

---

# Teaching parallel computing concepts with a desktop computer

Y. F. Fung<sup>a</sup>, M. F. Ercan<sup>b</sup>, Y. S. Chong<sup>a</sup>, T. K. Ho<sup>a</sup>, W. L. Cheung<sup>a</sup> and G. Singh<sup>a</sup>

<sup>a</sup>*Department of Electrical Engineering, The Hong Kong Polytechnic University, Hong Kong*

<sup>b</sup>*School of Electrical and Electronic Engineering, Singapore Polytechnic, Singapore*

*E-mail: eeyffung@polyu.edu.hk*

**Abstract** Parallel computing is currently used in many engineering problems. However, because of limitations in curriculum design, it is not always possible to offer students specific formal teaching in this topic. Furthermore, parallel machines are still too expensive for many institutions. The latest microprocessors, such as Intel's Pentium III and IV, embody single instruction multiple-data (SIMD) type parallel features, which makes them a viable solution for introducing parallel computing concepts to students. Final year projects have been initiated utilizing SSE (streaming SIMD extensions) features and it has been observed that students can easily learn parallel programming concepts after going through some programming exercises. They can now experiment with parallel algorithms on their own PCs at home.

**Keywords** electrical engineering; parallel computing; SIMD paradigm

Parallel programming is a viable method for solving computationally intensive problems in various fields. In electrical engineering, for instance, solving power systems network equations is an area where parallel algorithms are being developed and applied.<sup>1</sup> A popular approach to implementing parallel algorithms is to employ a cluster or a network of PCs. With the advances made in computer hardware and software, it is now quite a simple matter to configure a computer network and program it to solve problems cooperatively. The parallel wavelet transform<sup>2</sup> and software simulation demonstrated by Sena *et al.*<sup>3</sup> are interesting applications that have been implemented on a computer cluster. The common programming paradigm for this type of parallel algorithm is either MPMD (multiple program multiple data) or SPMD (single program multiple data). Communication is based on messages passing between processors. Very often, standard message-passing interfaces, such as MPI<sup>4</sup> or PVM,<sup>5</sup> are used for this purpose. As parallel computers, either in the form of a network of PCs or dedicated machines, become common, so the skills to utilize them fully will become very valuable.

Although parallel computing is a very useful technique, it is often excluded from the traditional engineering curriculum, which therefore hinders the deployment of parallel programs in industry. In the Electrical Engineering Department at Hong Kong Polytechnic University, there are only two computing subjects being taught during the three years of the undergraduate degree program. In their first year, students learn a programming language: currently, the C language is being taught. In the second year, students study a subject to acquire computer hardware and assembly language programming skills. There are, however, some elective subjects including software engineering, computer networks, and industrial computer applications.

On the other hand, in the School of Electrical and Electronic Engineering at Singapore Polytechnic, there are specialized diploma courses in computers and networking, as well as mainstream courses such as electronics and communication engineering. During their studies students are taught the basic theory of computing and given a great amount of practical knowledge. However, parallel computing is not included in the curriculum.

As discussed,<sup>6</sup> it is desirable to provide students with parallel programming skills early in their studies so that these skills can be applied to other fundamental and advanced subjects. However, for an engineering programme it is not possible to implement such a well-defined structure. Considering the significance of parallel computing and its applications in engineering, it is desirable to introduce the concept through some other means. In our case, the final year project is the most suitable method.

As discussed earlier, the current trend in parallel computing is to employ a cluster of workstations and the SPMD programming paradigm. Availability, maintenance and network traffic concerns deterred us from using this solution to train undergraduates. Fortunately, in many modern microprocessors, including the Intel Pentium series, SIMD type parallelism is supported so we can develop a parallel program with a PC even at home. Most importantly, students can learn the concept of parallel computing and implement programs to solve real engineering problems.

