CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
*Concurrency Computat.: Pract. Exper.* 0000; **00**:1–17
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

# Array Languages and the N-body problem

P. Cockshott, Y. Gdura, P. Keir

SUMMARY

This paper is a description of the contributions to the SICSA multicore challenge on many body planetary simulation made by a compiler group at the University of Glasgow. Our group is part of the Computer Vision and Graphics research group and we have for some years been developing array compilers because we think these are a good tool both for expressing graphics algorithms and for exploiting the parallelism that computer vision applications require.
We shall describe experiments using two languages on two different platforms and we shall compare the performance of these with reference C implementations running on the same platforms. Finally we shall draw conclusions both about the viability of the array language approach as compared to other approaches used in the challenge and also about the strengths and weaknesses of the two, very different, processor architectures we used. Copyright © 0000 John Wiley & Sons, Ltd.

Received ...

## 1. ARRAY LANGUAGES

By the term array language, we mean a programming language that allows array values to be operated on as a whole: passed to functions as a whole, returned from functions as a whole, subjected to arithmetic operations as a whole, and, in the case of an imperative language, assigned as a whole.

Array languages arose out of Iverson's attempt in the 1950s to rationalise mathematical matrix notations into a from suitable for computer interpretation. As a PhD student of Leontief working on the preparation of Input Output matrices for the US economy he first developed Iverson Notation as a means of specifying algorithms that would then be hand coded in assembler. The notation was later developed into the language APL [1].

This language had the key concepts of mapping operators and functions over arrays and introduced generalised reduction and scan functionals. These functional forms: map, scan, and reduce have been influential in the subsequent development of functional parallel programming [2, 3, 4].

Not only were functional languages influenced by APL. It also shaped the development of imperative languages such as ZPL [5], MATLAB [6], versions of Fortran [7, 8, 9] and Pascal [10, 11, 12, 13]. Both of the compilers used in the experiments reported here borrow concepts for expressing data parallelism that have a long history, dating back to APL in the early '60s.

The key concept is that if we define a vector of type $T$ as having type $T[]$. Then if we have a binary operator X:(T,T)$\rightarrow T$ , in languages derived from APL we automatically have an operator X:(T[],T[]) $\rightarrow$ T[]. Thus if $x, y$ are arrays of integers $k = x + y$ is the array of integers where $k_i = x_i + y_i$.

The basic concept is simple, there are complications to do with the semantics of operations between arrays of different lengths and different dimensions, but APL provides a consistent treatment of these. The recent languages built around this model include J, an interpretive

```
var v1, v2, v3: array[0..1023] of integer;
begin
   v1 := v2 + v3 ;
end;
```

Figure 1.1. Operation on Arrays of the Same Rank (Pascal Code)

language [14], and F [15] a modernised Fortran. In principle though any language with array types can be extended in a similar way.

Vector Pascal (VP) and F incorporate Iverson's approach to data parallelism. They aim to provide a notation that allows the natural and elegant expression of data parallel algorithms within a base language that is already well established (Pascal and Fortran) and combine this with modern compilation techniques.

By an elegant algorithm we mean one which is expressed as concisely as possible. Elegance is a goal that one approaches asymptotically, approaching but never attaining [16]. APL and J allow the construction of very elegant programs, but at a cost. An inevitable consequence of elegance is the loss of redundancy. APL programs are as concise, or even more concise than conventional mathematical notation and use a special character-set. This makes them hard for the uninitiated to understand. J attempts to remedy this by restricting itself to the ASCII character-set, but still looks dauntingly unfamiliar to programmers brought up on more conventional languages. Both APL and J are interpretive which makes them ill suited to many of the applications for which SIMD speed is required. The aim of our compilers is to provide the conceptual gains of Iverson's notation within a framework more familiar to imperative programmers.

Iverson's approach to data parallelism was machine independent. It can be implemented using scalar code or using combinations of scalar SIMD and multi-core code.

The following Pascal code segment illustrates how operations can be applied to arrays in the same way that could have been applied to scalars:

In Figure 1.1, the first line declares *v1,v2* and *v3* as arrays of size 1024 of type integer. The third line is an additional operation that involves all the elements of *v1* and *v2*. This statement results in adding each element in vector *v1* to the corresponding element of vector *v2* and storing the sum in the corresponding element in vector *v3*. Figure 1.1 shows a simple example of an array expression that a compiler can safely chop into blocks and then process in parallel.

## 2. PROBLEM TESTED

### *2.1. N-body Benchmark*

The N-body problem is a scientific simulation that involves computing the motion of a number of planets (bodies) under physical forces such as gravity. The gravitational force between each pair of bodies is defined by their position and mass. The algorithm 1 presents the main steps of this problem.

Because N-body simulations require significant computing power, a number of experiments have used this problem for evaluating machine performance. The N-body problem has been also used recently for comparing the performance of modern parallel technology such as GPUs [17] and other parallel architectures such as the new SIMD extension supported by the Sandy Bridge processor [18]. This problem was also selected by the Scottish Informatics and Computer Science Alliance (SICSA) research body as a challenge problem in Phase II of the SICSA Multicore Challenge [19].



DOI: 10.1002/cpe

**Algorithm 1** N-body Problem Pseudocode.

```
For N bodies
Each time step
   For each body B in N
      Compute force on it from each other body
      From these derive partial acceleration
      Sum the partial accelerations
      Compute new velocity of B
   For each body B in N
     Compute new position
```

### *2.2. N-Body Algorithms*

The N-body simulation is required to compute the force between each pair of bodies. In reality, the number N of bodies or particles is often very large, and thus a number of algorithms and methods have been developed to optimise the simulation. The two common algorithms for computing the total force on each body are the All-Pairs method and Barnes-Hut Treecode [17]. The total number of interactions needed to be computed using an ordinary approach, such as All-Pairs algorithm, is $N^2$ while the Barnes-Hut method is an $O(NlogN)$ algorithm [17]. This benchmark drawn from the Great Computer Language Shootout which was originally contributed by Christoph Bauer [19]. The implementation of the problem in VP made explicit use of operations on whole arrays. However, to express the problem in parallel style using arrays, the full $N^2$ forces have to be computed. The VP parallel algorithm is similar to the All-Pair method as it also goes over each body in $N$, say B, and computes the forces of all other bodies $N-1$ on the body B. This additional computation associated with the parallel solution ensures that the calculations are independent and can be safely carried out on multiple processors in parallel. The main part core of the algorithm is a procedure, called advance, whose main loop uses a position matrix to compute a matrix of acceleration components for each body in $N$. These components are summed along the rows to yield a final velocity increment.

