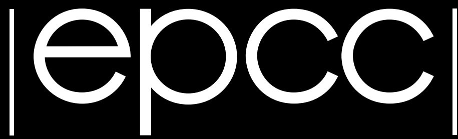

OpenMP

A Parallel Programming Model for Shared Memory Architectures

Paul Graham

Edinburgh Parallel Computing Centre

The University of Edinburgh

March 1999

Version 1.1

Available from: http://www.epcc.ed.ac.uk/epcc-tec/documents/

|epcc|

Table of Contents

1 Introduction **1**

2 Shared memory platforms **2**

3 Why OpenMP? **4**

3.1 Background and Support for OpenMP 5

3.2 A simple parallelisation example 5

3.3 Special features of OpenMP 6

4 The OpenMP Specification **10**

4.1 Parallelisation directives 10

4.2 Parallel region construct 10

4.3 Data environment constructs 11

4.4 Work-sharing constructs 15

4.5 Synchronisation constructs 17

4.6 Conditional compilation 19

5 Library Routines and Environment Variables **20**

5.1 Execution Environment Routines. 20

5.2 Lock Routines 21

5.3 Environment Variables. 22

6 Performance and Scalability **24**

6.1 The Game of Life. 24

6.2 Performance. 26

7 References **31**

8 Acknowledgements **33**

A MPI version of the Game of Life **35**

B HPF version of the Game of Life **39**

1 Introduction

Parallel programming on shared memory machines has always been an important area in high performance computing (HPC). However, the utilisation of such platforms has never been straightforward for the programmer. The Message Passing Interface (MPI) commonly used on massively parallel distributed memory architectures offers good scalability and portability, but is non-trivial to implement with codes originally written for serial machines. It also fails to take advantage of the architecture of shared memory platforms. The data parallel extension to Fortran90, High Performance Fortran (HPF) offers easier implementation, but lacks the efficiency and functionality of MPI. Over the years there have been several other products from both hardware and software vendors which have offered scalability and performance on a particular platform, but the issue of portability has always been raised when using these products.

OpenMP is the proposed industry standard Application Program Interface (API) for shared memory programming. It is based on a combination of compiler directives, library routines and environment variables that can be used to specify shared memory parallelism in Fortran and C/C++ programs. OpenMP is intended to provide a model for parallel programming that is portable across shared memory architectures from different vendors. In relation to other parallel programming techniques it lies between HPF and MPI in that it has the ease of use of HPF, in the form of compiler directives, combined with the functionality of MPI.

This document provides a background and introduction to OpenMP and its implementation. Section 2 looks at shared memory platforms. Section 3 describes what OpenMP offers along with the vendors that are supporting its implementation. Sections 4 and 5 provide information on how to program using OpenMP along with some examples.

2 Shared memory platforms

The shared memory architecture consists of a number of processors which each have access to a global memory store via some interconnect or bus. The key feature is the use of a single address space across the whole memory system, so that all the processors have the same view of memory. The processors communicate with one another by one processor writing data into a location in memory and another processor reading the data. With this type of communications the time to access any piece of data is the same, as all of the communication goes through the bus.

The advantage of this type of architecture is that it is easy to program as there are no explicit communications between processors, with the communications being handled via the global memory store. Access to this memory store can be controlled using techniques developed from multi-tasking computers, *e.g.*, semaphores.

However, the shared memory architecture does not scale well. The main problem occurs when a number of processors attempt to access the global memory store at the same time, leading to a bottleneck. One method of avoiding this is memory access conflict is by dividing the memory into multiple memory modules, each connected to the processors via a high performance switching network. However, this approach tends to shift the bottleneck to the communications network.

As well as stand-alone machines, the shared memory architecture is also found as part of some massively parallel processor (MPP) machines such as the SGI Power Challenge or the Digital AlphaCluster. These machines use a hybrid architecture of multiple shared memory nodes connected by a high performance communication system. In order to achieve high performance on these systems, both shared memory and distributed memory programming techniques are essential.

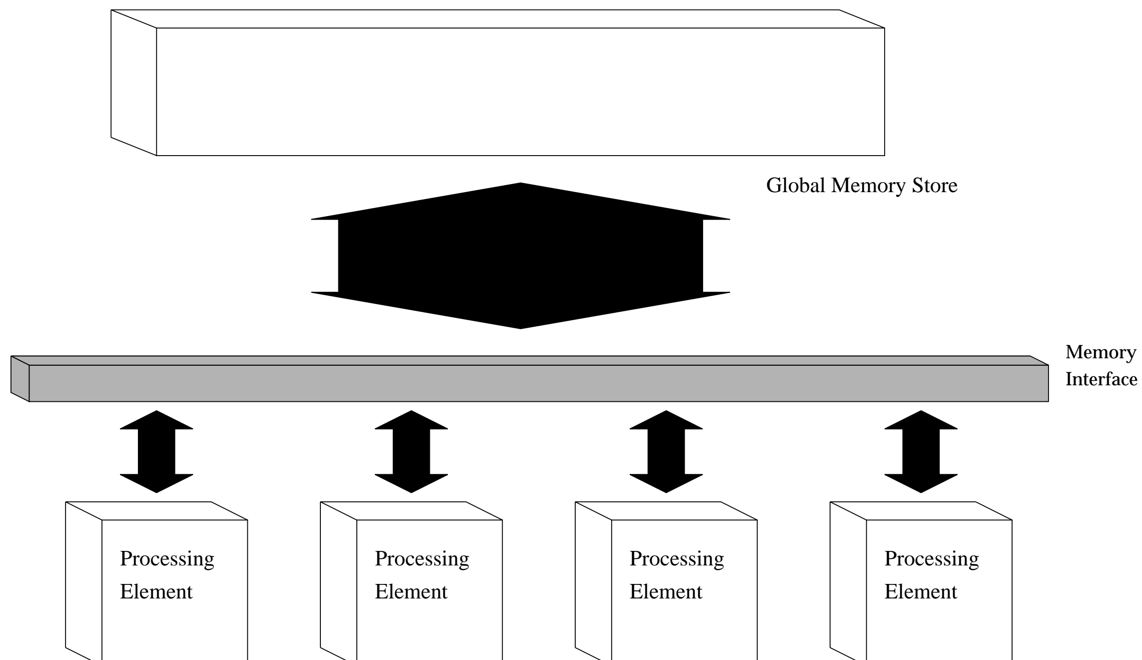

Figure 1: Schematic of a shared memory architecture

Examples of shared memory machines include:

- SGI Origin2000: This is effectively a hybrid shared and distributed memory architecture. The memory is physically distributed across nodes, with two processors located at each node having equal access to their local memory. It is a shared memory platform in the sense that all other nodes have similar access to this memory but are physically more distant, but it can still be programmed as a symmetric multi-processor (SMP) machine. Also as the number of nodes accessing this memory increases a bottleneck situation will arise, but this is a limitation one would expect. (**http://www.cray.com/products/systems/origin2000**)
- Sun HPC servers, such as the Enterprise 3000 or the Enterprise 10000 (Starfire). These are true shared memory boxes, with the E3000 containing 1 to 6 processors and the E10000 between 4 and 64 processors (**http://www.sun.com/servers**).
- HP Exemplar series, such as the S-Class (4 to 16 processors) and the X-Class (up to 64 processors). These use a memory crossbar for data traffic to/from the I/O system and processors. (**http://www.hp.com/pressrel/sep96/30sep96a.htm**)
- DEC Ultimate Workstation. Consists of only 2 processors but each processor is powerful (533 MHz). (**http://www.workstation.digital.com/products/uwseries/uwproduct.html**).

3 Why OpenMP?

The main technique used to parallelise code in OpenMP are the compiler directives. The directives are added to the source code as an indicator to the compiler of the presence of a region to be executed in parallel, along with some instruction on how that region is to be parallelised. Advantages of this technique include relative ease of use and portability between serial and multi-processor platforms, as for a serial compiler the directives are ignored as comments. Amongst others, SGI, Cray and Sun have all generated their own set of compiler directives independently. They are all similar in style and functionality but of course are not trivially portable across platforms from different vendors.

