10517     1.22140
     1.22140     1.49182
     1.34986     1.82205
```

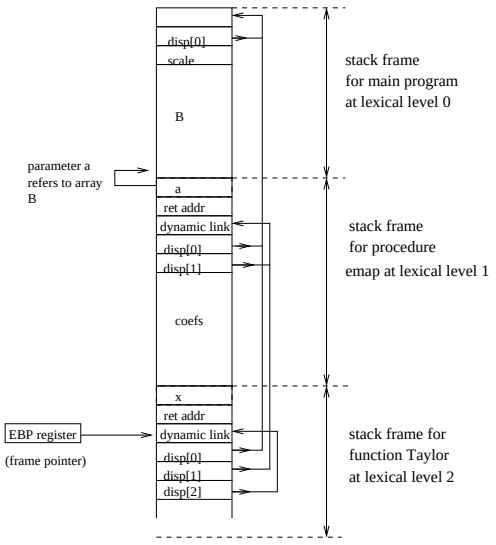

Figure 1.2: Stack for nestpar in single core mode.

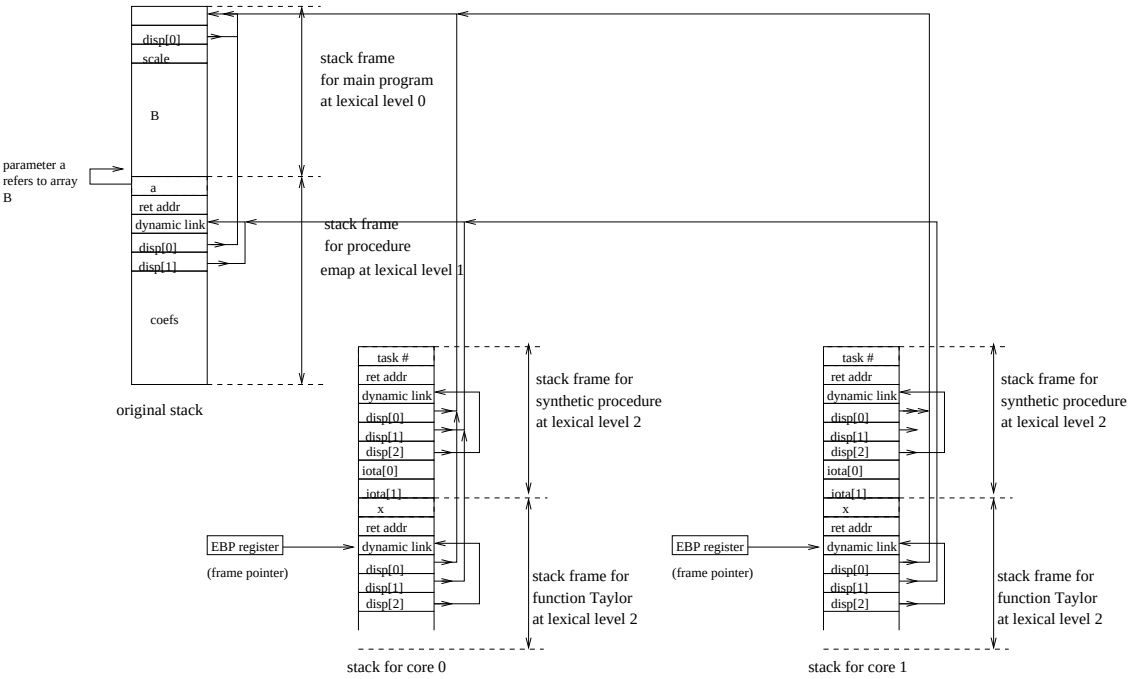

Figure 1.3: The 3 stacks used by nestpar in dual core mode.