## 3. LANGUAGES USED

We used two languages in our team's entry to the challenge: F and VP. These have in common that both are derived from mature sequential programming languages, and both have been extended to use implicit parallelism by the use of APL-inspired constructs.

### *3.1. Fortran and F*

Fortran was originally developed by John Backus and others at IBM in the 1950s. Like C, Fortran is a statically typed, imperative language. Fortran has historically differentiated itself from C by its absent pointer arithmetic; longstanding support for complex numbers; argument passing by reference; and with Fortran 90, first class array types. This allowed whole array operations to be expressed as a single assignment statement. The Fortran language is ISO standardised, and Fortran 2008 was approved in September 2010. Of the mainstream programming languages, Fortran has distinguished itself within the field of computational science, due to its relatively high level, and excellent performance profile.

In Fortran 9x, each array has a statically declared rank. The lower bound, upper bound, and extent of each dimension may then be obtained using built-in functions. In contrast to C, this implies that a single formal array parameter is sufficient for a function definition. In fact a selection of built-in functions are supplied, to query; construct; combine; and, for a fixed operator set, reduce arrays. Furthermore, a scalar function, qualified as *elemental*, will be lifted implicitly to a pure, map-like, operation upon application to conforming array arguments.




DOI: 10.1002/cpe

```
arr : array[0..4, 0..4] of integer ;
vec : array[0..4] of integer ;
begin
   vec := iota(0) ;
   arr := vec * itoa(0) ;
end;
```

Figure 3.1. Operations on Arrays of Different Ranks (Vector Pascal Code) . First line initialises vec to the integers 0..4, the second line forms a matrix such that arr[i,j] = i*j.

```
program showReduction;
type vec = array[1..8] of real;
const v : vec = (0.0001, 0.001, 0.01,0.1,1.0, 10.0, 100.0, 2000.0);
var x,y : real;
begin
  x:= \+ (v+1);
  y:= \* v;
  writeln(x:15:4,y:15:5);
  writeln(v+8:11:4);
end.
```

Figure 3.2. The reduction functional

The "F" programming language is a subset of Fortran 95, designed to provide a lightweight version of Fortran, free of the requirement to support 40 years of language artifacts. The primary motivation of the language design was to create a Fortran-based language for education, however "F" is a perfectly adequate general-purpose language. Furthermore, any Fortran compiler will compile a program conforming to the "F" language standard.

Unlike Fortran, the "F" language is not ISO standardised. The book, "The F Programming Language", by Metcalf and Reid [20], provides the most authoritative language reference. No standard "F" language test suite exists, though a small set of "F" codes have been developed for internal use with E♯. The outputs of these programs are automatically compared against those of the GNU Fortran (GFortran) compiler using Ivano Primi's numdiff program.

### *3.2. Pascal and Vector Pascal*

Like Fortran, Pascal [21] is an old programming language whose relative popularity has declined over the years, but like Fortran and C it lends itself to efficient code generation. VP was first developed by Turner [11] and Formella [12]. It supports whole array operations in a manner that is similar to Fortran 9x. In our tests we used the Glasgow vector Pascal Compiler [13] which supports SIMD instruction sets. The key features already reported in the literature are the existence of a reduction functional borrowed from APL. In APL for example we can write:

```
×/2 3 4
```

and the system replies:

```
24
```

The reduction functional, / in APL, inserts an operator between every element of an array so:

×/2 3 4 → 2 × 3 × 4 → 24

Because the forward slash is already used to indicate division in Pascal we use the \ symbol instead as the reduction functional. Thus given the Pascal example in Figure 3.2 we get the answer:

```
2119.1111        0.00020
8.0001   8.0010   8.0100   8.1000   9.0000   18.0000  108.0000  2008.0000
```



*Prepared using* cpeauth.cls
DOI: 10.1002/cpe

The write expression `v+8:11:4` means compute the sum of the vector `v` with the number `8` and print in fields 11 wide and with 4 digits after the decimal place. Note that the reduction operation `\+ (v+1)` is equivalent to $(v_1 + 1) + (v_2 + 1) + \ldots$. The other feature in VP is the overloading of operators and functions to map arrays of any rank. VP extended the Pascal array assignment operator to handle operations on different rank expressions. The example in Figure 3.1 shows VP operations involving different rank expressions.

The code given in Figure 3.1 results in assigning `iota(0)`, the index vector of `vec` to vector `vec`, and then forming the values:

```
0 0 0 0
0 1 2 3
0 2 4 6
0 3 6 9
```

in array `arr`. To write the same code segment in Pascal requires several more lines of code [13]. VP also allows mixed rank expressions by depending on the VP compiler to automatically generate a loop that can span the ranks of the destination and the source operands, and then the evaluation is carried out based on the number of dimensions of the array on the left-hand side.

These features have more recently been extended by the introduction of the *pure* keyword that marks a function as being side-effect free, and potentially permits mapping of this function over array arguments to be performed in parallel.

The Glasgow Pascal Compiler also incorporates some of the 'Extended Pascal' features described in [22], in particular it allows the declaration of dynamically sized arrays on the heap. For the Intel code generator selected, the Glasgow Pascal Compiler was found to meet the ISO conformance test suite from 1 to 218. This is a relatively high conformance level. In comparison the ultimate release of the commercial Turbo Pascal compiler was found to fail 25 of these conformance tests. The main difference between the Glasgow compiler and the ISO standard is that the Glasgow compiler uses the now more common syntax for file handling introduced by Turbo Pascal rather than the original Jensen and Wirth file handling syntax.

The two languages share a number of common features:

- Strict typing.
- Whole array operations.
- Dynamic arrays.
- Pure functions can be mapped in parallel over arrays.

## 4. HARDWARE USED

### *4.1. The Cell Architecture*

The Cell BE, or Cell, is a heterogeneous multi-core processor. It was designed mainly for multimedia applications [23], though has been used in other areas such as high performance computing. The Cell BE has two quite distinct processors: a 64-bit PowerPC Processor Element (PPE) and eight Synergistic Processor Elements (SPEs) [24]. Both PPE and SPEs support SIMD operations on 128 bit registers, but they have two different instruction sets; one for the PPE and one for the SPEs [23]. The PPE has 3 levels of storage (512 MB RAM, 64KB L1 and 512KB L2 cache) and 32 x 128-bit vector registers. Each SPE has only 256KB Local Store (LS) and 128 x 128-bit registers. the SPE local store is quite distinct from, and does not shadow or cache, the main memory. Each SPE has also a Memory Flow Controller (MFC) to handle communication and data transfers between the PPE and the SPEs. The MFC provides three means of communications: Mailboxes, Signal Notification Registers and Direct Memory Access (DMA) mechanisms to exchange messages to move data between the PPE and the



*Prepared using* cpeauth.cls
DOI: 10.1002/cpe

SPEs. The Mailboxes are shallow hardware FIFOs which accept a sequence of 32 bit words for transmission between processing units. Instructions can write to and read from these FIFOs with low latency.