The closest approximation to a standard shared memory programming model is from the X3H5 group [1]. Although it has never become an ANSI standard it is widely used as the basis for the compiler directives for shared memory programming. However, X3H5 has limitations which make it suitable only for loop level parallelism which limits the scalability of any applications which use it (see [2] for more detail on X3H5 and other compiler directives and their advantages/limitations).

The MPI standard is widely used and allows source code portability as well as efficient implementation across a range of architectures. However, the message passing technique requires that data structures in the program are explicitly partitioned. This implies that the whole application must be written with MPI in mind, thus increasing development time and making the parallelisation of old serial codes more difficult. HPF offers portability along with ease of use and reasonable efficiency, but is limited in its functionality. The release of the HPF-2 [3] standard should confront some of the functionality issues when a fully compliant compiler is available, but OpenMP has a distinct and important advantage in that there is a C/C++ implementation of OpenMP.

Pthreads is a low-end shared memory programming model, but it is not targeted at the technical/HPC end-user. There is little Fortran support and under C it is difficult to use for scientific applications as it is aimed more at task parallelism with minimum support for data parallelism.

Thus there exists a demand for a solution which has the following properties:

- portable
- scalable
- efficient
- high level
- supports data parallelism
- relatively easy to implement (for both old codes and new developments)
- wide range of functionality

which is where OpenMP comes in. For a more detailed discussion of the advantages of OpenMP see [4].

3.1 Background and Support for OpenMP

OpenMP was announced to the computing industry as a:

> *"portable, scalable model that gives shared-memory programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer"*

OpenMP Press Release, October 28, 1997

It was jointly defined by:

- Digital Equipment Corp. (`http://www.digital.com/info/hpc/`)
- IBM (`http://www.ibm.com/`)
- Intel Corporation (`http://www.intel.com/`)
- Kuck & Associates Inc. (`http://www.kai.com/`)
- Silicon Graphics Inc. (`http://www.sgi.com/Technology/OpenMP/`)

Various other hardware and software vendors are endorsing the OpenMP API, as well as key application developers. For a full up to date list, as well as the press release and standard for OpenMP, see `http://www.openmp.org/`. An important recent development is Sun Microsystems announcement in August 1998 of their endorsement of OpenMP and a new seat on the OpenMP Architecture Board.

Examples of compilers supporting OpenMP:

- Absoft Pro FortranMP 6.0 (`http://www.absoft.com/pro.win.html`)
- IBM XL Fortran (`http://www.software.ibm.com/ad/fortran/xlfortran/`)
- KAI KAP/Pro Toolset (`http://www.kai.com/kpts/_index.html`)

Most of the other major vendors are due to release their OpenMP compliant compilers towards the latter end of 1998.

At the time this document was written EPCC had access to the KAI Guide f77 and C/C++ compiler for OpenMP on a Sun E3000. The functionality of the compiler in relation to the standard is extensive, the main omission being support for the lock routines (section 5.2). Code was also tested on an SGI Origin2000 at Manchester Computing Centre (`http://www.mcc.ac.uk/hpc/origin/`) running the latest version of the MIPSpro Fortran compiler which supports OpenMP (version 7.2.1).

3.2 A brief note on terminology

Throughout this document we shall be looking at both the Fortran (f77) and C/C++ implementations of OpenMP concurrently, as they are very similar. However, the main difference between them in terms of syntax is that in the f77 implementation, a parallel region must be closed with another directive, whereas in the C/C++ the extent of the region is explicitly defined using curly brackets, for example:

```
f77:    !$OMP PARALLEL
f77:          call work(x,y)
f77:    !$OMP END PARALLEL


C/C++:  #pragma omp parallel
C/C++:    {
C/C++:      work(x,y);
C/C++:    }
```

Therefore when an **`END`** directive is mentioned in the text it will be referring to the f77 implementation only. Also, when directives are being discussed the f77 version (e.g. **`DO`**) will be given first, followed by the C/C++ version (e.g. **`for`**), i.e. **`DO/for`**.

3.3 A simple parallelisation example

Before we delve into the details of the specification let us consider a simple example. Figures 2 and 3 show the f77 and C/C++ versions respectively of a routine for finding the sum of an array. More information on the concepts introduced here can be found in section 4.

The parallel region starts with the **`!$OMP PARALLEL/#pragma omp parallel`** directive and ends at **`!$OMP END PARALLEL`** for the f77 version, at the end of the structured block for the C version. This region is to be executed by multiple threads in parallel.

Now say this subroutine was executed on a shared memory machine with four threads available to it. The parallelisation directive **`DO/for`** means that the following do loop is to be executed in parallel, that is, the loop over **`i`**. The **`SHARED/shared`** clause on the directive means that all the threads executing in parallel have access to the same storage area for the variables **`a`** and **`n`**. The **`PRIVATE/private`** clause means that each thread has its own private copy of the named variable, **`i`**. Thus for this case each of the four threads will perform calculations on a quarter of the iteration space, for example thread 1 has the range of **`i`** between **`1`** and **`n/4`**, thread 2 has **`n/4+1`** to **`n/2`** and so on. Now if the directive was left at that, at the end of the parallel loop (**`!$OMP END DO`**) the variable **`sum`** would be undefined, as each thread would have its own local total for its section of the iteration space rather than the global total which is what we are looking for. This is where the **`REDUCTION`** clause comes in. It causes a local copy of the shared variable **`sum`** to be created for each thread as if the **`PRIVATE(sum)`** clause had been stated. Then at the end of the **`DO/for`** loop the original shared variable **`sum`** is updated from the private copies using the operator specified, so in this case each local sum is added together to create the global **`sum`**.

```
      subroutine array_sum(a,n,sum)
      implicit none
      integer i, n
      real a(n), sum
      sum = 0.0
!$OMP PARALLEL
!$OMP DO SHARED(a,n) PRIVATE(i) REDUCTION(+:sum)
      do i = 1, n
        sum = sum + a(i)
      enddo
!$OMP END DO
!$OMP END PARALLEL
      return
      end
```

Figure 2: a simple example: f77 version

```
float array_sum(float a[], int n){
  int i;
  float sum = 0.0;
  #pragma omp parallel
    {
    #pragma omp for shared(a,n) private(i) reduction(+:sum)
      {
        for (i=0; i<n; i++)
          sum += a[i];
      }
    }
  return(sum);
}
```

Figure 3: a simple example: C version

Now this example is written using several lines of directives, however the defaults and shortcuts of OpenMP mean that the following single line would have exactly the same effect:

```
f77:     !$OMP PARALLEL DO REDUCTION (+:sum)
C/C++:   #pragma omp parallel do reduction (+:sum)
```

This demonstrates the conciseness and ease of use of OpenMP for generating parallel versions of serial code at the do loop level.

3.4 Special features of OpenMP

This section highlights some of the features included in OpenMP which were not present in some of the previous shared memory programming models.

3.4.1 Orphaning

Orphan directives are directives encountered outside the lexical but within the dynamic extent of the parallel region:

Table 1: lexical and dynamic extents

lexical extent	Statements lexically contained within a structured block
dynamic extent	All statements in the lexical extent, plus any statement inside a function that is executed as a result of the execution of statements within the lexical extent.

