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

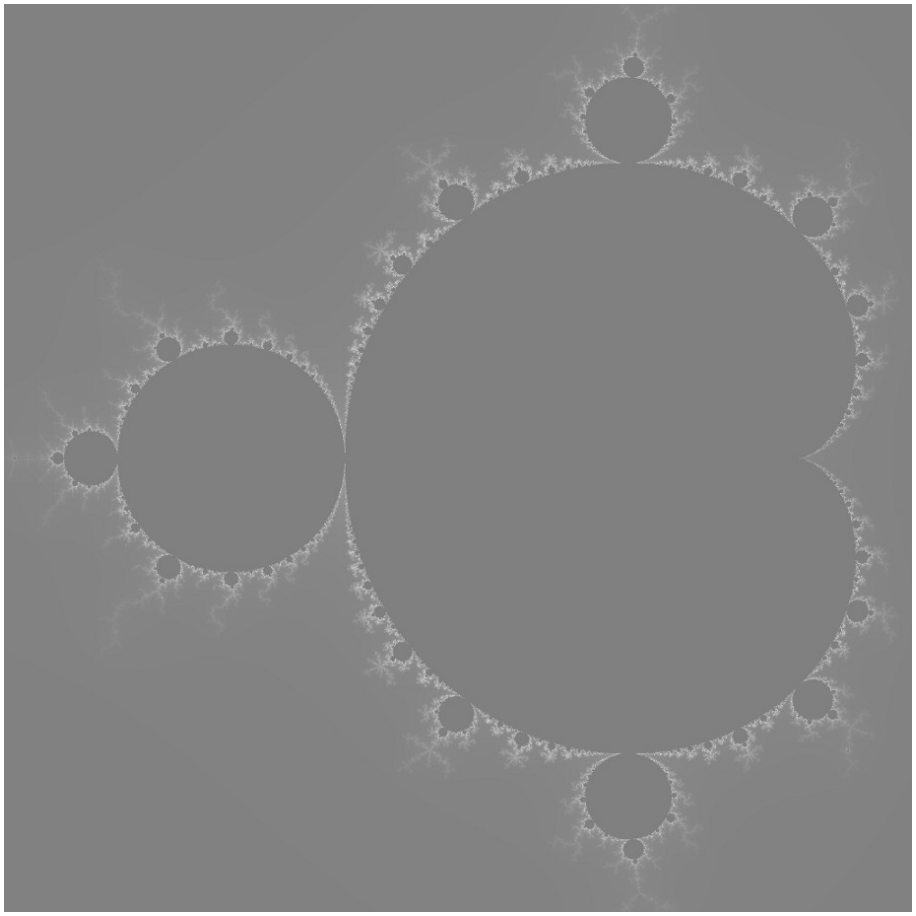

Figure 1.5: The Mandelbrot set image produced by Algorithm 1.8. The original file produced by the algorithm is 4 megabytes in size. Note that since the type Pixel is a signed 8 bit number, 0 translates to mid grey. We have limited ourselves to this rather dull rendering to make the rendering procedure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