The Cell processor potentially offers high levels of parallelism, but it is not easy to program due to its heterogeneity of memory structures and instruction sets. Currently, there is a number of parallel programming models for Cell such as OpenMP, Sieve C++ and Offload. These models offer semi-automatic parallelisation environments because the user is required to identify possible parallelism [23, 25, 26]. Recent releases of the GNU tool chain and IBM XL offer compilers for C/C++ and Fortran on both architectures and support OpenMP for Linux platforms.

### *4.2. The Intel Xeon Nehalem*

The hardware for the Intel test used 2 Intel Xeon Nehalem (E5620) each with 4 cores. Each core was hyperthreaded giving a maximum of 16 simultaneous threads supported in hardware. There were 24 GB RAM, and the clockspeed was 2.4 GHz. The machine was running Linux and the C reference tests were compiled with GCC version 4.1.2. The cores contain scalar units and vector units, with the vector units being able to operate on vectors of 128 or 64 bits made up of integer or floating point quantities.

## 5. COMPILERS USED

### *5.1. E♯ compiler*

E♯ is an auto-parallelising compiler targeting the heterogeneous multicore architecture of the STI Cell processor. A source to source compiler, E♯ translates from the "F" subset of Fortran 95 to Offload C++, a C++ language extension [27] utilising pointer locality. Having the requisite support for arrays, the "F" programming language is a suitable language to research the use of array expressions as a mechanism to drive implicit parallelism for scientific computing. The language has a large standard library, and this is made available to both the PPU and the SPU using the GFortran runtime libraries. A C++ array template class has also been developed which provides both an abstraction over the multifarious internal array representations of essentially all Fortran compilers; alongside compatibility with the dual memory address space exposed by Offload C++. The E♯ compiler is written in the pure functional programming language Haskell. Haskell's Parsec [28] parsing library allows the structure of the published "F" grammar to be followed exactly within the Haskell source code, while the Scrap Your Boilerplate [29] package is used to perform the crucial auto-parallelisation transformations upon the abstract syntax trees.

Upon execution by E♯ of an array expression, a team of threads is launched, each assigned a statically allocated and contiguous chunk of the outermost array dimension. The precise number of threads is set on program startup using an environment variable, ESHARP_NUM_THREADS, and may range from 1 to 128. Each individual thread is given the full resources of an SPU, and sits in a notional FIFO queue until an SPU is available. Having six available SPUs, it may be assumed that six threads will maximise resource usage while minimising thread administration costs. Nevertheless, a program with a large working set may need to be split into more than six pieces. For example, an array expression with a 6000KB working set, will exceed the 256KiB local store of an SPU if partitioned across only six threads. With 32 or more threads, such a program should run.

By avoiding language extensions, an "F" program which will run in parallel using E♯, can also run in serial using a command-line switch or using another Fortran compiler. This approach provides code longevity, as well as a familiar deterministic and sequential development paradigm. Unlike many auto-parallelising compilers, the user also has the certainty that *every* array expression will execute in parallel. Consequently, traditional iterative constructs of the "F" language, such as do or while loops remain useful. Such constructs should be used where




DOI: 10.1002/cpe

there is insufficient work to justify the small cost of thread administration and direct memory access (DMA) operations.

### *5.2. Vector Pascal on Intel Architecture*

The Glasgow Vector Pascal compiler is implemented in Java allowing a single executable jar file to be distributed for all platforms. The machine code to be output is selected by a command-line flag. This flag selects one of a library of code generator classes included in the jar file. The code generators themselves are automatically produced by a code generator generator from formal specifications of the target processor instruction-sets [30]. It is also possible, on Intel processors to specify whether the Nasm or Gnu assembler syntax is used. Another command-line flag specifies how many cores are available on the target processor.

The compiler will attempt to parallelise array statements across two dimensions. It will attempt to distribute calculations of the rows of the result across different cores and attempt to parallelise the column results of each row using SIMD instructions. The degree of parallelism achieved depends both on the width of the SIMD registers available, and on the number of cores available.

For the experiments on the Intel processors the target processor selected was the 32-bit P4 instruction set using the gnu assembler syntax. This instruction set includes the SSE and SSE2 instructions along with the legacy integer and floating point opcodes.

### *5.3. Vector Pascal on the Cell Processor*

We used the recent port of the Glasgow Vector Pascal parallelising compiler to the Cell processor (CellVP). The CellVP compiler is made of two components: a PowerPC compiler for the PPE and a virtual machine model to access the SPEs. The compiler works as following:

1. Plants code to create the threads and load the VSM interpreter into the SPE at the beginning of program execution in order to avoid thread launching overhead.
2. Generates PowerPC assembly instructions that correspond to the sequential source code excluding array expressions which need to be checked first.
3. Examines every array expression in the source code and determines if it is parallelisable:
   - the expression must contain no function calls;
   - the array expression must not be part of or include a scatter/gather construct, that is array elements must be selected from consecutive locations;
   - arrays or matrices must not be the target of a transpose operator;
   - the array size must be bigger than the virtual SIMD register size ;

   to be nominated for parallelisation. These are the same rules as the compiler applies to SIMD parallelisation for the Intel Architecture except in that case real rather than virtual SIMD registers are used.
4. If it is to be parallelised the nested loops representing the array expression in the intermediate code tree are rewritten from loops with step 1 to loops with step $n$ where $n$ is the vector length, and the individual arithmetic operations are replaced with overloaded arithmetic operations on vectors of length $n$.
5. Transforms the nominated array expressions into VSM instructions. This is done by feeding the intermediate code tree to a pattern directed code generator whose library of patterns includes both the Power PC instructions and the VSM instructions. When the pattern matcher recognises an operation that corresponds to a VSM instruction, it outputs a sequence of Power PC codes whose effect is to output the VSM binary instruction to the hardware inter-core Mailbox for the SPE that is going to execute that operation.

Note that the CellVP compiler was designed for parallelising large arrays, and therefore it is not expected to perform well on problems of small sizes such as N-body problem of size 1024.