To demonstrate this figures 3 and 4 show an alternative way of writing the code from the previous example (for brevity the routine `array_init` is not included here).

```
program main
real a(100),sum
```

```
!$OMP PARALLEL
      call array_init(a,100)
      call array_sum(a,100,sum)
!$OMP END PARALLEL

      write(*,*) 'Array sum =',sum
      end

      subroutine array_sum(a,n,sum)
      integer i, n
      real a(n), sum

      sum=0.0
!$OMP DO REDUCTION(+:sum)
      do i = 1, n
        sum = sum + a(i)
      enddo
!$OMP END DO

      return
      end
```

Figure 4: f77 example to illustrate orphaning

```
main(){
  void array_init(float[], int);
  float sum, array_sum(float[], int);
  #pragma omp parallel
    {
      array_init(a, 100);
      sum = array_sum(a, 100);
    }
  printf("Array sum = %f\n",sum);
}

float array_sum(float a[], int n){
  int i;
  float sum = 0.0;
```

```
    #pragma omp for reduction (+:sum)
      {
        for (i=0; i<n; i++) sum += a[i];
      }
    return(sum);
  }
```

Figure 5: C example to illustrate orphaning

This may seem like a trivial difference to the previous example, but X3H5 has no equivalent. Orphaning is powerful in the sense that it greatly simplifies the implementation of coarse grain parallel algorithms. It gives one the ability to specify control or synchronization from *anywhere* inside the parallel region, not just within the lexically contained portion. Under X3H5 all the control and synchronization must be lexically visible within the parallel construct. In this example that implies that the routine would have had to be written explicitly into the main program, which is highly restrictive to the programmer for anything other than trivial coarse grain parallelism. OpenMP provides the functionality of orphaning by specifying binding rules for all directives and allowing them to be encountered dynamically within the call chain of the parallel region.

3.4.2 Nested Parallelism

Under X3H5 nested parallelism is allowed, however under some other directive based parallelisation techniques (such as SGI's `DOACROSS` model) it is not. OpenMP allows nested parallelism. If a `PARALLEL/parallel` directive is encountered dynamically within another `PARALLEL/parallel` directive a new team (that is, a group of threads executing the same section of code in parallel) is established. This team is composed of only the current thread unless nested parallelism is enabled by using the `OMP_SET_NESTED/omp_set_nested` subroutine/function or the `OMP_NESTED` environment variable, in which case the number of threads in the team is implementation dependent. Figure 4 shows an example of its use.

```
      program main
      real x(100),y(100)
c Enable nested parallelism
      call OMP_SET_NESTED(.true.)

c Start parallel region.
!$OMP PARALLEL

c Start parallel sections, one for x and one for y.
!$OMP SECTIONS
c Section A. Perform work on x. Start a new team to do this work.
!$OMP SECTION
!$OMP PARALLEL
!$OMP DO
      do i = 1, 100
```

```
          call do_work_on_x(x,i,100)
        enddo
!$OMP END PARALLEL

c Section B. Perform work on y. Start a new team to do this work.
!$OMP SECTION
!$OMP PARALLEL
!$OMP DO
        do i = 1, 100
          call do_work_on_y(y,i,100)
        enddo
!$OMP END PARALLEL
!$OMP END SECTIONS

c Do work involving both x and y
!$OMP DO
        do i = 1, 100
          x(i) = x(i)*y(i)
        enddo

!$OMP END PARALLEL
end
```

Figure 6: Example demonstrating nested parallelism

Say for this example that the implementation has eight threads available to it. At the start of the parallel region, parallel sections are started, of which there are two, one for work on variable `x` and one for `y`. Thus two threads are utilised for the parallel sections. However, within each section a new parallel region is started. As `OMP_SET_NESTED` has been set to be true , then these nested parallel regions perform their operations using an implementation dependent number of threads. Finally the sections region is over and the last do loop involving both `x` and `y` is executed using *all* eight threads.

N.B. One should be aware that the number of threads actually used within a nested parallel region is implementation dependent. This is because the `OMP_SET_NUM_THREADS` library routine used to determine the number of threads only has effect if called from a serial portion of the code. In the case described above one might imagine that the sections will execute using four threads each, but there is ***no*** guarantee that this is the case.

3.4.3 Atomic update

Neither X3H5 nor SGI's `DOACROSS` model support atomic updating. The OpenMP `ATOMIC/atomic` directive ensures serial access to the single assignment statement succeeding it, that is, it prevents the possibility of a memory location be updated simultaneously by different threads. In this sense it is similar to the `CRITICAL/critical` directive (section 4.5.2), but it lacks the functionality as it applies only to the statement immediately following it which must

be of a certain allowed form (section 4.5.4 for more details). However, the lack of functionality is made up for by the fact that it permits optimisation beyond that of the `CRITICAL/critical` directive and should be used over that whenever possible.

3.4.4 Parallel sections

As seen in section 3.4.2 OpenMP has a `SECTIONS/sections` directive which allows parallelisation of non-iterative sections of code. This removes the limitations of only being able to implement loop-level parallelism.

4 The OpenMP Specification

Conceptually OpenMP is similar to X3H5 with additional functionality to support coarse grain parallelism. However, as well as the parallelisation directives there are callable runtime library routines and environment variables. This section deals with the use of the directives for parallelising code, while section 5 defines the library routines and environment variables. The full standard can be found at `http://www.openmp.org`. These two sections are based on material from the *OpenMP Fortran Application Program Interface*, with permission from the OpenMP Architecture Review Board, copyright © 1997-98 OpenMP Architecture Review Board.

4.1 Parallelisation directives

The parallelisation directives added to the source code are based on the following form:

```
sentinel        directive_name [clauses]
```

For f77, the sentinel can take the forms **!$OMP**, **C$OMP**, or ***$OMP**, and usually must appear in column one as a single word. As we have seen already this report uses the **!$OMP** sentinel. Standard Fortran syntax applies to the directive line, that is, the initial directive lines must have a space or zero in column six, and continuation directive lines must have a character other than a space or a zero in column six.

For C/C++, the sentinel is replaced with **#pragma omp** followed by the directive names and clauses. Also a new-line must be used at the end of the directive.

The **directive_name** and **[clauses]** are covered in the sections following.

4.2 Parallel region construct

The **PARALLEL/parallel** and **END PARALLEL** directives define a *parallel region*, a block of code that is to be executed by multiple threads in parallel. This is the fundamental parallel construct in OpenMP that starts parallel execution. The directives take the format:

f77:

```
!$OMP PARALLEL [clause[[,] clause] . . . ]
block
!$OMP END PARALLEL
```

C/C++:

```
#pragma omp parallel [clause[ clause] . . . ] new-line
  structured block
```

where ***clause*** can be one of the following (see section 4.3):

- **PRIVATE / private**
- **SHARED / shared**
- **DEFAULT / default**
- **FIRSTPRIVATE / firstprivate**
- **REDUCTION / reduction**

- **IF** / **if** (*scalar_logical_expression*)
- **COPYIN / copyin**

The ***block*** denotes a structured block of Fortran statements. It is illegal to branch into or out of the block.

If the **IF/if** clause is present the enclosed region is executed in parallel only if the scalar_logical_expression evaluates to .true./non-zero.

When a thread encounters a parallel region, it creates a team of threads, and it becomes the master of the team. The master thread is a member of the team and it has a thread number of 0 within the team. The number of threads in the team is controlled by environment variables and/or library calls (see section 5). The number of physical processors actually hosting the threads at any given time is implementation dependent.

The **END PARALLEL** directive denotes the end of the parallel region. There is an implied barrier at this point and only the master thread continues execution at the end of a parallel region.

4.3 Data environment constructs

4.3.1 PRIVATE

Syntax: **f77: PRIVATE (** *list* **)** **C/C++: private (** *list* **)**

The **PRIVATE/private** clause declares that the variables *list*ed are treated as private to each thread in a team; that is, a separate copy of the variable exists on each process. These copies are no longer storage associated with the original variable, and as such are undefined on entering the parallel construct. Conversely, the corresponding shared variable is undefined on exiting the construct.

Example: A simple parallel loop

f77:

```
!$OMP PARALLEL DO PRIVATE(i) SHARED(xnew, xold, dx, n)
      do i = 2, n
        xnew(i) = (xold(i) - xold(i-1)) / dx
      enddo
!$OMP END PARALLEL DO
```

C/C++:

```
#pragma omp parallel for private(i) shared(xnew, xold, dx, n)
  {
    for (i=1: i<n; i++) xnew[i] = (xold[i]-xold[i-1])/dx;
  }
```

(It is not actually necessary to explicitly declare **i**, **a** and **b**, as the loop iteration variable is **PRIVATE** by default and **a** and **b** are **SHARED** by default. Also the **END PARALLEL DO** directive is optional.)