In the following section, we introduce the SIMD feature embedded in the Intel microprocessors and the programming model based on it. In later sections, we present our attempts to foster parallel computing concepts at Hong Kong Polytechnic University, where we passed a real engineering problem to the students and asked them to solve it using the Pentium processor's parallel computing features. We also present the work done by another project group at Singapore Polytechnic, where students worked with the same concept and implemented an image processing application. This is then followed by a discussion of the execution of student projects. The final section discusses the merits of this self-learning exercise.

## **SIMD parallelism**

The traditional classification of parallel algorithms was introduced by Flynn and is based on parallelism in instructions and data. SIMD (single instruction multiple data) type parallelism was widely applied in bit-serial massively parallel machines, including the MPP<sup>7</sup> and connection machine.<sup>8</sup> As the name implies, a massively parallel machine consists of many processors and they perform the same operation simultaneously on large quantities of data. Because of advances made in microprocessor fabrication techniques, SIMD is now a common feature included in many recent microprocessors (see, for example, the AMD processor,<sup>9</sup> the SunSparc processor<sup>10</sup> and the PowerPC processor<sup>11</sup>). Early SIMD machines processed a single bit at a time in parallel. However, recent SIMD features embedded in processors allow in-parallel processing of multiple-bit data structures, ranging from integer to double-precision floating-point numbers. With such flexibility, it is possible to implement

algorithms manipulating various data types on these systems. We use Intel microprocessors in our experiments as these processors are commonly used in PCs. Consequently we concentrate on the SIMD feature embedded in Intel processors. On the other hand, AMD microprocessors are also a popular choice in the desktop computer market and the 3D NOW!<sup>9</sup> technology embedded in the AMD processors supports the SIMD mechanism. We can, therefore, also implement the parallel programs with AMD processors.

The SIMD feature that is included in the Intel Pentium processors is called SSE<sup>12,14</sup> and is currently available in the Pentium III and IV classes of microprocessors. This can be regarded as a second generation SIMD feature, its predecessor being the MMX feature.<sup>13</sup> The major difference between MMX and SSE is the data structure that they can process in parallel. MMX registers can operate only on integers, whereas SSE can manipulate both integer and floating-point data types. In many engineering problems, floating-point arithmetic is used, so SSE is a natural choice.

### Streaming SIMD extension (SSE)

SSE registers are 128-bit wide and they can store packed values as characters, integers and floating-points. There are eight SSE registers and they can be directly addressed using their register names.<sup>13,14</sup> Therefore, utilizing these registers in any program becomes a straightforward process with suitable tools. In the case of integers, eight 16-bit integers can be packed into a single 128-bit register and processed in parallel. Similarly, four 32-bit floating-point values can also be fitted into the 128-bit registers and processed in parallel, as shown in Fig. 1. For the Pentium IV microprocessors, the SSE feature is further extended to support the parallel processing of two 64-bit double precision floating-point values.

The first step in applying the SSE feature is packing data into the 128-bit SSE register. When two vectors of four floating-point values are loaded into two SSE registers, as shown in Fig. 1, SIMD operations, such as *add*, *multiply*, etc. can be

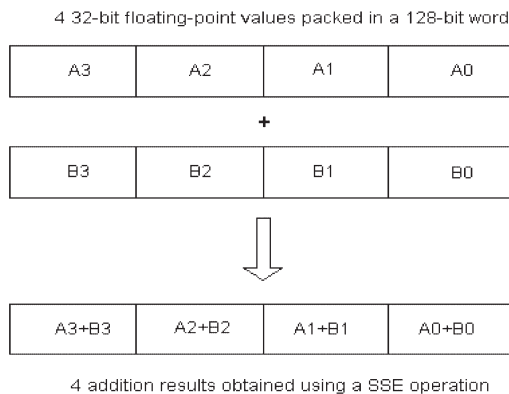


Fig. 1 *Parallelism based on a SSE operation.*

applied to the two vectors in one single operation step, so that theoretically a four times speed-up is possible.