DOI: 10.1002/cpe

```
struct planet {double x, y, z; double vx, vy, vz;double mass; };
void advance(int nbodies, struct planet * bodies, double dt) {
  int i, j;
  for (i = 0; i < nbodies; i++) {
    struct planet * b = &(bodies[i]);
    for (j = i + 1; j < nbodies; j++) {
      struct planet * b2 = &(bodies[j]);
      double dx = b->x - b2->x;
      double dy = b->y - b2->y;
      double dz = b->z - b2->z;
      double distance = sqrt(dx * dx + dy * dy + dz * dz + EPS);
      double mag = dt / (distance * distance * distance);
      b->vx -= dx * b2->mass * mag;
      b->vy -= dy * b2->mass * mag;
      b->vz -= dz * b2->mass * mag;
      b2->vx += dx * b->mass * mag;
      b2->vy += dy * b->mass * mag;
      b2->vz += dz * b->mass * mag;
    }
  }
  for (i = 0; i < nbodies; i++) {
    struct planet * b = &(bodies[i]);
    b->x += dt * b->vx;
    b->y += dt * b->vy;
    b->z += dt * b->vz;
  }
}
```

Figure 6.1. The reference version of the advance fuare providednction.

The better performance can be obtained on arrays of size $4P$ KB where $P$ is the number of SPEs.

## 6. THE N-BODY PROGRAMS

The program starts with an initial position, mass and velocity of a group of particles at a given time. It then uses that data to work out the motions of all particles and to calculate their positions at later times. The calculation is based on the laws of motion and gravitation. For each timestep the algorithm has to integrate the forces on each body due to all the other bodies, compute the resulting acceleration and then calculate the changes in velocity and position. Initial velocities and positions are specified as 3-element vectors for each body, and for each body the mass is given. The metric for performance used was the time taken for a single simulation step which computes forces and updates positions and velocities for all bodies. We present here this problem in three array programming languages, Fortran, VP and E♯ and compare their performance with a C version from the Great Computer Language Shootout at http://shootout.alioth.debian.org originally contributed by Christoph Bauer. The original C version had only 5 bodies. In order to provide a more realistic load the problem was scaled up, to handle an arbitrary number of bodies.




DOI: 10.1002/cpe

### 6.1. E♯ version

From earlier work [31] we were aware that an $O(n^2)$ "all-pairs" n-body simulation on the Cell BE can exhibit good scaling at the expense of wall clock time, and so a tiled decomposition of the problem, inspired by research at Nvidia [32], was developed. Though the complexity of this "all-pairs" algorithm remains $O(n^2)$, the use of computational tiles maximises the ratio of computation to data transfer.

The kernel of our n-body algorithm performs the $O(n^2)$ force calculation in parallel using the SPUs, while the remaining leapfrog-Verlet integration updates the positions and velocities, and is run in serial by the host PPU processor. This balance of loads seems a reasonable choice as, having only linear complexity, the percentage of runtime expended on the integration stage becomes insignificant with larger body counts. A square shaped tile of the pairwise body interactions, maximises the number of calculations that can be performed per body. That is to say, a DMA transfer of $2p$ body positions and masses, will provide $p^2$ components of force for the integrator.

The E♯ compiler parallelises only the outermost of the generated loops. To fully exploit the two-dimensional decomposition already outlined, a "flattened", *one-dimensional*, array is used to feed the requisite driving scalar *elemental* function. User-defined scalar types are required for the input and output elements. For input and output respectively the two types `pchunk2d` and `accel_chunk` are shown below:

```
type, public :: pchunk2d
    type(vec4), pointer, dimension(:) :: ivec4, jvec4
end type pchunk2d
type, public :: accel_chunk
    type(vec3), dimension(CHUNK_SIZE) :: avec3
end type accel_chunk
```

The use of array *pointers* for the `ivec4` and `jvec4` fields of the `pchunk2d` input scalar type avoids the need for an $n^2$ host memory footprint. The kernel may then be concisely expressed as follows, first by multiple calls to the scalar function `advance_p_1d`:

```
do i=1,NSTEPS
   call advance_p_1d(TSTEP,pchunks_1d,pos_masses,vel)
end do
```

Within `advance_p_1d`, a call to the `elemental` function, `calc_accel_p`, shown below, is provided a one-dimensional array of `pchunk2d` values, `pchunks_1d`, as a first argument; a real-valued scalar as the second; and will execute in parallel across all SPUs. The one-dimensional array returned has equal shape to `pchunks_1d`, and `accel_chunk` element type.

```
accels = calc_accel_p(pchunks_1d,tstep)
```

### 6.2. Pascal versions

*6.2.1. Array of structs* The reference C version is shown in Figure 6.1. This is not suitable for parallelisation because of side effects. Each iteration updates the acceleration data for two planets. To parallelise the algorithm side effects were removed. The first Pascal version without side effects, shown in Figure 6.2, used the same array of structs data organisation. Parallelisation is achieved by the italicised line which maps the function `computevelocitychange` over the index set of the column vector of velocity changes `dv`.

*6.2.2. Struct of arrays* An array of structs representation prevents SIMD paralellisation of the code. The alternative struct of array representation shown in Figure 6.3, stores the planet positions and relative displacements as 3×N matrices and represents the distance from the $i$th planet to each other one as an N element vector. Thus the line



*Prepared using* cpeauth.cls
DOI: 10.1002/cpe

```
type planet= record x, y, z, vx, vy, vz, mass:real;  end;
     coord = record pos:array[1..3] of real; end;
  procedure advance(   dt:real);
  var dv:array[1..n,1..1] of coord;
      i,j:integer;
   pure function computevelocitychange(i:integer;dt:real):coord;
   var row:array[1..3]of real;
       pos:coord;
       j:integer;
       dx,dy,dz,distance,mag,t:real;
       b,b2:pplanet;
    begin
     row:=0;      b := planets[i];
     for j :=   1 to n do begin
        b2 := planets[j];
        dx := b^.x - b2^.x;
        dy := b^.y - b2^.y;
        dz := b^.z - b2^.z;
        distance := sqrt(dx * dx + dy * dy + dz * dz);
        mag := dt*b2^.mass / (distance * distance * distance+epsilon);
        row[1] :=row[1]- dx* mag;
        row[2] :=row[2] -dy* mag;
        row[3] :=row[3] -dz* mag;
    end;
    pos.pos:=row;
    computevelocitychange:=pos;
  end;
begin
    dv :=computevelocitychange(iota[0],dt);
    for i:= 1 to N do
      for j:= 1 to 3 do
       v^[j,i]:=v^[j,i]+ dv[i,1].pos[j];
    x^ := x^ + v^ *dt;
 end;
```

Figure 6.2. The array of records Pascal version showing data type and advance function.

```
distance:= sqrt(xp^*xp^+ yp^*yp^+ zp^*zp^)+epsilon;
```

computes the distance of the $i$th planet to all other planets potentially in parallel since `xp, yp, zp` refer to vectors of relative displacements in the x, y, z directions. This can be done to the maximum parallelism supported by the SIMD instructions.