4.3.2 FIRSTPRIVATE

Syntax: **f77: FIRSTPRIVATE (** *list* **)** **C/C++: firstprivate(** *list* **)**

This clause is similar to the **PRIVATE/private** clause but has the additional functionality that the private copies of the variables are initialised from the original variables existing before the construct.

Example: Not all the values of **a** are initialised in the loop before they are used, so using **FIRSTPRIVATE** for **a** causes the initialization values produced by subroutine **init_a** to be copied into a **PRIVATE** copy of **a** for use in the loops.

```
      integer n
      real a(100), c(n,n)
      call init_a(a, n)
!$OMP PARALLEL DO SHARED(c, n) PRIVATE (i, j)
!$OMP& FIRSTPRIVATE(a)
        do i = 1, n
          do j = 1, i
            a(j) = calc_a(i)
          enddo
          do j = 1, n
            c(i,j) = a(i)**2 + 2.0*a(i)
          enddo
        enddo
!$OMP END PARALLEL DO
```

4.3.3 LASTPRIVATE

Syntax: **f77: LASTPRIVATE (***list***)**   **C/C++: lastprivate (** *list* **)**

As for the **PRIVATE** clause, but causes the thread that executes the sequentially last iteration of a do loop to update the version of the variable it had before the construct.

Example: This example causes the value of **i** at the end of the parallel region to be equal to **n**, as it would have been for the sequential case.

```
#omp pragma parallel for lastprivate(i)
  {
    for (i=0; i<n; i++) a[i]=b[i]+c[i];
  }
```

4.3.4 SHARED

Syntax: **f77: SHARED (***list***)**   **C/C++: shared (** *list* **)**

This clause causes all the variables that appear in the *list* to be shared among all the threads in the team, that is, each thread within the team have access to the same storage area for **SHARED/shared** data.

4.3.5 DEFAULT

Syntax: **f77: DEFAULT (PRIVATE | SHARED | NONE)**

C/C++: default (shared | none)

The **DEFAULT** clause allows the user to determine the attributes for all the variables in a parallel region. Variables in **THREADPRIVATE** common blocks are not affected by this clause.

- **DEFAULT (PRIVATE)** makes all the variables in the parallel region private to a thread

as if each were listed in a **PRIVATE** clause. (Only on available in f77)

- **DEFAULT (SHARED)** makes all the variables in the parallel region shared among the threads of the team, as if each variable were listed explicitly in a **SHARED** clause. This is the default behaviour if there is no explicit **DEFAULT** clause.
- **DEFAULT (NONE)** declares that there is no implicit default as to whether variables are private or shared, so the attributes of each variable in the parallel region must be explicitly declared.

Variables can be exempted from the defined default clause by explicitly declaring them as **PRIVATE** or **SHARED** etc., for example:

```
!$OMP PARALLEL DO DEFAULT(PRIVATE), FIRSTPRIVATE(I), SHARED(X)

#pragma omp parallel for default (shared) firstprivate(i) \
        private(x)  private(r) lastprivate(i)
```

4.3.6 REDUCTION

Syntax: **f77: REDUCTION ({** *operator* | *intrinsic* **}** **:** *list* **)**

C/C++: reduction (*operator* **:** *list* **)**

This clause performs a reduction on the variables that appear in list, with the operator ***operator*** or the intrinsic ***intrinsic***, where ***operator*** (f77) is one of:

+, *****, **-**, **.AND.**, **.OR.**, **.EQV.**, **.NEQV.**

and ***intrinsic*** is one of :

MAX, **MIN**, **IAND**, **IOR**, **IEOR**

and similarly for the comparable operators in C/C++.

A **PRIVATE** temporary variable is created for the reduction variable, and is replaced into the original variable after the end of the construct. This variable is initialised depending on the operator (see the standard for details). For example:

```
!$OMP PARALLEL

!$OMP DO SHARED (a, n) PRIVATE(i) REDUCTION(max : maxa)

      do i = 1, n

        maxa = max ( maxa, a)

      enddo

!$OMP END PARALLEL
```

So at the end of this loop the private values of **maxa** are combined to give a global value.

4.3.7 SCHEDULE

Syntax: **f77: SCHEDULE(** *type* [, *chunk*]**)**   **C/C++: schedule (** *type*[, *chunk*]**)**

The **SCHEDULE** clause specifies how iterations of a **DO** loop are divided among the threads of the team. Table 1 shows the values ***type*** can take. This clause is mainly used for load balancing between threads.

In the absence of the **SCHEDULE** clause the default schedule is implementation dependent.

Table 2: Use of the **SCHEDULE** *clause (lower-case for C/C++)*

type	Effect
STATIC	Iterations are divided into pieces of a size specified by ***chunk*** where ***chunk*** is a scalar integer expression. The pieces are statically assigned to threads in the team in a round-robin fashion in the order of the thread number. When no ***chunk*** is specified, the iterations are divided among threads in contiguous pieces, and one piece is assigned to each thread.
DYNAMIC	As for **STATIC**, except as each thread finishes a piece of iteration space, it dynamically obtains the next set of iterations.
GUIDED	For this option the size of each chunk is reduced in an exponentially decreasing manner with each dispatched piece of the iteration space. The variable ***chunk*** specifies the minimum number of iterations to dispatch each time, except when there are less than ***chunk*** iterations remaining, at which point the rest are dispatched.
RUNTIME	When **SCHEDULE(RUNTIME)** is set scheduling is deferred until run time, when the schedule type and chunk size can be chosen by setting the **OMP_SCHEDULE** environment variable (section 5.3.1). If this variable is not set, the resulting schedule is implementation-dependent. It is illegal to specify a chunk when **SCHEDULE(RUNTIME)** is specified.

4.3.8 THREADPRIVATE

f77:

Syntax:

!$OMP THREADPRIVATE (/*cb*/[,/*cb*/] . . .)

where ***cb*** is the name of the common block to be made private to a thread

This directive makes named common blocks private to a thread but global within the thread. It must appear in the declaration section of the routine after the declaration of the listed common blocks. Each thread gets its own copy of the common block, so data written to the common block by one thread is not directly visible to other threads. During serial portions and **MASTER** sections of the program, accesses are to the master thread's copy of the common block.

On entry to the first parallel region, data in the **THREADPRIVATE** common blocks should be assumed to be undefined unless a **COPYIN** clause (section 4.3.9) is specified on the **PARALLEL** directive.

When a common block that is initialised using **DATA** statements appears in a **THREADPRIVATE** directive, each thread's copy is initialised once prior to its first use. For subsequent parallel regions, the data in the **THREADPRIVATE** common blocks is guaranteed to persist only if the dynamic threads mechanism has been disabled and if the number of threads is the same for all parallel regions.

It is illegal for a **THREADPRIVATE** common block or its constituent variables to appear in any clause other than a **COPYIN** clause. As a result they are not permitted in a **PRIVATE**, **FIRSTPRIVATE**, **LASTPRIVATE**, **SHARED**, or **REDUCTION** clause. They are not affected by the **DEFAULT** clause. See the next section for an example of the use of **THREADPRIVATE**.

C/C++:

Syntax:

```
#pragma omp threadprivate ( list ) new-line
```

This directive makes the file-scope or namespace-scope variables specified in *list* private to a thread but file-scope visible within the thread. Outside of the parallel region references to these variables update the master thread's copy.

After the first parallel region executes, the data in the threadprivate objects is only guaranteed to persist only if the dynamic threads mechanism has been disabled and the number of threads remains unchanged for all parallel regions.

Each variable in a **threadprivate** directive must have a file-scope or namespace-scope declaration that lexically precedes the directive. Also, the directive itself must appear at file-scope or namespace-scope, must appear outside of any definition or declaration and must lexically precede any references to any of the variables in its list.