```
  begin
    iota[0]:=start;
    while iota[0]<=3 do
    begin
      for iota[1]:=1 to 2 do
       a[iota[0],iota[1]]:=Taylor(a[iota[0],iota[1]]*scale);
      iota[0]:=iota[0]+2;
    end;
  end;
begin
   coefs:= expc;
   post_job(dummy,1);post_job(dummy,0);
   wait_on_done(0);wait_on_done(1);
end;
```

The function `dummy` has to run on a task stack and yet have access to the variable `a` in `emap` and `scale` in the main program, both of which are executing on the main stack. It then has to call `Taylor` in such a way as to ensure that `Taylor` can access the global `scale`. Provided that the displays can be set up as shown in Figure 1.3, this will work, but it is impossible to set up the displays this way when using standard intel call conventions along with the `pthreads` library. Whenever a function is executed within a thread it is allocated a new stack that does not contain display pointers, hence variables from containing scopes cannot be accessed.

In order to support sharing of the global stack amongst multiple tasks, we have implemented an assembly routine *taskexecute*, which corresponds to the following C function signature:

```
void taskexecute(struct threadblock *);
```

As can be seen, the function expects a single parameter which is a pointer to a structure of type *struct threadblock* defined as

```
struct threadblock{

    char * savedframepointer;
    char * savedcodepointer;
    int threadnumber;

}
```

Above, `savedframerpointer` is the pointer to the original stack in which the displays are already setup, `savedcodepointer` is the pointer to the function that is being parallelised, and `threadnumber` is a number in the range0..$n-1$ for a programme running on$n$ cores.. The following assembly code implements `taskexecute` on the Pentium architecture.

Assembly code sequence required to implement the task execute

```
.globl taskexecute
taskexecute:
     # on entry we have a pointer in %esp to the task block
     # this task block has the C definition
     # struct threadblock{
     #         char * savedframepointer;
     #         char * savedcodepointer;
     #         int threadnumber;}
     # the first thing we do is save the framepointer on entry
     push %ebp
     # next get the address of the stored frame pointer in the task block
     mov 8(%esp) , %eax
     #we load the frame pointer into the hardware frame pointer (ebp)
     mov 0(%eax), %ebp
     # get the task number
     push 8(%eax)
     # make the call on the task
     call * 4(%eax)
     # unwind stack pointer
     add $4,%esp
     # restore framepointer we were called with
     pop %ebp
     ret
```

The essence of this form of implementation is that the pthread is setup to execute `taskexecute` which is passed `threadblock` from the calling environment that contains the stack pointer used by the calling environment. `taskexecute` substitutes the stack allocated by the pthread library with the above stack before executing the code sequence contained in the `savedcodepointer`. The effect of substituting the stack pointer is undone once the called code sequence halts to ensure a clean exit of the wrapper.

1.9 Example Programs

1.9.1 Image convolution

The first example we will look at is the use of a seperable convolution kernel to blur an image. Convolution of an image by a matrix of real numbers can be used to smooth or sharpen an image, depending on the matrix used. If A is an output image, K a convolution matrix, then if B is the convolved image

$$B_{y,x} = \sum_i \sum_j A_{y+i,x+j} K_{i,j}$$

A separable convolution kernel is a vector of real numbers that can be applied independently to the rows and columns of an image to provide filtering. It is a specialisation of the more general convolution matrix, but is algorithmically more efficient to implement. We can do a seperable convolution provided that the kernel is formed by the outer product of two vectors **a**,**b**. A symmetric separable convolution can be done if $\mathbf{a} = \mathbf{b}$.

If **k** is a symmetric separable convolution vector, then the corresponding matrix K is such that $K_{i,j} = \mathbf{k}_i \mathbf{k}_j$.

Given a starting image A as a two dimensional array of pixels, and a three element kernel c_1, c_2, c_3, the algorithm first forms a temporary array T whose whose elements are the weighted sum of adjacent rows $T_{y,x} = c_1 A_{y-1,x} + c_2 A_{y,x} + c_3 A_{y+1,x}$. Then in a second phase it sets the original image to be the weighted sum of the columns of the temporary array: $A_{y,x} = c_1 T_{y,x-1} + c_2 T_{y,x} + c_3 Ty, x+1$.

Clearly the outer edges of the image are a special case, since the convolution is defined over the neighbours of the pixel, and the pixels along the boundaries a missing one neighbour. A number of solutions are available for this, but for simplicity we will perform only vertical convolutions on the left and right edges and horizontal convolutions on the top and bottom lines of the image. A Vector Pascal routine to do this is given below. The source has been pretty printed in the latex format that is automatically generated by the compiler is listing enabled. An equivalent sequential C routine is given in Algorithm 1.6.

In comparing the C and Vector Pascal, note two features which give performance advantages to the Vector Pascal form of the algorithm.

1. The support for fixed point 8 bit arithmetic with the pixel type. This allows a higher level of parallelism to be achieved since a P4 or AMD64 can in principle operate on 16 $\times$8 bit numbers with a single instruction. Lacking these types, the C algorithm has to use 32 bit floats. The pixel type automatically uses saturated arithmetic.
2. The data parallel form of expression of the Vector Pascal allows more efficient optimisation of the code.

1.9.1.1 Vector Pascal convolution algorithm