\+ is the summation operator, so line

```
changes.pos:= \+(M^*mag*di);
```

means $changes.pos_i \leftarrow \sum_j M_j \cdot mag_j \cdot di_{i,j}$.

This is a direct borrowing from APL

The struct of arrays approach improves use of the SIMD machine registers, and as an added advantage it may improve cache line utilisation. In the inner loop of Figure 6.2, the velocity vectors of each planet will be loaded into the cache every time its position vector is accessed, but for the algorithm as a whole, the velocity vectors only have to be updated at the end of each phase of the advance function. The array of struct organisation thus wastes cache space during the inner loop, which could impact performance.



DOI: 10.1002/cpe

```
type  vect=array[1..n] of realt; pvect=^vect;
      matr=array[1..3,1..N] of realt; pmatr=^matr;
pure function computevelocitychange(i:integer):coord;
    var row:array[1..3] of real;
        tm:real;
        distance,mag :vect;
        xp,yp,zp:^vect;
        di:matr;
        changes:coord;
    begin
       row:=x^[iota [0],i];
       {Compute the displacement vector between each planet and planet i.}
       di:=  row[iota[0]]- x^;
       {Next compute the euclidean distances }
       xp:=@ di[1,1];yp:=@di[2,1];zp:=@di[3,1];
       distance:= sqrt(xp^*xp^+ yp^*yp^+ zp^*zp^)+epsilon;
       mag:=dt/(distance *distance*distance );
       changes.pos:= \+(M^*mag*di);
       computevelocitychange:=changes;
   end;
```

Figure 6.3. The struct of array version.

## 7. RESULTS

### *7.1. E♯ versus Fortran and C*

E♯ targets the Cell BE of the PlayStation 3 (PS3), which hosts the Fedora Core 7 (Moonshine) Linux distribution; facilitated by the *OtherOS* feature of the PS3's *Game OS*. In this setup, only six of the eight SPEs are available: one SPE runs a mandatory Sony hypervisor; while another is lost to low semiconductor fabrication yields for PS3. The SPUs of the PS3's Cell BE have hardware support only for single-precision floating-point calculations, and hence the N-Body problem is examined in this form only.

Our first performance evaluation of E♯ and the N-Body problem is through the relative speedup obtained by compiling the same F source code using both the E♯ compiler and another Fortran compiler. The fastest alternative Fortran compiler available to us is version 4.1.1 of GFortran, selected in preference to the more recent GFortran 4.6.0 as it provides better performance for the N-Body application; on the Cell BE at least. The GFortran `mcpu` flag was set to `cell`, and the optimisation flag to `O3`.

As described in Section 6.1, a *tiled* decomposition of the N-Body problem maximises the ratio of SPU computation per byte of data transfered from main memory; into SPU local store. A 16x16 tile size was then chosen* in preference to 8x8 as the latter resulted in an SPU memory footprint too large to permit a problem size of 16K. As Table I shows, 16K bodies provides an important data point as we observe a direct correlation between problem size and speedup continue to a peak of 4.91 here.

The *absolute* speedup obtained from the E♯ compiler is now considered. Figure 7.1 shows the performance scaling relationship between the N-Body problem written in F, and compiled with E♯; against the fastest available sequential version, written in C, and compiled using version 4.1.1 of the GCC compiler, again in preference to version 4.6.0. Speedup is observed

*16x16 tiles were thus also used for the serial reference version compiled by GFortran.



*Prepared using* cpeauth.cls
DOI: 10.1002/cpe

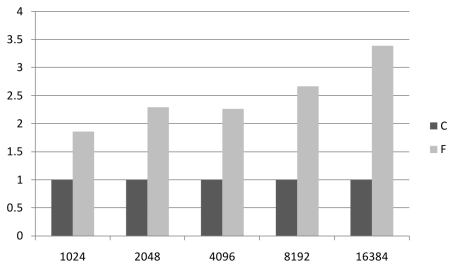



Figure 7.1. Performance scaling wrt. data size of F against C.

Table I. N-Body timings, in seconds per timestep, with E♯ and GFortran on Cell, against relative speedup on E♯, using the same F source code.

| Problem Size | E♯ | GFortran | E♯ Speedup |
|---|---|---|---|
| 1024 | 0.025 | 0.087 | 3.46 |
| 2048 | 0.083 | 0.34 | 4.13 |
| 4096 | 0.34 | 1.37 | 4.00 |
| 8192 | 1.24 | 5.47 | 4.42 |
| 16384 | 4.86 | 23.9 | 4.91 |

Table II. Performance of Vector Pascal and C on Cell. The 1024 row is using 4KB Virtual Register

| N-body Problem Size | Performance (sec per iteration) | | | | |
|---|---|---|---|---|---|
| | Vector Pascal | | | | C |
| | PPE | 1 SPE | 2 SPEs | 4 SPEs | O3 |
| 1024 | 0.38 | 0.10 | 0.065 | 0.048 | 0.045 |
| 4096 | 4.85 | 1.38 | 0.78 | 0.47 | 0.77 |
| 8192 | 20.4 | 5.71 | 3.33 | 2.05 | 3.23 |
| 16384 | 100.2 | 22.3 | 13.2 | 8.09 | 16.5 |

to increase in proportion with data size. Results are surprisingly similar between the 2048 and 4096 body counts; while the final speedup value, at 16386 bodies, is relatively high.

### *7.2. VP cell versus C cell*

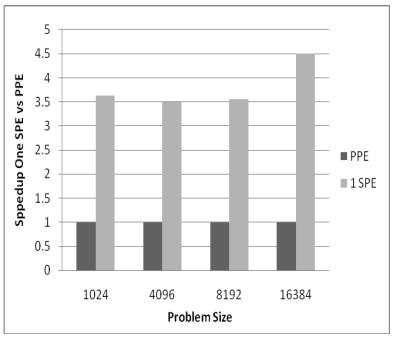
We compare here the performances of VP and C code on the Cell processor. The VP code was compiled by the CellVP compiler which parallelises array expressions on the SPEs automatically. For divisibility reasons we only used 1,2 and 4 SPEs. The C version was compiled and optimized at level 3 using GNU C compiler version 4.6 and run only on the PPE. We shall first look at the VP performance on both the PPE and the SPEs, then expand our discussion to consider the performance of the two languages on the Cell processor.