A **threadprivate** variable must not appear in any clause other than the **copyin**, **schedule** or the **if** clause. As a result, they are not permitted in **private**, **firstprivate**, **lastprivate**, **shared** or **reduction** clauses. The **default** clause has no effect on them.

4.3.9 COPYIN

Syntax: **f77: COPYIN (*list*)** **C/C++: copyin(*list*)**

The **COPYIN** clause applies only to **THREADPRIVATE** common blocks/variables. A **COPYIN** clause on a parallel region specifies that the data in the master thread of the team be copied to the threadprivate copies of the variable for each thread at the beginning of the parallel region. In f77, it is not necessary to specify a whole common block to be copied in as named variables appearing in the **THREADPRIVATE** common block can be specified in *list*.

Example:

```
      common /block/ scratch
      common /coords/ x, y, z
!$OMP THREADPRIVATE (/block/, /coords/)
!$OMP PARALLEL DEFAULT(PRIVATE) COPYIN (/block/, z)
```

In this example the common blocks **block** and **coords** are specified as thread private, but only the **z** variable in coords is specified to be copied in.

4.4 Work-sharing constructs

There is no work distribution in a parallel region until a work-sharing construct is encountered, as up to that point each active thread executes the entire region redundantly. The work-sharing constructs divide the region among the members of the team of threads that encounter it, and must be enclosed dynamically within a parallel region in order for the directive to execute in parallel.

The following restrictions apply to the work-sharing directives :

- Work-sharing constructs and **BARRIER/barrier** directives must be encountered by all threads in a team or by none at all.
- Work-sharing constructs and **BARRIER/barrier** directives must be encountered in the same order by all threads in a team.

4.4.1 DO/for

Syntax :

f77:

```
!$OMP DO [clauses]
```

Fortran do loop

[!$OMP END DO [NOWAIT]]

C/C++:

#pragma omp for [clause[clause] . . .] *new-line*

C/C++ for loop

The clauses (f77) can be:

- **PRIVATE (*list*)**
- **FIRSTPRIVATE (*list*)**
- **LASTPRIVATE (*list*)**
- **REDUCTION ({*operator* | *intrinsic*} : *list*)**
- **SCHEDULE (*type*[,*chunk*])**
- **ORDERED**

Similarly for C/C++, with the addition of:

- **nowait**

There is an implicit barrier at the end of a **for** construct unless the **nowait** clause is specified.

The **DO/for** directives provide a mechanism for the distribution of loop iterations across the available threads in a parallel region.

If the **!$OMP END DO** directive is excluded the **!$OMP DO** is assumed to end with the enclosed do loop. There is an implicit barrier after the end of the parallel loop, that is the first thread to complete its portion of work will wait until the other threads have finished before continuing. If the option **NOWAIT** is specified, the threads will not synchronise at the end of the parallel loop, that is, the first thread to finish will then start on the next piece of code.

See section 4.5.6 for a description on the use of the **ORDERED** directive.

Where the parallel region contains a single **DO/for** directive the following short-cut can be used:

f77:

!$OMP PARALLEL DO [*clauses*]

do_loop

[!$OMP END PARALLEL DO]

C/C++:

#pragma omp parallel for [*clauses*] new-line

for loop

which is equivalent to explicitly specifying a **PARALLEL/parallel** directive immediately followed by a **DO/for** directive.

In C/C++ there are several restrictions on the structure of the **for** loop, which essentially boil down to the **for** loop looking like a fortran **do** loop, i.e. it must have a *canonical* shape. For a more detailed description see the standard.

4.4.2 SECTIONS

Syntax:

f77:

```
!$OMP SECTIONS [clauses]
[!$OMP SECTION]
block
[!$OMP SECTION
block]
. . .
!$OMP END SECTIONS [NOWAIT]
```

C/C++:

```
#pragma omp sections [clause[ clause] . . . ] new-line
  {
    [#pragma omp section new-line]
      structured-block
    [#pragma omp section new-line
      structured-block
      . . . ]
  }
```

where *clauses* can be any of the following for f77:

- **PRIVATE**
- **FIRSTPRIVATE**
- **LASTPRIVATE**
- **REDUCTION**

Similarly for C/C++ with the addition of **nowait**. There is an implicit barrier at the end of a **sections** construct, unless a **nowait** is specified.

The **SECTIONS/sections** directives causes the sections of code within the construct to be divided among threads in the team such that each section is executed once by a thread in the team.

Each section is preceded by a **SECTION/section** directive (optional for the first section). For f77, threads that complete execution of their sections wait at a barrier at the **END SECTIONS** directive unless a **NOWAIT** is specified.

As for the **DO/for** directive there is a short-cut for specifying a parallel region that contains a single **SECTIONS**/sections directive:

f77:

```
!$OMP PARALLEL SECTIONS [clauses]
[!$OMP SECTION]
block
[!$OMP SECTION
block]
. . .
!$OMP END PARALLEL SECTIONS
```

C/C++:

```
#pragma omp parallel sections [clauses] new-line
  {
    [#pragma omp section new-line]
      structured-block
    [#pragma omp section new-line]
      structured-block
      . . . ]
  }
```

which is equivalent to explicitly specifying a **PARALLEL/parallel** directive immediately followed by a **SECTIONS/sections** directive.

4.4.3 SINGLE

Syntax:

f77:

```
!$OMP SINGLE [clauses]
block
!$OMP END SINGLE [NOWAIT]
```

C/C++:

```
#pragma omp single [clauses] new-line
structured-block
```

Where *clauses* can be any of the following:

- **PRIVATE**
- **FIRSTPRIVATE**

with the additional **nowait** for C/C++. There is an implicit barrier after the **single** construct unless a **nowait** clause is specified.

This directive specifies that the enclosed code is to be executed by only one thread in the team which is necessary for portions of code lying in the parallel region which must be executed serially. In f77, threads not executing the **SINGLE** directive wait at the **END SINGLE** unless **NOWAIT** is specified.

4.5 Synchronisation constructs

These constructs allow the user to manipulate the thread behaviour in a parallel region.

4.5.1 MASTER

Syntax:

f77:

```
!$OMP MASTER
block
!$OMP END MASTER
```

C/C++:

```
#pragma omp master new-line
 structured-block
```

The code enclosed by these directives is executed by the master thread of the team. The other threads skip the enclosed code and continue execution (there is no implied barrier either on entry or exit from the master section).

4.5.2 CRITICAL

Syntax:

f77:

```
!$OMP CRITICAL [(name)]
block
!$OMP END CRITICAL [(name)]
```

C/C++:

```
#pragma omp critical [(name)] new-line
  structured-block
```

The code enclosed by these directives is accessed by only one thread at a time. A thread waits at the beginning of a critical section until no other thread in the team is executing a critical section with the same name. The optional *name* identifies a particular critical section and if used in f77 must be specified on both the `CRITICAL` and `END CRITICAL` directives.

4.5.3 BARRIER

Syntax:

f77:

```
!$OMP BARRIER
```

C/C++:

```
#pragma omp barrier new-line
```

This directive synchronises the threads in a team by causing them to wait until all of the other threads have reached this point in the code.

4.5.4 ATOMIC

Syntax:

f77:

```
!$OMP ATOMIC
```

C/C++:

```
#pragma omp atomic new-line
```

This directive ensures that a specific memory location is to be updated atomically rather than exposing it to the possibility of multiple, simultaneous writing threads. It applies only to the statement following on immediately after the directive, which must have one of the following forms:

f77:

```
x = x operator expr
x = expr operator x
x = intrinsic (x, expr)
x = intrinsic (expr, x)
```

where

- **x** is a scalar variable of intrinsic type
- **expr** is a scalar expression that does not reference **x**
- intrinsic is one of **MAX**, **MIN**, **IAND**, **IOR**, or **IEOR**
- operator is one of **+**, *, **-**, **/**, **.AND.**, **.OR.**, **.EQV.**, or **.NEQV.**

C/C++:

```
x binop = expr
x++
++x
x--
--x
```

where

- **x** is an lvalue expression with scalar type.
- **expr** is an expression with scalar type, and it does not reference the object designated by x.
- **binop** is not an overloaded operator and one of +, *, -, /, &, ^, |, <<, or >>.