Programming with the SSE may be achieved by two different means. SSE operations can be invoked directly by assembly codes embedded in standard C/C++ programs. Alternatively, development of C/C++ programs without any assembly coding is also possible through the use of some special data types provided by compilers supporting SSE. The new data type designed for the SSE features is F32vec4.<sup>13</sup> The F32vec4 type represents 128-bit storage, usually for holding four items of 32-bit floating-point data. Similarly, there is also type F32vec8, for storing eight 16-bit values. These data types are defined as C++ classes and can therefore be used in a C/C++ program directly. Program codes showing the matrix multiplication of a  $1 \times 4$  matrix times a  $4 \times 1$  vector using the standard C language syntax and using the special data type F32vec4 are shown in Fig. 2. Comparing the two sample functions, we can see that the programming style using SSE is similar to that associated with standard C. Consequently, this style can easily be adopted by someone with some programming experience. In our experiments, we used an Intel compiler since it provides special data types and functions to support manipulation of the SSE mechanism. It is, therefore, easier to learn, as well as implement, programs with SSE. With other compilers, such as the GNU C compiler,<sup>15</sup> the SSE features are invoked through assembly codes in contrast to the special data type mechanism employed in the Intel compiler. Assembly coding is not suitable for our projects as we want to identify a self-learning tool.

In addition to the new data types, operations are provided to load, or to pack, traditional data, such as floating-point values, into the new data structure. In order to load, or to pack, four floating-point values into a F32vec4 data, the function

```

/* a C function to determine the dot product */
float dot (float x[], float y[])
{
    return x[0]*y[0]+x[1]*y[1]+x[2]*y[2]+x[3]*y[3];
}

/* a function to determine the dot product using SSE and data type
F32vec4 */
float dotsimd (float x[], float y[])
{
    F32vec4 vector1 = (F32vec4*) x;
    F32vec4 vector2 = (F32vec4*) y;
    F32vec4 result;
    result = vector1[0] * vector2[0]; // parallel multiplication
    Return result[0]+result[1]+result[2]+result[3];
}

```

Fig. 2 Sample codes for dot product using standard C and SIMD mechanism.

`_mm_load_ps` is called upon. Once data is stored into the 128-bit data structure, functions manipulating the `F32vec4` type data are then applied. This will result in parallel processing of two sets of four floating-point values. On the other hand, the function `_mm_store_ps` is used to convert, or unpack, the data from the `F32vec4` type back to four floating-point values, which are then stored in an ordinary array.

### Solving matrix operations in parallel using SSE

In the above section, we introduced the concept of SSE together with the programming approach. In order to understand the programming mechanism and to examine the advantages of parallel computing, in this section we describe in detail the implementation of a parallel algorithm using the SSE feature. The multiplication of two matrices is selected for this case study. In many engineering problems, data represented in matrix form and matrix multiplication is required. For example, a power network is usually modelled with a matrix of linear equations as shown in equation (1). For large networks, matrix  $A$  and  $B$  will be large, so multiplying such matrices will be time consuming. Hence, the parallel multiplication algorithm developed in this study also has a practical value.

$$AX = B \quad (1)$$

All the matrices in equation (1) are two dimensional. An element in  $B$  will be represented by the symbol  $B_{ij}$ , where  $i$  represents the row, and  $j$  represents the column number. Elements in  $B$  are obtained by multiplying a row in  $A$  with a column in  $X$ , as illustrated in the pseudocode shown in Fig. 3.

The algorithm in Fig. 3 includes three loops, inside which the operations multiplication and addition are performed. In every iteration of loop  $j$ , the multiplication involves different elements, which are independent of each other, so parallelism can

```

For k=0 to k=m-1 /* loop k */
For h=0 to h=m-1 /* loop h */
For j=0 to j=m-1 /* loop j */
Do
    Bhk += Ahj * Xjk
end for
end for
end for
where m represents the size of the matrices
Bhk, Ahj, and Xjk are elements of the matrix B, A and X
respectively.