```
type
    plane(rows,cols:integer )= array [0..rows ,0..cols ] of pixel ;
var
    Let T , I ∈^plane;
    Let i ∈ integer;
    begin
```

Allocates a temporary buffer to hold a plane, and 3 temporary buffers to hold the convolution co-ordinates as lines of pixels.

```
    new ( T ,im .maxrow ,im .maxcol );
    new ( I ,3,im .maxcol );
    I↑[0]← c1;
    I↑[1]← c2;
    I↑[2]← c3;
```

Perform convolution on each of the planes of the image. This has to be done with an explicit loop as array maps only works with functions not with procedures.

```
    for i← 0 to im.maxplane do convpar (im_i, I↑, T↑); { see section 1.9.1.2}
```

This sequence frees the temporary buffers used in the convolution process.

```
    dispose ( I );
    dispose ( T );
end ;
```

1.9.1.2 convpar

```
procedure convpar ( var p ,I ,T :plane );
```

This convolves a plane by applying the vertical and horizontal convolutions in turn.

```
var
    Let r, c ∈ integer;
begin
```

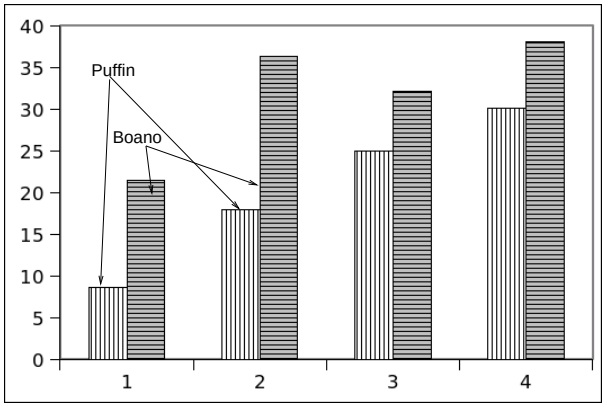

Figure 1.4: Comparison of perfomance gains over the sequential C implementation of the convolution routine on different processors and numbers of cores. The x-axis shows the number of cores for which the program was compiled, the y-axis shows the the relative speedup. Puffin has 4x Quad-Core AMD Opteron 8350 Processor chips running the AMD64 instruction set with a CPU clock of 1Ghz. Boano has 4x Single-Core Intel Xeon CPUs at 2.80GHz running the 32 bit Intel P4 instructions. In each case the performance is normalised so that the speed of the C code counts as 1.

This sequence performs a vertical convolution of the rows of the plane p and places the result in the temporary plane T. It uses the lines of pixels $I[i]$ as convolution weights. Use of lines of pixels rather than the floating point numbers for the kernel weights allows the computation to proceed 8 pixels at a time in parallel. The lines $T_0 \leftarrow p_0$; and $T_r \leftarrow p_r$; deal with the top and bottom rows of the picture which are left unchanged.