In parallel, if an object is updated using this directive, then all references to that object must use this directive to avoid race conditions.

4.5.5 FLUSH

Syntax:

f77:

!$OMP FLUSH [(*list*)**]**

C/C++:

#pragma omp flush [(*list***)]** *new-line*

This directive causes thread visible variables to be written back to memory and is provided for users who wish to write their own synchronisation directly through shared memory. The optional *list* may be used to specify variables that need to be flushed, otherwise all variables are flushed to memory. The directive is implied for the following directives (unless the **NOWAIT/nowait** clause is present):

- **BARRIER/barrier**
- **CRITICAL and END CRITICAL/entry to and exit from critical**
- **END DO/exit from for**
- **END PARALLEL/exit from parallel**
- **END SECTIONS/exit from sections**
- **END SINGLE/exit from single**

- `ORDERED` and `END ORDERED/entry to and exit from ordered`

4.5.6 ORDERED

Syntax:

f77:

```
!$OMP ORDERED
block
!$OMP END ORDERED
```

C/C++:

```
#pragma omp ordered new-line
    structured-block
```

This directive must appear within a `DO/for` or `PARALLEL DO/parallel for` directive, which in turn must have the `ORDERED/ordered` clause specified. Only one thread a time is allowed into an ordered section. The threads executing the iterations of the `DO/for` section are allowed to enter the ordered section in the same order as the iterations are executed in the sequential version of the loop. This sequentialises and orders code within the ordered sections while allowing code outside the section to run in parallel.

4.6 Conditional compilation

The OpenMP Fortran API permits Fortran statements to be compiled conditionally. The sentinel used is `!$` and must be followed by a legal Fortran statement, for example:

```
!$ 10 IAM = OMP_GET_THREAD_NUM() + INDEX
```

During OpenMP compilation the sentinel is replaced by two spaces and the rest of the line is treated as a normal Fortran statement. Also a C preprocessor macro can be used for conditional compilation:

```
#IFDEF _OPENMP
   10 IAM = OMP_GET_THREAD_NUM() + INDEX
#ENDIF
```

OpenMP-compliant compilers define this macro during compilation, but essentially the two forms are equivalent.

The `_OPENMP` macro name is defined by OpenMP-compliant implementations as the decimal constant *yyyymm*, which will be the year and month of the approved specification. This macro must not be the subject of a `#define` or a `#undef` preprocessing directive.

5 Library Routines and Environment Variables

5.1 Execution Environment Routines

These routines can be used to control and query the parallel execution environment. For the C/C++ routines the OpenMP header file must be included, i.e.:

```
#include <omp.h>
```

must appear before the use of the functions.

5.1.1 OMP_SET_NUM_THREADS

subroutine OMP_SET_NUM_THREADS (*scalar_integer_expression*)

void omp_set_num_threads(int *num_threads*);

The ***scalar_integer_expression*** is evaluated and its value is used to set the number of threads to use for the next parallel region.***num_threads*** must be positive. This function only has effect when called from serial portions of the program. When dynamic adjustment of the number of threads is enabled, this subroutine sets the maximum number of threads to use for the next parallel region.

This call has precedence over the **OMP_NUM_THREADS** environment variable.

5.1.2 OMP_GET_NUM_THREADS

```
integer function OMP_GET_NUM_THREADS()
```

```
int omp_get_num_threads(void);
```

This function returns the number of threads currently in the team executing the parallel region from which it is called, or 1 if called from a serial portion of the code. If the number of threads has not been explicitly set by the user, the default is implementation dependent.

5.1.3 OMP_GET_MAX_THREADS

```
integer function OMP_GET_MAX_THREADS()
```

```
int omp_get_max_threads(void);
```

This returns the maximum value that can be returned by calls to **OMP_GET_NUM_THREADS**.

5.1.4 OMP_GET_THREAD_NUM

```
integer function OMP_GET_THREAD_NUM()
```

```
int omp_get_thread_num(void);
```

This function returns the thread number within the team that lies between **0** (the master thread) and **OMP_GET_NUM_THREADS()-1** inclusive.

5.1.5 OMP_GET_NUM_PROCS

```
integer function OMP_GET_NUM_PROCS()
```

```
int omp_get_num_procs(void);
```

This function returns the number of processors that are available to the program.

5.1.6 OMP_IN_PARALLEL

```
logical function OMP_IN_PARALLEL()
```

```
int omp_in_parallel(void);
```

This function returns `.TRUE./non-zero` if it is called from the dynamic extent of a region executing in parallel and `.FALSE./0` otherwise. A parallel region that is serialised is not considered to be a region executing in parallel. However, this function will always return `.TRUE./non-zero` within the dynamic extent of a region executing in parallel, regardless of nested regions that are serialised.

5.1.7 OMP_SET_DYNAMIC

subroutine OMP_SET_DYNAMIC(*scalar_logical_expression***)**

void omp_set_dynamic(int *dynamic_threads***);**

This subroutine enables or disables dynamic adjustment of the number of threads available for execution of parallel programs. If *scalar_logical_expression/dynamic_threads* evaluates to `.TRUE./non-zero`, the number of threads that are used for executing subsequent parallel regions can be adjusted automatically by the run-time environment to obtain the best use of system resources. As a consequence, the number of threads specified by the user is the maximum thread count.The number of threads always remains fixed over the duration of each parallel region and is reported by the `OMP_GET_NUM_THREADS()` function. If *scalar_logical_expression/dynamic_threads* evaluates to zero, dynamic adjustment is disabled.

A call to this subroutine has precedence over the `OMP_DYNAMIC` environment variable.

5.1.8 OMP_GET_DYNAMIC

```
logical function OMP_GET_DYNAMIC()
```

```
int omp_get_dynamic(void);
```

This function returns `.TRUE./non-zero` if dynamic thread adjustment is enabled, `.FALSE./0` otherwise.

5.1.9 OMP_SET_NESTED

subroutine OMP_SET_NESTED(*scalar_logical_expression***)**

void omp_set_nested(int *nested***);**

If *scalar_logical_expression/nested* evaluates to `.FALSE./0` (the default), then nested parallelism is disabled, and such regions are serialised and executed by the current thread. If set to `.TRUE./non-zero`, nested parallelism is enabled, and parallel regions that are nested can deploy additional threads to form the team, but recall that the number of threads in the teams is implementation dependent.

This call has precedence over the `OMP_NESTED` environment variable.

5.1.10 OMP_GET_NESTED

```
logical function OMP_GET_NESTED()
```

```
int omp_get_nested(void);
```

This function returns `.TRUE./non-zero` if nested parallelism is enabled and **`.FALSE.`/0** if nested parallelism is disabled. If an implementation does not implement nested parallelism, this function always returns 0.

5.2 Lock Routines

This section details the OpenMP general-purpose locking routines which are used to guarantee that only one process accesses a variable at a time to avoid race conditions. For all these routines the lock variable *var* should be of type integer and of a precision large enough to hold an address. The C/C++ lock variable must have type `omp_lock_t` or `omp_nest_lock_t`. All lock functions require an argument that has a pointer to `omp_lock_t` or `omp_nest_lock_t` type. Also the `omp.h` file must be included.

5.2.1 OMP_INIT_LOCK

```
subroutine OMP_INIT_LOCK(var)
```

```
void omp_init_lock(omp_lock_t *lock);
```

```
void omp_init_nest_lock(omp_nest_lock_t *lock);
```

These functions initialise a lock associated with lock variable *var* or parameter *lock* for use in subsequent calls. For a nestable lock, the initial nesting count is zero.

5.2.2 OMP_DESTROY_LOCK

```
subroutine OMP_DESTROY_LOCK(var)
```

```
void omp_destroy_lock(omp_lock_t *lock);
```

```
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
```

This subroutine dissociates the given lock variable *var* or parameter *lock* from any locks.