```

Fig. 3 Pseudocode for the matrix multiplication algorithm.

be exploited.  $A_{hj}$  represents elements along the row  $h$  and they map naturally into the SSE register: see Fig. 1. Similarly, the column elements can also be stored in the SSE register and therefore multiplication of four elements can be executed in a single operation. In Fig. 4, pseudocode of the SSE based parallel algorithm is given. As shown in Fig. 4, by applying the SSE mechanism, the number of multiplications being performed is now reduced by a factor of 4. In our algorithm, by making use of SSE data as a temporary storage, we are also able to reduce the number of additions by a factor of almost 4 (the number of additions performed is equal to  $(m + 1)/4$ ). The performance of the parallel algorithm has been tested using different matrix sizes: the resulting speed-up ratios are illustrated in Table 1.

### Computation speed-up

In order to study computation speed-up we compare two different cases: the traditional approach (that is, without SSE) and a solution obtained using SSE. Students implemented all experiments on a Pentium III system with an operating frequency of 550Mhz.

Table 1 shows the experimental results obtained from these tests. It can be observed that the tests achieved a moderate speed-up for all the experimental matrix sizes, although these results are far from the theoretical speed-up of four. Certainly,

```

F32vec4 temp, par1, par2; /* 128-bit values */
For (k=0; k<m; k++) /* the outer loop */
For (h=0; h<m; h++) {
for (j=0; j<m; j+=4){
/* the inner loop */
store four values from A(h,j) to A(h,j+3) into par1;
store four value from X(j,k) to X(j+3,k) into par2;
temp += par2 * par1; /* carry out parallel operations */      }
Bhk = temp[0]+temp[1]+temp[2]+temp[3];
}
}

```

Fig. 4 Pseudocode for LU decomposition with SSE functions.

TABLE 1 Processing time and speed-up ratios for matrix multiplication

Processing time in ms	Size of matrix				
	100	200	300	400	500
Traditional	28.1	279.4	1129.6	3146	5888
SSE	17.5	197.8	845.2	2286	4226
Speed-up	1.60	1.41	1.33	1.38	1.39

this was a good exercise to help students to become aware of the drawbacks of parallel programming. They recognized the overheads involved in preparing data to be processed in parallel when they started investigating why the performance was not as good as theoretically expected. A further study would be interesting to see how they can improve the above performance by applying various optimization techniques.

### The student project

As discussed in the previous section, the implementation of parallel computing has now become the objective of a final year project. The duration of the project is about ten months, beginning in July and finishing in April the following year. During their summer vacation, between July and August, students attend an industrial training programme so that during this period they can work on their project on a part-time basis only.

Students were first introduced to the concept of SSE, and publications related to the SSE mechanism were provided as references. Students could study the topic at their own pace as most of the materials were available on the Internet, particularly on Intel's web site. We mentioned earlier that computer programming is introduced in the first year of the degree programme. Unfortunately, most students have lost their programming skills by the time they reach their final year. Therefore, students also need time to *re-capture* computer programming concepts before taking up the project. Hence, they spend about four months learning the SSE mechanism and sharpening their programming skills.

After gaining a basic knowledge of the SSE mechanism, they were provided with a sample program based on the LU decomposition algorithm described in Ref. 16. In addition, a program using the traditional method to implement the LU decomposition was also given to them so that they could compare the programs and understand the requirements of writing a SSE-based program. The students were then requested to develop a parallel algorithm for matrix multiplication utilizing the SSE mechanism in order to familiarize themselves with the programming mechanism. Since the SSE feature is available in both Pentium III and Pentium IV microprocessors, any computer system that comes with a suitable Pentium processor can be used for this project. In our case, the students used their own computer and carried out most of the tasks at home using the Intel C++ compiler with its high-level data types (F32vec4) and functions.

By completing the matrix multiplication program, students demonstrated their understanding of both the SIMD and the SSE mechanism. In the following stage, students were asked to identify algorithms that could be optimized by the SSE technique. This was a very important step because it led them to study algorithms related to their studies and to identify locations where they could apply parallel techniques. By doing so, students experienced how parallel computing techniques can be applied to solve *real problems* related to their own work: this will develop awareness of other parallel techniques.