Table II shows the performance of VP and C code on the Cell processor using different sizes of the n-body problem. Notice here that on 1024 bodies, as commented in Table II, we had to use a virtual SIMD registers (VSR) of length 4KB due to the small problem size but elsewhere we used a VSR of size 16KB to get better performance.

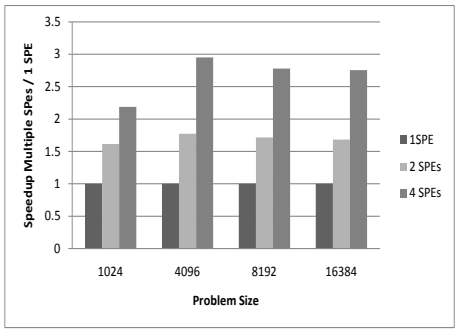
The VP performance on the PPE, as shown in Table II, is much slower than using a single SPE. This is partially due to the fact that the PPE compiler is not vectorised. The speedup gained by using one SPE against using the PPE is shown in Figure 7.2(a). The SPE performs 3.6x as fast as the PPE on problems that are less than 16K and 4.5x as fast as the PPE when the size is 16K. The performance improvement on 16K (16384 bodies) is a result of the PPE relatively poor performance which is due to issues related to caching.



*Prepared using* cpeauth.cls
DOI: 10.1002/cpe

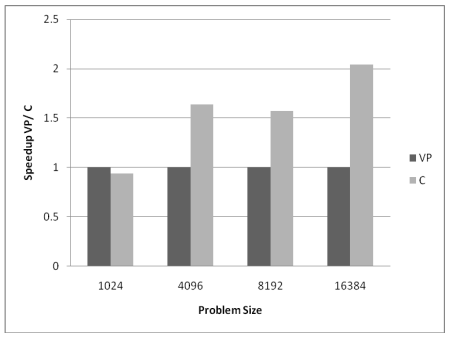



(a) Performance of 1 SPE vs PPE, normalised to PPE.

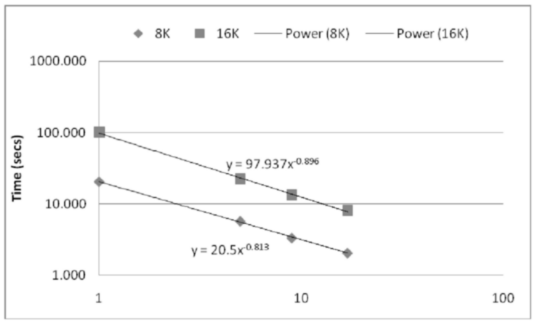



(b) SPEs Scalability

Figure 7.2.




(a) VP vs C on Cell normalised to C.




(b) Performance as function of degree of FPU Parallelism

Figure 7.3.

The scalability achieved by running the VP code on the SPEs is shown in Figure 7.2(b). This figure shows a near-linear speedup but not fully scalable.

Figure (7.3)(a) shows a comparison between the VP code performance using 4 SPEs against the C optimized code running on the PPE. The effect of the problem size can be clearly seen again on 1024 bodies in which the VP code performed relatively slower than the C code. On bigger problem sizes, however, such as 4096 bodies and up, the VP performance was improving as the number of cores increases. Though the VP performance on 4096 bodies is slightly better than the achieved performance on 8192 bodies, VP was roughly 1.5x as fast as C. On the 16384 bodies, VP was 2.1x as fast as C.

Figure (7.3)(b) shows the consistent behavior of the CellVP compiler on large arrays. The diagram is a log log plot of parallelism against time in seconds. Parallelism is measured in terms of the number of effective floating point channels with the PPE counting as 1 and each additional SPE as 4. The regression lines are the best fitting power law regressions. Note that for large numbers of bodies the power law function is almost linear with performance being $\approx p^{0.9}$ for number of processing channels $p$.

### *7.3. VP versus C Xeon and Sandbridge*

The C and Pascal versions were both compiled at optimisation level 3. The Pascal versions were then compiled for between 1 and 16 cores. For each version of the programme timings were averaged over 50 iterations per run, and each compiled version was run 16 times to allow for variations in the background loading of the test machine.



*Prepared using* cpeauth.cls
DOI: 10.1002/cpe

Table III. C versus Pascal on Cell, times for each call of the `advance` function, 1024 planets. GFLOPS calculates on the basis that each call of the C code requires 15728640 floating point operations and each call of the Pascal code 20971520 floating point operations.

| Algorithm | cores used | time per `advance` in ms | GFLOPS |
|---|---|---|---|
| Vector Pascal (PPE) | 1 | 381 | 0.055 |
| Vector Pascal (SPE) | 1 | 105 | 0.119 |
| Vector Pascal (SPEs) | 4 | 48 | 0.436 |
| C (PPE, O3) | 1 | 45 | 0.349 |

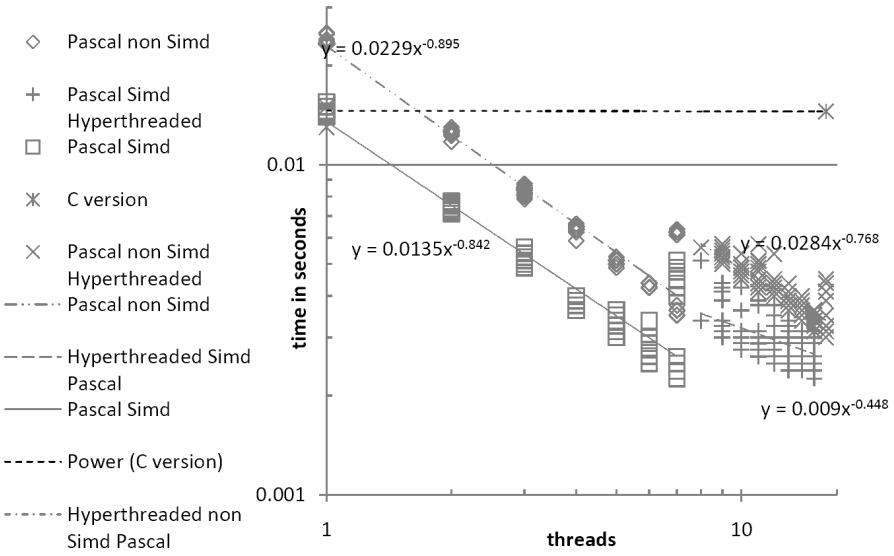



Figure 7.4. Log Log Graph showing Xeon performance for Pascal and C. Note the discontinuity between the threading and hyperthreading domains.