5.2.3 OMP_SET_LOCK

```
subroutine OMP_SET_LOCK(var)
```

```
void omp_set_lock(omp_lock_t *lock);
```

```
void omp_set_nest_lock(omp_nest_lock_t *lock);
```

This subroutine forces the executing thread to wait until the specified lock is available. The thread is granted ownership of the lock when it is available.

5.2.4 OMP_UNSET_LOCK

```
subroutine OMP_UNSET_LOCK(var)
```

```
void omp_unset_lock(omp_lock_t *lock);
```

```
void omp_unset_nest_lock(omp_nest_lock_t *lock);
```

This releases the executing thread from ownership of the lock. For a nested lock, the function `omp_unset_nest_lock` decrements the nesting count, and releases the thread executing the function from ownership of the lock if the resulting count is zero.

5.2.5 OMP_TEST_LOCK

```
logical function OMP_TEST_LOCK(var)
```

```
int omp_test_lock(omp_lock_t *lock);
```

```
int omp_test_nest_lock(omp_nest_lock_t *lock);
```

This function tries to set the lock associated with the lock variable, returning **.TRUE./non-zero** if the lock was set successfully and **.FALSE./0** otherwise. For a nestable lock, the **omp_test_nest_lock** function returns the new nesting count if the lock is successfully set; otherwise, it returns zero.

5.3 Environment Variables

These environment variables can be used to control the execution of parallel code. The names must be uppercase, the values assigned to them are case insensitive.

5.3.1 OMP_SCHEDULE

Example:

```
setenv OMP_SCHEDULE "GUIDED,4"
```

This applies to **DO** and **PARALLEL DO** that have the schedule type **RUNTIME**. The schedule type and chunk size for all such loops can be set at run time by setting this environment variable to any of the recognized schedule types and optional chunk size (see Table 1).

5.3.2 OMP_NUM_THREADS

Example:

```
setenv OMP_NUM_THREADS 16
```

This sets the number of threads to use during execution unless that number is explicitly changed using the subroutine **OMP_SET_NUM_THREADS** (section 5.1.1). If dynamic adjustment of the number of threads is enabled this variable is the maximum number of threads to use.

5.3.3 OMP_DYNAMIC

Example:

```
setenv OMP_DYNAMIC TRUE
```

This enables (**TRUE**) or disables (**FALSE**) the dynamic adjustment of the number of threads available for execution of parallel regions.

5.3.4 OMP_NESTED

Example:

```
setenv OMP_NESTED TRUE
```

This enables (**TRUE**) or disables (**FALSE**) nested parallelism.

6 Performance and Scalability

To examine the performance of OpenMP it was decided it to compare it with other shared memory directives and against different parallelisation paradigms, namely MPI and HPF.

6.1 The Game of Life

The Game of Life is a simple grid-based problem which demonstrates complex behaviour. It is a cellular automaton where the world is a 2D grid of cells which have two states: alive or dead. At each iteration the new state of the cell is determined by the state of its neighbours at the previous iteration as seen in Figure 7.

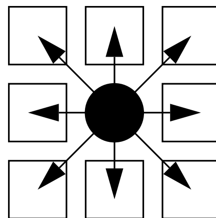

Figure 7: The Game of Life cell and its neighbours

The rules for the evolution of the system are;

- if a cell has exactly two live neighbours it maintains state
- if it has exactly three live neighbours it is (or becomes) alive
- otherwise, it is dead (or dies)

The figure below shows the serial code for the game of life, with OpenMP directives added for parallel execution. As we are looking at performance all the I/O has been inhibited and the output subroutine is not included here.

```
      program game_of_life
      implicit none
      integer i,j,loop,num_alive,n, maxloop
      parameter (n=512, maxloop=10000)
      integer board(0:n+1,0:n+1),num_neigh(n,n)
      logical output
      parameter (output = .false.)

c Initialise board (a simple + pattern)
      num_alive=0
!$OMP PARALLEL DO REDUCTION(+:num_alive)
      do j = 1, n
```

```
        do i = 1, n
          if ((i.eq.n/2).or.(j.eq.n/2)) then
            board(i,j) = 1
            num_alive = num_alive+1
          else
            board(i,j) = 0
          endif
        enddo
      enddo
      if (output) write(6,1111) num_alive
c Output initial board
      if (output) call output_board(board,n,0)

c Perform maxloop updates of the board
      do loop = 1, maxloop

c Edges have periodic boundary conditions
!$OMP PARALLEL DO
        do i = 1, n
          board(i,0) = board(i,n)
          board(i,n+1) = board(i,1)
          board(0,i) = board(n,i)
          board(n+1,i) = board(1,i)
        enddo

c loop over board generating data for number of neighbours
!$OMP PARALLEL DO
        do j = 1, n
          do i = 1, n
            num_neigh(i,j) = board(i,j+1) + board(i,j-1)
     &                     + board(i+1,j) + board(i-1,j)
     &                     + board(i+1,j+1) + board(i-1,j-1)
     &                     + board(i+1,j-1) + board(i-1,j+1)
          enddo
        enddo

c Update board and calculate the number of cells alive
        num_alive = 0
!$OMP PARALLEL DO REDUCTION(+:num_alive)
        do j = 1, n
          do i = 1, n
            if ((num_neigh(i,j).lt.2).or.(num_neigh(i,j).gt.3))
then
              board(i,j) = 0
```

```
              else if (num_neigh(i,j).eq.3) then
                board(i,j) = 1
              endif
              if (board(i,j) .eq. 1) num_alive=num_alive+1
            enddo
          enddo
          if (output) write(6,1111) num_alive
c Output board
          if (output) call output_board(board, n, loop)

        enddo
 1111 format('Number alive = ',i4)
        end
```

Figure 8: Code for Game of Life with OpenMP directives

6.2 Performance

As we can see from Figure 8 the game of life is quite straightforwardly parallelisable using OpenMP. Below are the equivalent Sun directives and SGI parallel directives in comparison with OpenMP.

- OpenMP

  ```
  !$OMP PARALLEL DO
  !$OMP PARALLEL DO REDUCTION(+:num_alive)
  ```

- Sun

  ```
  C$PAR DOALL
  C$PAR DOALL REDUCTION(num_alive)
  ```

- SGI

  ```
  C$DOACROSS PRIVATE(i,j)
  C$DOACROSS PRIVATE(i,j), REDUCTION(num_alive)
  ```

Also the MPI and HPF versions of the code were written (full versions in Appendix A and Appendix B).

6.2.1 Sun Enterprise 3000

The MPI, OpenMP, HPF and Sun versions were run on a Sun SparcStation E3000 with four processors and 1 gigabyte of memory. The OpenMP version was compiled using the Guide compiler from Kuck & Associates Inc. (`http://www.kai.com`). The HPF version was compiled using the Portland Group HPF compiler, ***pghpf*** (version 2.4) (`http://www.pgroup.com`).

Table 3 and Table 4 show the timings (in seconds) for the game of life with gridsizes of 250×250, 500×500 and 1000×1000 respectively.

Table 3: Problem size = 250x250, Serial f77 version time = 79.6 seconds

NPES	1	2	3	4
Sun	80.0	41.6	29.1	21.9
OpenMP	79.6	41.9	29.1	21.4
MPI	93.2	49.9	35.6	29.9
HPF	205.0	103.1	76.6	64.0

Table 4: Problem size = 1000x1000, Serial f77 version time = 1812.6 seconds

NPES	1	2	3	4
Sun	1824	941.7	667.4	476.7
OpenMP	1812	943.6	669.2	490.7
MPI	1916	998.1	681.1	500.3
HPF	4900	2615	1748	1391

As we are interested in the scaling properties of the various parallelisation techniques, Figure 9 and Figure 10 show the speedup obtained for the different paradigms. Speedup is defined in this case to be :

$$Speedup = T(1)/T(n) \tag{1}$$

where $T(n)$ is the time for execution of the particular code on n processors, rather than the more usual :

$$Speedup = T_1/T(n) \tag{2}$$

where T_n is the time for execution of the serial version of the code. This is so we can compare the scalability of HPF with the other paradigms, as the times for the MPI, OpenMP and Sun codes on one processor are very close to the time for the serial code whereas the HPF code is approximately 2.5-3 times slower. One would perhaps expect this as the HPF code is Fortran90 which is generally slower than Fortran77. Also, the method of parallelism is based on the intrinsic data parallelism of HPF, using the `cshift` command rather than the `DO` utlised in the other versions, but the results are included here anyway for the sake of completeness.