```
{$r-}{disable range checks}
```
$r \leftarrow p.rows;$
$T_{1..r-1} \leftarrow p_{0..r-2} \times I_0 + p_{1..r-1} \times I_1 + p_{2..r} \times I_2;$
$T_0 \leftarrow p_0;$
$T_r \leftarrow p_r;$

Now perform a horizontal convolution of the plane T and place the result in p.

$c \leftarrow p.cols;$
$p_{0..r,1..c-1} \leftarrow T_{0..r,0..c-2} \times I_0 + T_{0..r,2..c} \times I_2 + T_{0..r,1..c-1} \times I_1;$
$p_{0..r,0} \leftarrow T_{0..r,0};$
$p_{0..r,c} \leftarrow T_{0..r,c};$
```
{$r+}{enable range checks}
```

end ;

1.9.2 Performance comparisons

We give examples of running the two algorithms on both 32bit and 64bit multi-core machines. Results are summarised in Figure 1.4 and Table 1.1. In all cases the algorithms were compiled at the optimisation level 0 for both compilers.

As would be expected the parallel version of the algorithm significantly out-performs the sequential version. However it is clear that SIMD parallelism provides a more reliable form of acceleration than MIMD. The initial speedup figures using a single core rely entirely on the use of SIMD instructions. SIMD alone gives an acceleration of over $8\times$ on an Opteron and $21\times$on a Xeon. The gain from MIMD is more modest and tails off after markedly after 2 processors on Boano, and after 3 processors on Puffin. This result is consistent with the SIMD code, which operates on wide words, saturating the available memory bandwidth.

Algorithm 1.6 C version of the convolution routine.