Table IV shows the performance of the Pascal versions compared to the reference C algorithm. On a single processor the C code is faster than either Pascal version. This is in part due to the fact that the paralellisable code performs more operations per call. Normalising by the number of operations in the last column we see that the Gflop performance of the two Pascal versions straddles that of the C.

Compiled for 7 cores, Figure 6.3 performs 6.1 times as fast as the C version or Figure 6.3 on one core. This approximates to a linear speedup. Average processor throughput was 9.13 Gflops under these conditions. Figure 7.4 plots the performance of the different versions as scatters with superimposed power law regressions. Performance increases almost linearly with threads until hyperthreading sets in at 8 threads. Up until this point performance scales roughly as $x^{0.9}$ where $x$ is the number of threads. Hyperthreading imposes an immediate penalty and the subsequent power law shows performance increasing roughly as $x^{3/4}$ with threads used in the non SIMD case and by about $x^{1/2}$ for the SIMD case.

### *7.4. CellVP versus E♯*

From the perspective of the user, there are two important differences between the CellVP and E♯ compilers; both of which are relevant to performance:

- CellVP requires that number of SPUs employed is a power of two: 1, 2, or 4. E♯ is therefore able to exploit an additional 50% of the capability provided by the SPUs.



DOI: 10.1002/cpe

Table IV. C versus Pascal on Xeon, times for each call of the `advance` function, 1024 planets. Gigaflops calculated on the basis that each call of the C code requires 15728640 floating point operations and each call of the Pascal code 20971520 floating point operations.

| Algorithm | threads used | time per `advance` in ms | Gflops |
|---|---|---|---|
| 6.1 | 1 | 14 | 1.12 |
| 6.2 | 1 | 23 | 0.91 |
| 6.3 | 1 | 14.1 | 1.48 |
| 6.2 | 16 | 3.37 | 6.22 |
| 6.3 | 16 | 1.75 | 11.98 |

Table V. Comparison with other approaches at the workshop, all times on 8 core Xeons. The C++ Threading Building Blocks (TBB) example is taken from the run xeon32-O3-sse-v-sq+tbb-bodies20 in Adam Sampson's results. Results ordered by overall performance. The Pascal timing shown in this table was obtained after the workshop by switching on thread affinity, which had not been done in the original tests.

| Language | threads | time per advance | | clock GHz |
|---|---|---|---|---|
| Vector Pascal | 16 | 1.75 | ms | 2.4 |
| C++ Threading Building Blocks | 12 | 2.05 | ms | 2.27 |
| Go | 16 | 8 | ms | 2.4 |
| C sequential | 1 | 14 | ms | 2.4 |
| Eden | 8 | 16.6 | ms | 2.5 |
| C♯ | 12 | 18.2 | ms | 2.33 |

- The E♯ compiler is unable to *stage* the loading of each working set into SPU local stores. Consequently, only CellVP can run the N-Body problem on less than 6 SPUs.

Nevertheless, when these compiler implementation differences are factored out, performance results per SPU are remarkably similar.

### *7.5. Comparision with other languages*

At the workshop on the second phase of the SICSA challenge implementations were presented for 8 processor/language combinations. Of these, 6 were run, using different languages on very similar 8 core Xeon machines allowing meaningful comparison. The fastest implementation reported was one in C++ using TBB for multi-core parallelism and SIMD intrinsics for vector parallelism. This achieved peak performance of 2.05 ms per call of advance when using 12 threads. In comparison the VP implementation achieved its best performance of 2.25 ms when using 7 threads. In the range 1 to 7 threads the Pascal version is faster than the C++ version, but after 7 threads, in the hyperthreading range, performance slows and then improves only gradually for the Pascal (Figure 7.4), whereas for the TBB version whilst there is a continuous, albeit slow, increase in performance as we go through the hyperthreading range, leading to a faster maximum performance. It appears that Threaded Building Blocks make better use of hyperthreading than the pthreads parallelism that the Pascal compiler targets.

There is a large gap between the C++ and Pascal versions and all others. 1. B11 Computer Science and Informatics 3*: Classical Econophysics Remove Add/Edit Info 2. B11 Computer Science and Informatics 3*: Physical constraints on hypercomputation Also selected by: Dr Lewis Mackenzie Remove Add/Edit Info 3. B11 Computer Science and Informatics 3*: Computation and its Limits Remove Add/Edit Info 4. B11 Computer Science and Informatics Computers and economic democracy Remove Add/Edit Info

Eden and C♯ failed, even at maximal parallelism, to outperform sequential C.




DOI: 10.1002/cpe

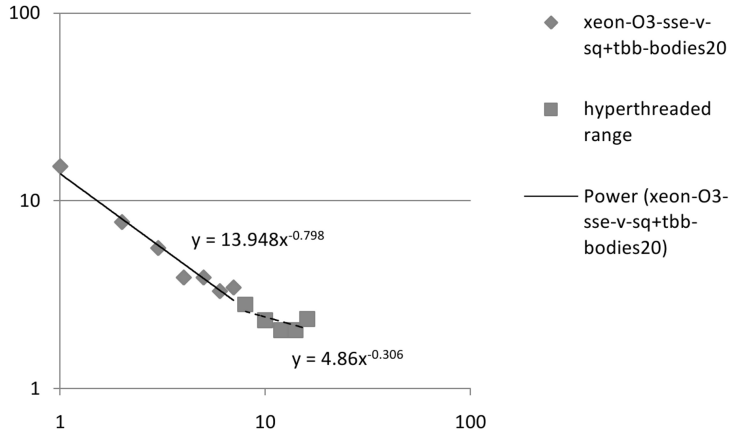



Figure 7.5. Log Log graph for Xeon performance of the C++ Threading Building Blocks code.

### *7.6. Verbosity*

The C reference version, Figure 6.1, contains 21 executable statements, the SIMD Pascal version, ( Figure 6.3 plus its containing function) contains 14 executable statements, so the parallel code is slightly more concise than the sequential code. On the other hand the C contains only 1 line of data declarations, with other variables being introduced via initialising assignments. Pascal does not allow this and contains 8 lines variable declarations. The total length of the two algorithms including data declarations is thus the same.

In comparison to the C++ with TBB and SIMD intrinsics, the array compilers allowed a considerable saving in code. To use the SSE instruction set, the C++ version had to add over 80 lines of class definition for a short vector class with SIMD arithmetic.