At the end of the project, students were able to develop parallel algorithms individually. The algorithms included matrix inverse, Gauss elimination, and fast

TABLE 2 *Speed-up ratio for different SSE algorithms*

Algorithm	Speed-up
Fast Fourier transform (Forward)	1.4
Gauss elimination (size of matrix 800)	2.2
Gauss elimination (Complex) (size of matrix 800)	1.7
Matrix multiplication (size of matrix 800)	1.75
Complex matrix multiplication (size of matrix 800)	2.6
Matrix inverse (size of matrix 400)	2.05

Fourier transform. The performance of the parallel algorithms was compared with the sequential version and speed-up results are shown in Table 2. Through this project, students experimented with SIMD type parallel programming and recognized its advantages. The project, as a whole, achieved the teaching objectives for programming and parallel computing and also enabled students to apply technical concepts to real engineering problems and to take initiatives in dealing with tasks in hand.

### Solving Hough transform in parallel using SSE

We implemented another case study at Singapore Polytechnic with final year project students as a joint effort. These students had taken modules that provided them with specialization in instrumentation and control areas. Prior to their project they had studied robotics and machine vision subjects during their course. Here, we used the Hough transform as a test bed. This algorithm is widely used for detecting linear line segments or circular objects in an image. Before explaining its parallel implementation, we will briefly explain how the algorithm works.

*Line detection:* Computation of the Hough transform has two phases. The first phase is a voting process where the result is accumulated in a parameter space. In the second phase, the parameter space is elaborated and strong candidates are selected. The voting phase in line detection involves calculating candidate lines, which are represented in terms of parameters by using the following equation:

$$r = x \cos \theta + y \sin \theta \quad (2)$$

The pseudocode for the first phase of the computations is expressed in Fig. 5, where the calculation of parameters and mapping into an accumulator array is the most time-consuming step. Once the accumulator array is filled, line segment extraction is usually done by simply thresholding the accumulator array.

Figure 5 shows the pseudocode for sequential implementation of the Hough transform algorithm. Students exploited SSE registers in various ways to improve processing time. One method was to pack four consecutive  $\cos \theta$  and  $\sin \theta$  values into SSE registers and compute four possible  $r$  values for a given  $x, y$  coordinate. The students referred to this process as angle grouping (AG). They employed `_mm_load_ps1` intrinsic to copy row and column values of each valid edge pixel

```

F32vec C, A1, A2; /* 128-bit values */
For i=0 to ROW do
For j=0 to COL do
If img[i][j] is an edge pixel then begin
For m=0 to MaxTheta do
r=jCos(m)+iSin(m);d=quantize(r);
Accumulator(d,m)+=1;
end for
end if
end for
end for

```

Fig. 5 Pseudocodes for Hough transform.

into all the four words of a pair of `__m128` type data packs. Hence a significant speed-up can be accomplished when calculating parameters as four values of  $\theta$  angle can be calculated simultaneously. In further steps, quantization of four values is also performed simultaneously by utilizing SSE intrinsic.

Another student chose to pack  $x, y$  coordinates of four image pixels into SSE registers and referred to this method as pixel grouping (PG). This time `_mm_load_ps1` intrinsic was used to copy each  $\cos\theta$  and  $\sin\theta$  values into all the four words of a `__m128` data pack. Although the number of packing and unpacking operations is similar, the performance achieved with this method was slightly better.

*Circle detection:* The Hough transform technique can be extended to circles and any other curves that can be represented with parameters. If the point at  $(x, y)$  is positioned on a circle, then the gradient  $(x, y)$  points to the centre of that circle. For a given radius  $d$  the direction of the vector from point  $(x, y)$  can be computed so the coordinates of the centre can be found. Thus, circles are represented by the following equations, where  $a$  and  $b$  indicate the coordinate of the centre point.

$$a = x - d \cos \theta, b = y - d \sin \theta \quad (3)$$

Now, given the gradient angle  $\theta$  at an edge point  $(x, y)$ , we can compute  $\cos\theta$  and  $\sin\theta$ . In general, these quantities are already available as a result of edge detection prior to the Hough transform. Radius  $d$  can be eliminated from the above equations yielding:

$$b = a \tan \theta - x \tan \theta + y \quad (4)$$

Thus, the Hough transform algorithm for circle detection is composed of the following steps:

- 1 Quantize the parameter space for the parameters  $a$  and  $b$ . Initialize and zero the accumulator array  $M(a, b)$ .