```
#include <stdlib.h>
conv(char *im, int planes, int rows,int cols,float c1,float c2,float c3)
/* C version of a convolution routine */
{
 int i,j,p,temp;
 int planestep=rows*cols;
 char * plane, * buffplane;
 char * buff = malloc( rows*planes*cols);
 for (p=0;p<planes;p++){
  plane = &im[p*planestep];
  buffplane= &buff[p*planestep];
  /* convolve horizontally */
  for(i=0;i<rows;i++){
   for(j=1;j<(cols-1);j++) {
    temp= plane[i*cols+j-1]*c1+plane[i*cols+j]*c2+plane[i*cols+j+1]*c3;
    if (temp<0){temp=0;}
    else if (temp>255) { temp=255;} ;
    buffplane[i*cols+j]=temp;
   }
   buffplane[i*cols]=plane[i*cols];
   buffplane[i*cols+cols-1]=plane[i*cols+cols-1];
  }
/* convolve vertically */
  for(j=0;j<cols;j++) {
   for(i=1;i<rows-1;i++){
    temp= buffplane[(i-1)*cols+j]*c1+buffplane[i*cols+j]*c2+buffplane[(1+i)*cols+j]*c3;
    if(temp<0){temp=0;}
    else if (temp>255) { temp=255;} ;
    plane[i*cols+j]=temp;
   }
   plane[j]=buffplane[j];
   plane[(rows-1)*cols+j]=buffplane[ (rows-1)*cols+j];
  }
 }
 free(buff);
}
```

Table 1.1: Data used to produce Figure 1.4. It gives the times in seconds to perform 30 convolutions on a 1024×1024 pixel image with 24 bits per pixel, organised as 3 distinct colour planes, whilst using a 3 element separable kernel.

	gcc	vpc					
cores	1	1	2	3	4	8	16
Machine							
Puffin	5.48	0.81	0.39	0.28	0.23	0.18	0.16
Boano	23.02	1.07	0.63	0.72	0.60		

Table 1.2: Timings in seconds for Mandelbrot algorithms computing the Mandelbrot set at 2048×2048 resolution on the quad Intel Xeon processor Boano. Pascal algorithms as numbered in the chapter. The Fortran95 version is almost the same as Algorithm 1.8 apart from superficial language differences. It is compiled with an research Fortran95 compiler written by Paul Keir at the University of Glasgow. The C version is mandelbrot-1.c written by Michael Ashley, University of New South Wales.

Algorithm	1.7	1.8	1.9	Fortran95	C
Cores					
1	378	10.3	187	10.7	10.2
2		5.6		5.8	
3		4.5	91	4.6	
4		3.7	82	3.9	

1.9.3 Mandelbrot set

It is perhaps not surprising that the image convolution example gives very favourable results for parallel code. This was, after all, what Intel designed the MMX instruction set to do. The next case we look at is computing the Mandelbrot set. Strictly, this is defined as the set of complex numbers M such that for all $c \in M$ the sequence $z_0 = c, z_{n+1} = z_n^2 + c$ does not diverge, i.e, $|z_n| < k$ for some $k > 1$.

To generate a pretty picture like Figure 1.5, one typically plots the complex plane and for each pixel position one computes the number of iterations it takes for the formula to diverge. The divergence time is then used to define the colour or brightness of a pixel. The problem is potentially highly parallel, since the divergence time of each point is independent of all other points. On closer examination though we find that the sort of parallelism is not one readily amenable to SIMD evaluation. To understand this look at Algorithms 1.7 and 1.8.

Algorithm 1.7 is a simple sequential algorithm which is directly based on the definition of the Mandelbrot set. The core of the algorithm is the function *escapebrightness* which for the complex number given by c will compute the number of iterations required for divergence to occur. The picture is then built up by the procedure *buildpic* which uses nested loops to call for the complex number corresponding to each pixel position. This algorithm ran in 378 seconds to compute the set to a resolution of 2048 pixels square.

We then try and speed this up in Algorithm 1.8 by applying two transformations.

1. We replaced the use of complex numbers by reals. This can be expected to accelerate things as complex arithmetic is implemented with calls to a library. This is an inelegant but effective accelerator. Table 1.2 shows an acceleration from 378 seconds to 10.3

2. We replaced the sequential loops in *buildpic* with an implicit map. This will allow parallelism provided that we have qualified *escapebrightness* as a pure function. This allows the algorithm to be accelerated by a further factor of about 3 when when compiled for 4 cores.

Table 1.2 shows very similar timings for C, Vector Pascal and Fortran95 for the single core case. Since the file formats generated for the final image differ between implementations, the differences in timings are not significant. The multi-core acceleration achieved by the Fortran95 and Vector Pascal versions are also essentially the same.

One might hope that it should be possible to gain another factor of 4 in performance by taking advantage of the fact that a P4 class processor can handle 4 floating point numbers at a time. However a SIMD version of the algorithm runs into problems since it must be cast in a form that allows the same operations to be performed simultaneously on a number of data points. But the divergence time will differ between different positions on the complex plane, which makes it difficult to compute several points in lockstep. When one point has already diverged, others have not. Thus we can not have the loop breakout used in the earlier

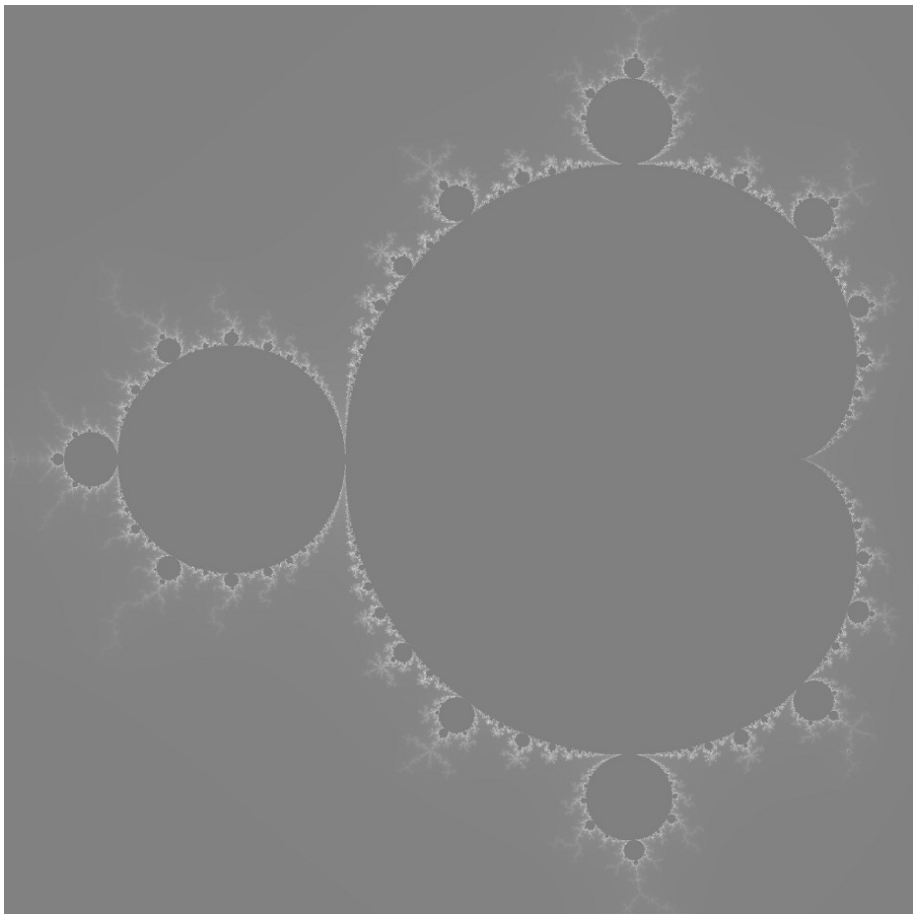

Figure 1.5: The Mandelbrot set image produced by Algorithm 1.8. The original file produced by the algorithm is 4 megabytes in size. Note that since the type Pixel is a signed 8 bit number, 0 translates to mid grey. We have limited ourselves to this rather dull rendering to make the rendering procedure more readily understandable.

Algorithm 1.7 Escape time computed by a sequential ISO-Pascal routine using complex numbers.