## 8. CONCLUSIONS

In this experiment the array language implementations show a number of strengths.

- The array languages delegate the mechanics of parallelism entirely to the compiler. Provided that the array statements are potentially parallelisable, the compiler will perform the parallelisation automatically. The only other example which allowed this in the multi-core challenge was the Eden system.
- The up until the hyper-threading region performance achieved was better than the other approaches. In the hyperthreading region, an implementation based on TBB was slightly better.
- The array language compilers allowed the same source codes to be compiled to a single core, a multi-core homogeneous or a multi-core hetrogeneous computer. In contrast, the low level hand tuning involved in the use of for example Intel TBB, or C++ SIMD intrinsics, precludes portability to non Intel architectures.
- The array algorithms are reasonably concise.

As against these advantages, there are some drawbacks.

- It takes some experience or a certain shift in programmer perspective to express calculations in terms of mapping operations rather than explicit loops.
- Some thought has to be given to the choice of appropriate array structures over which the maps are to be performed. This may involve for example transposing arrays from a



DOI: 10.1002/cpe

row to a column arrangement to accelerate data access, or defining appropriate elemental data types for the arrays.

It is arguable however, that the added attention that a coder must devote to data structuring is no more onerous than the explicit attention that they give to process structure in some alternative approaches.

References


1. Iverson K. *A programming language*. Wiley: New York, 1966.
2. Blelloch G. *Nesl: A Nested Data-Parallel Language*, vol. CMU-CS-95-170. Carnegie Mellon University, 1995.
3. Grelck C, Scholz SB. Sac — from high-level programming with arrays to efficient parallel execution. *Parallel Processing Letters* 2003; **13**(3):401–412.
4. Scholz SB. —efficient support for high-level array operations in a functional setting. *Journal of Functional Programming* 2003; **13**(6):1005–1059.
5. Snyder L. *A Programmer's Guide to ZPL*. MIT Press: Cambridge, 1999.
6. MathWorks I. *MATLAB: The language of technical computing*. MathWorks, 1984.
7. Brainerd W, Goldberg C, Adams J. *Programmer's guide to Fortran 90*. Springer Verlag, 1996.
8. Perrott RH, Zarea-Aliabadi A. Supercomputer languages. *ACM Comput. Surv.* 1986; **18**(1):5–22, doi: http://doi.acm.org/10.1145/6462.6463.
9. Ewing A, Richardson H, Simpson A, Kulkarni R. *Writing Data Parallel Programs with High Performance Fortran*. Edinburgh ParallelComputing Centre, 1998.
10. Perrott R, Crookes D, Milligan P, Purdy W. A compiler for an array and vector processing language. *Software Engineering, IEEE Transactions on* 1985; **SE-11**(5):471–478.
11. Turner T. Vector Pascal: a computer programming language for the FPS-164 array processor. *Technical Report*, Iowa State Univ. of Science and Technology, Ames (USA) 1987.
12. Formella A, Obe A, Paul W, Rauber T, Schmidt D. The SPARK 2.0 system-a special purpose vector processor with a VectorPASCAL compiler. *System Sciences, 1992. Proceedings of the Twenty-Fifth Hawaii International Conference on*, vol. 1, IEEE, 1992; 547–558.
13. Cockshott P, Michaelson G. Orthogonal parallel processing in Vector Pascal. *Computer Languages, Systems & Structures* 2006; **32**(1):2–41.
14. Iverson K. *Programming in J*. Iverson Software Inc, Toronto 1992.
15. Metcalf M, Reid J. *The F Programming Language*. Oxford Univesity Press, 1996.
16. Chaitin G. *Information, Randomness and Incompleteness*. World Scientific, 1987.
17. Playne D, Johnson M, Hawick K. Benchmarking gpu devices with n-body simulations. *Proc. 2009 International Conference on Computer Design (CDES 09) July, Las Vegas, USA., no. CSTN-077*, 2009.
18. Tanikawa A, Yoshikawa K, Okamoto T, Nitadori K. N-body simulation for astronomical collisional systems with a new simd instruction set extension to the x86 architecture, advanced vector extensions. *Arxiv preprint arXiv:1104.2700* 2011; .
19. Challenge phaseii. Http://www.macs.hw.ac.uk/sicsawiki/index.php/Challenge-PhaseII, 2011.
20. Metcalf M, Reid J. *The F programming language*. Oxford University Press, Inc.: New York, NY, USA, 1996.
21. Jensen K, Wirth N. *PASCAL user manual and report: ISO PASCAL standard*. Springer, 1991.
22. Hetherington T. An introduction to the extended pascal language. *ACM SIGPLAN Notices* 1993; **28**(11):42–51.
23. Koranne S. *Practical Computing on the Cell Broadband Engine*. Springer, 2009.
24. Arevalo Ae. *Programming the Cell Broadband Engine Architecture*. International Technical Support Organization, 2008.
25. Marowka A. Performance of openmp benchmarks on multicore processors. *Architectures for Parallel Processing (ICA3PP'08)* 2008; :208–219.
26. Donaldson A, Riley C, Lokhmotov A, Cook A. Auto-parallelisation of sieve c++ programs. *Euro-Par Workshops 2007. LNCS, vol. 4854* 2007; :18–27.
27. Cooper P, Dolinsky U, Donaldson AF, Richards A, Riley C, Russell G. Offload - automating code migration to heterogeneous multicore systems. *Proceedings of the 5th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC'10)*, *Lecture Notes in Computer Science*, vol. 5952, Springer, 2010; 337–352.
28. Leijen D, Meijer E. Parsec: Direct style monadic parser combinators for the real world. *Technical Report* 2001.
29. Lämmel R, Jones SP. Scrap your boilerplate: A practical design pattern for generic progra mming. *Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)*, ACM Press, 2003; 26–37.
30. Cockshott P, Renfrew K. *SIMD programming for Windows and Linux*. Springer, 2004.
31. Donaldson AF, Keir P, Lokhmotov A. Compile-time and run-time issues in an auto-parallelisation system for the Cell BE processor. *Proceedings of the 2nd EuroPar Workshop on Highly Parallel Processing on a Chip (HPPC'08)*, *Lecture Notes in Computer Science*, vol. 5415, Springer, 2008; 163–173.
32. Lars Nyland MH, Prins J. Fast n-body simulation with cuda. *GPU Gems 3*, Nguyen H (ed.). Addison-Wesley Professional, 2007; 677–694.




DOI: 10.1002/cpe