- 2 Compute the gradient magnitude and angle at  $(x, y)$ .
- 3 For each edge, increment all points in the accumulator array  $M(a, b)$  along the line using the equation  $b = a \tan \theta - x \tan \theta + y$ .
- 4 Evaluate  $M(a, b)$  where local maximums correspond to centres of circles in the image.

In the above algorithm, calculation of gradient magnitude and angle was performed by the Sobel operator. Students located time-consuming operations and possible data parallelism taking place at step 3 (this is where the accumulator array is filled) and proposed two methods for the solution. The first method they introduced is based on packing edge pixels and gradient angles. Computations are then performed for four of them simultaneously. This method will be called gradients grouping (GG). The second method deals with computation of values of  $a$  and  $b$  in the inner loop. This time coordinates  $x$ ,  $y$  and gradient angle are copied into all four words of `__m128` type data packs. The second method is named centre point grouping (CG). In order to test the performance of their algorithm they used test the images that contain a known percentage of edge pixels.

#### Students' experience with parallel Hough transforms

The main objective of this student project was to cultivate the parallel computing concept and to demonstrate its use in real applications. As mentioned earlier, the duration of the students' project is about nine months, beginning in July and finishing in March of the following year. Students who took up this project were familiar with C programming and they did not need any refreshing of their programming skills. However, they needed assistance in grasping parallel programming and utilizing SSE features. They did not face major difficulties in their first four months while they were building up the basic theory (and, in particular, understanding the Hough transform algorithm). One would expect that as diploma students they would have difficulty in dealing with theory but the result was not as expected. However, this can be explained when one realizes that they were aware of basic image processing and their knowledge was recent. After they demonstrated a successfully working sequential algorithm, we conducted a group discussion where each student decided on one method by which to employ SSE in order to speed up the sequential algorithm. Until the end of their project they performed their work individually. Results collected from their work are combined in Tables 3 and 4.

It is easy to see that the performance improvement obtained by the students working on the circular Hough transform was much better than that obtained from working on line detection. Once the process was explained to them, they were clear about the overhead involved in parallel computing although they were rather disappointed. It was surprising to find that students were competing among themselves to attain the best speed-up, although there was no incentive for such competition in this project. Their attitude can be explained as one of the side-effects of living in a competitive society such as Singapore. At the end of the project, students were confident about parallel programming and its advantages and difficulties. In addition to

TABLE 3 Performance of different approaches to Hough transform

Percentage of edge pixels	Image size												
	256 × 256				640 × 480				1024 × 1024				
	5%	10%	15%		5%	10%	15%		5%	10%	15%		
Execution time (in ms)													
Non_SSE	80	160	240		420	721	1092		1392	2744	3996		
AG	60	130	191		340	591	892		1101	2274	3315		
PG	60	120	180		320	550	841		1042	2133	3054		
Speed-up													
AG	1.25	1.19	1.20		1.19	1.18	1.18		1.21	1.17	1.17		
PG	1.25	1.25	1.25		1.24	1.24	1.23		1.25	1.21	1.24		

TABLE 4 Performance of different approaches to circular Hough transform (in msec)

Percentage of edge pixels		Image size					
		256 × 256			640 × 480		
		10%	15%	20%	10%	15%	20%
Execution time (in ms)	Non_SSE	280	360	440	720	912	1592
	GG	175	220	259	480	507	885
	CG	185	212	254	450	536	838
Speed-up	GG	1.6	1.63	1.7	1.5	1.8	1.8
	CG	1.5	1.7	1.73	1.6	1.7	1.9

the parallel Hough transform, they also prepared a small library of other basic image processing algorithms.