```
function escapebrightness ( c : complex ):real ;
label 99;
var
    Let z ∈ complex;
    Let i ∈ integer;
begin
    z← 0.0;
    for i← escapelimit downto 1 do
    begin
        z← z × z + c;
        if escaped (z) then
        begin
            escapebrightness← i × pixelshift;
            goto 99;
        end ;
    end ;
    escapebrightness← 0;
    99:;
end ;


procedure buildpic ( var p :picture );
var
    Let x, y ∈ integer;
begin
    for x← 0 to imlim do
        for y← 0 to imlim do
            p_{y,x}← escapebrightness (cmplx (xorigin + xstep × x, yorigin + ystep × y));
end ;
```

Algorithm 1.8 MIMD Vector Pascal version of Algorithm 1.7 using real arithmetic.

```
pure function escapebrightness (cx, cy : real) : real;
    label 99 ;
var
    Let xx, y, x, x2, y2 ∈ real;
    Let iteration ∈ integer;
begin
    x← 0.0;
    y← 0.0;
    iteration← 1;
    while iteration < escapelimit do
    begin
        xx← (x)^2 - (y)^2 + cx;
        y← 2.0 × x × y + cy;
        x← xx;
        if (((x)^2 + (y)^2) > escapebound) then
            goto 99;
        iteration← iteration + 1;
    end ;
    99: if iteration < escapelimit then escapebrightness← iteration × pixelshift
    else escapebrightness← 0.0;
end ;


procedure buildpic ( var p :picture );
var
    Let x, y ∈ integer;
begin
    p← escapebrightness (xorigin + xstep × ι_1, yorigin + ystep × ι_0);
end ;
```

Algorithm 1.9 Version of the Mandelbrot algorithm that exploits both SIMD and MIMD parallelism. In this, the variables *x, y, cx, cy, times* are all arrays rather than scalars.

```
procedure buildpic ( var p :picture );
var
    Let iteration ∈ integer;

begin
    x← 0.0;
    y← 0.0;
    cx← xorigin + xstep × iota1;
    cy← yorigin + ystep × iota0;
    times← 0;
    for iteration← 1 to escapelimit do
    begin
        xx← x × x - y × y + cx;
        y← 2.0 × x × y + cy;
        x← xx;
        times ← if times =0 then
            if (x × x + y × y > escapebound) then iteration else 0
            else times;

    end ;