## Conclusions

Parallel computing is used in many engineering problems. Therefore, we should provide suitable training for engineering students so that they are able to exploit the opportunities that parallelism presents. However, it is sometimes not possible to include the topic in the normal curriculum. This paper has described the experience gained when introducing the concept of SIMD type parallel programming in final-year projects. Studies were carried out by students at the Hong Kong Polytechnic University and students at the Singapore Polytechnic. By simply exploiting the SSE features of Intel Pentium III and Pentium IV microprocessors, students can learn both the concept and the programming mechanism of SIMD type parallel software development. This a major advantage since we do not require very expensive parallel computer systems in order to experiment with parallel programming. Students were able to understand the concept of SIMD programming and made use of the technique to develop a number of parallel algorithms that could be applied in tackling problems that emerged in various engineering applications. The hardware requirement for the project is a PC with an Intel microprocessor (Pentium III model or above), and the software is an Intel compiler. Both items are commonly available, so it is possible for students to make use of this technique. Moreover, knowledge of the SIMD mechanism will pave the way for students to learn other types of parallel computing paradigms. Our study has shown that the concept of SIMD mechanism can be acquired by going through programming exercises. Therefore it can be incorporated into the syllabus of other subjects as a self-learning topic or a mini-project.

## Acknowledgement

This work is supported by The Hong Kong Polytechnic University under the grant number A-PD59.

## References

- 1 J. Q. Wu and A. Bose, 'Parallel solution of large sparse matrix equations and parallel power flow', *IEEE Trans. Power Syst.*, **10** (1995), 1343–1349.
- 2 S. Hungenahally and J. You, 'Parallel wavelet transform over distributed computer network for real-time applications', *Real Time Imaging*, **6** (2000), 375–389.
- 3 G. A. Sena, D. Megherbi and G. Isern, 'Implementation of a parallel genetic algorithm on a cluster of workstations: travelling salesman problem, a case study', *Future Generation Comput. Syst.*, **17** (2001), 477–488.
- 4 W. Gropp, E. Lusk and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface* (MIT Press, Cambridge, MA, 1999).
- 5 A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam, *PVM: Parallel Virtual Machine a Users' Guide and Tutorial for Networked Parallel Computing* (MIT Press, Cambridge, MA, 1994).
- 6 L. Pollock and M. Jochen, 'Making parallel programming accessible to inexperienced programmers through cooperative learning', *32<sup>nd</sup> SIGCSE Technical Symposium on Computer Science Education*, 2001, pp. 224–228.
- 7 D. H. Schaefer, J. R. Fischer and K. R. Wallgren, 'Massively parallel processor', *J. Guiding Control Dynam.*, **5** (1982), 313–315.
- 8 J. Helin, 'Performance analysis of the CM-2, a massively parallel SIMD computer', *Concurrency Pract. Experience*, **5** (1993), 71–85.
- 9 3D NOW!™ Technology Manual (AMD Inc., 2000).
- 10 *VIS Instruction Set User Manual* (Sun Microsystems Inc., 2001).
- 11 *Altivec Programming Environments Manual* (Motorola, 2001).
- 12 G. Conte, S. Tommesani and F. Zanichelli, 'The long and winding road to high-performance image processing with MMX/SSE', *Proc. of the Fifth IEEE International Workshop for Computer Architectures for Machine Perception*, 2000, pp. 302–310.
- 13 *The Complete Guide to MMX Technology* (Intel Corporation, McGraw Hill, New York, 1997).
- 14 Streaming SIMD extension application note, no. 931, <http://developer.intel.com>.
- 15 GNU Manual, Free Software Inc., <http://gcc.gnu.org/onlinedocs>.
- 16 Y. F. Fung, M. F. Ercan, T. K. Ho and W. L. Cheung, 'A parallel solution to linear systems', *Microprocess. Microsyst.*, **26** (2002), 39–44.