    p← times × pixelshift;
end ;
```

algorithms. Algorithm 1.9 shows how the problem can be expressed in SIMD fashion, operating in lockstep on all points in the complex plane. A conditional expression is now used to gather the escape times. These can be executed in SIMD fashion with no branches.

The performance of the SIMD version is frankly disappointing as shown in Table 1.2. It runs in about 1/20th the speed of the MIMD version. This can be explained by the fact that most of the points being examined on the complex plane will diverge rapidly, a great deal of wasted computation is done because the SIMD version can not exploit this. A second factor will be the much poorer cache usage because each statement uses 2D arrays that are too big to fit in the cache.

1.10 Conclusion

Modern desktop computers have the potential to perform highly parallel computations. But realising this potential remains tricky. Array languages like Vector Pascal, SAC or Fortran95 are one promising approach. Highest performance is attained when one can utilise both the SIMD and the MIMD potential of modern chips. This not only requires a compiler that is able to target both forms of parallelism but also requires an appropriate problem domain. Even problems which, on first sight, are highly parallel, may not lend themselves to both sorts of parallelism. But when both SIMD and MIMD can be harnessed, the performance gains are startling.

Bibliography

[1] GE Blelloch. Nesl: A nested data-parallel language. volume CMU-CS-95-170. Carnegie Mellon University, 1995.

[2] W.S. Brainerd, C.H. Goldberg, and J.C. Adams. *Programmer's guide to Fortran 90*. Springer Verlag, 1996.

[3] T. Budd. An apl compiler for a vector processor. *ACM Transactions on Progeramming Languages and Systems*, 6(3), July 1984.

[4] Paul Cockshott. Vector pascal reference manual. *SIGPLAN Not.*, 37(6):59–81, 2002.

[5] G. Conte, S. Tommesani, and F. Zanichelli. The long and winding road to high-performance image processing withMMX/SSE. In *Fifth IEEE International Workshop on Computer Architectures for Machine Perception, 2000. Proceedings*, pages 302–310, 2000.

[6] Peter Cooper. Porting the vector pascal compiler to the playstation 2. Master's thesis, University of Glasgow Dept of Computing Science, http://www.dcs.gla.ac.uk/~wpc/reports/compilers/compilerindex/PS2.pdf, 2005.

[7] L. Dagum, R. Menon, and S.G. Inc. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.

[8] JT Davie and R. Morrison. *Recursive descent compiling*. John Wiley & Sons, 1982.

[9] AK Ewing, H Richardson, AD Simpson, and R Kulkarni. *Writing Data Parallel Programs with High Performance Fortran*. Edinburgh ParallelComputing Centre, 1998.

[10] Susan L. Graham. Table driven code generation. *IEEE Computer*, 13(8):25–37, August 1980.

[11] C. Grelck and S.-B. Scholz. Sac — from high-level programming with arrays to efficient parallel execution. *Parallel Processing Letters*, 13(3):401–412, 2003.

[12] W. Daniel Hillis. *The connection machine*. MIT Press, Cambridge, MA, 1986.

[13] ISO. *Pascal ISO 7185*, 1990.

[14] K. Iverson. *A programming language*. Wiley, New York, 1966.

[15] Kenneth Iverson. *Programming in J*. Iverson Software Inc, Toronto, 1992.

[16] Iain Jackson. Opteron support for vector pascal. Final year thesis, Dept Computing Science, University of Glasgow, 2004.

[17] K. Jensen and N. Wirth. *PASCAL user manual and report: ISO PASCAL standard*. Springer, 1991.

[18] J.A. Kahle, M.N. Day, H.P. Hofstee, C.R. Johns, T.R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589, 2005.

[19] I. MathWorks. *MATLAB: The language of technical computing*. MathWorks, 1984.

[20] B. Nichols and D. Buttlar. *Pthreads programming*. O'Reilly Media, Inc., 1996.

[21] A. Peleg, U. Weiser, I.I.D. Center, and I. Haifa. MMX technology extension to the Intel architecture. *IEEE Micro*, 16(4):42–50, 1996.

[22] R. H. Perrott and A. Zarea-Aliabadi. Supercomputer languages. *ACM Comput. Surv.*, 18(1):5–22, 1986.

[23] S. F. Reddaway. Dap a distributed array processor. *SIGARCH Comput. Archit. News*, 2(4):61–65, 1973.

[24] L Snyder. *A Programmer's Guide to ZPL*. MIT Press, Cambridge, 1999.

[25] M. Tremblay, JM O'Connor, V. Narayanan, L. He, S.M. Inc, and CA Mountain View. VIS speeds new media processing. *IEEE micro*, 16(4):10–20, 1996.