Examining figures 7 and 8 one sees that all the paradigms perform well, in particular for the larger problem size. However, perhaps one might not expect to see too much difference over a small number of processors. It is still encouraging that the OpenMP version performs as well as the other more established paradigms.

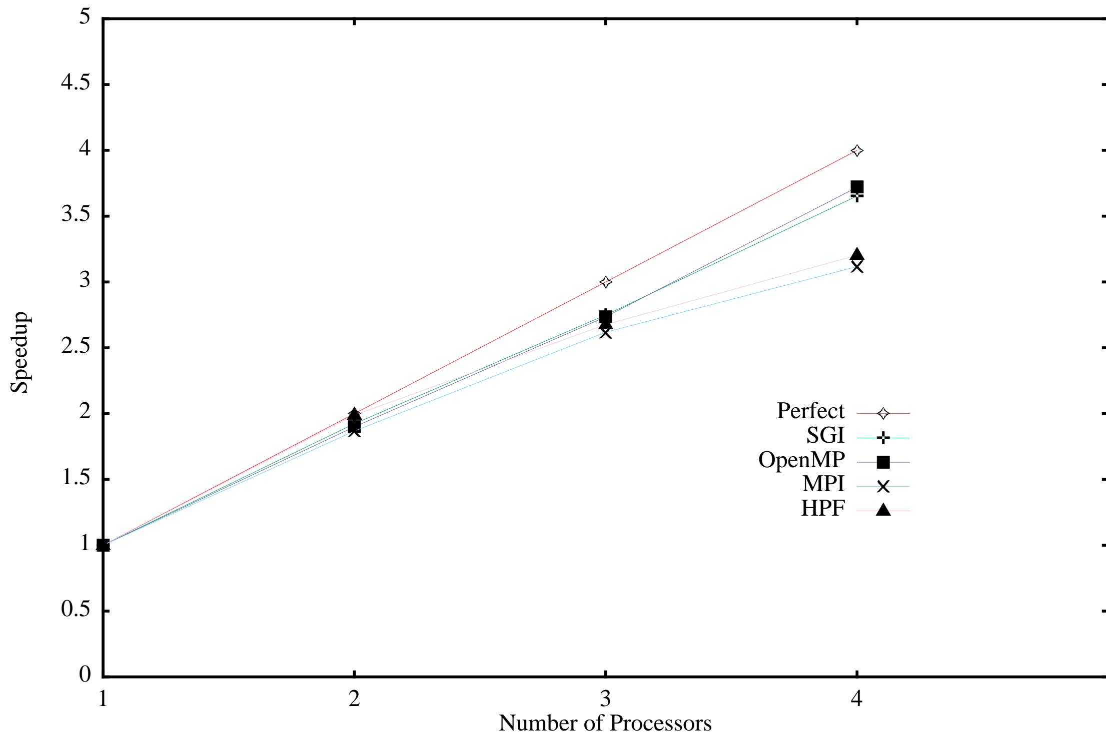

Figure 9: Parallel performance on the Sun E3000 for the 250 x 250 game of life grid

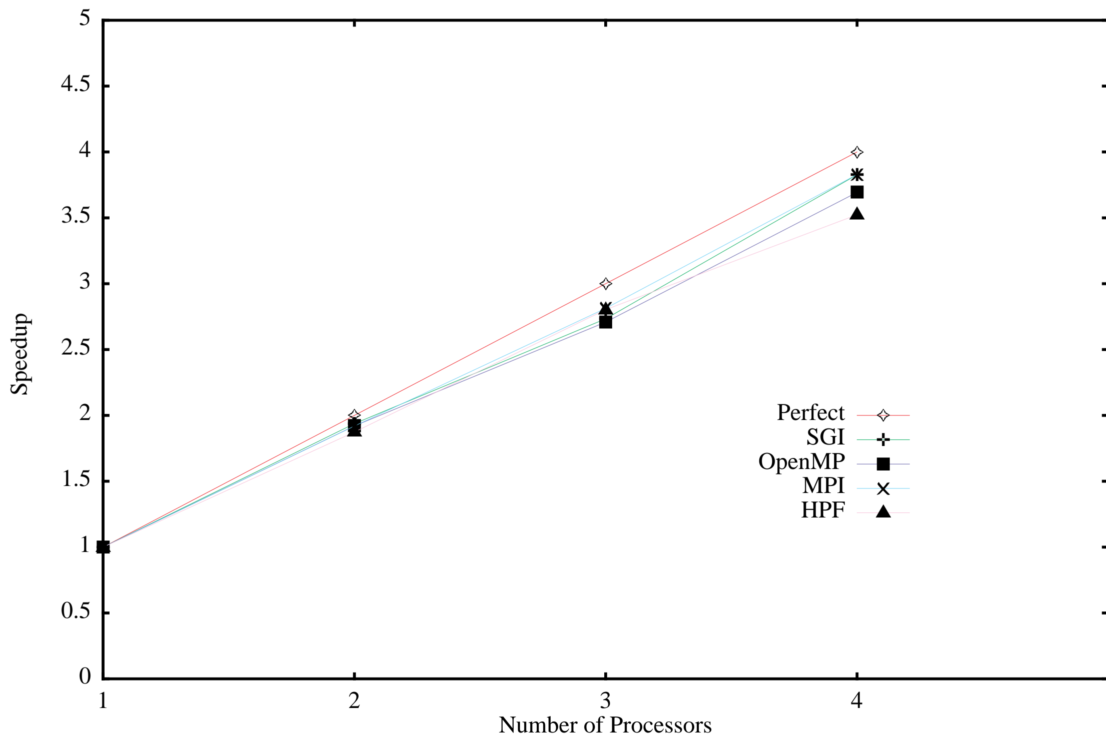

Figure 10: Parallel performance on the Sun E3000 for the 1000 x 1000 game of life grid

6.2.2 SGI Origin 2000

The MPI, OpenMP and SGI versions of the code were run on the SGI Origin2000 at the Manchester Computing Centre. The Origin 2000 is a distributed shared memory machine, that is each node has its own memory which every other node has access to through the global address space. Each node consists of two processors, and the nodes are inter-connected in such a way as to create an augmented hypercube, a section of which is shown in Figure 11.

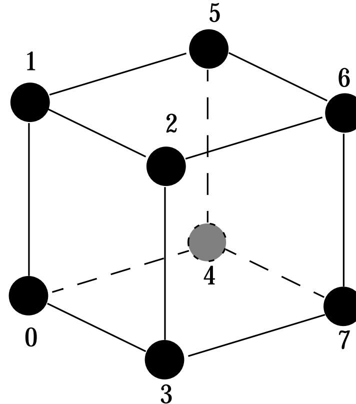

Figure 11: Node interconnections on the Origin 2000

Figure 12 and Figure 13 show the speedup for the three different versions of the code for a 512×512 and 1024×1024 gridsize respectively. HPF was not available on the Origin2000 so the second definition of speedup (Equation 2 from Section 6.2.1) was used for generating these graphs.

Comparing the two graphs one can see that for the smaller problem size the overhead of running using multiple threads was much more noticeable than for the larger problem size. For the larger grid all three paradigms perform well as the number of processors are increased, with near perfect speedup for the OpenMP and SGI directives, and super-linear speedup for the MPI version (this may be due to inefficiencies for the serial version of the code caused by poor cache management for the large data space).

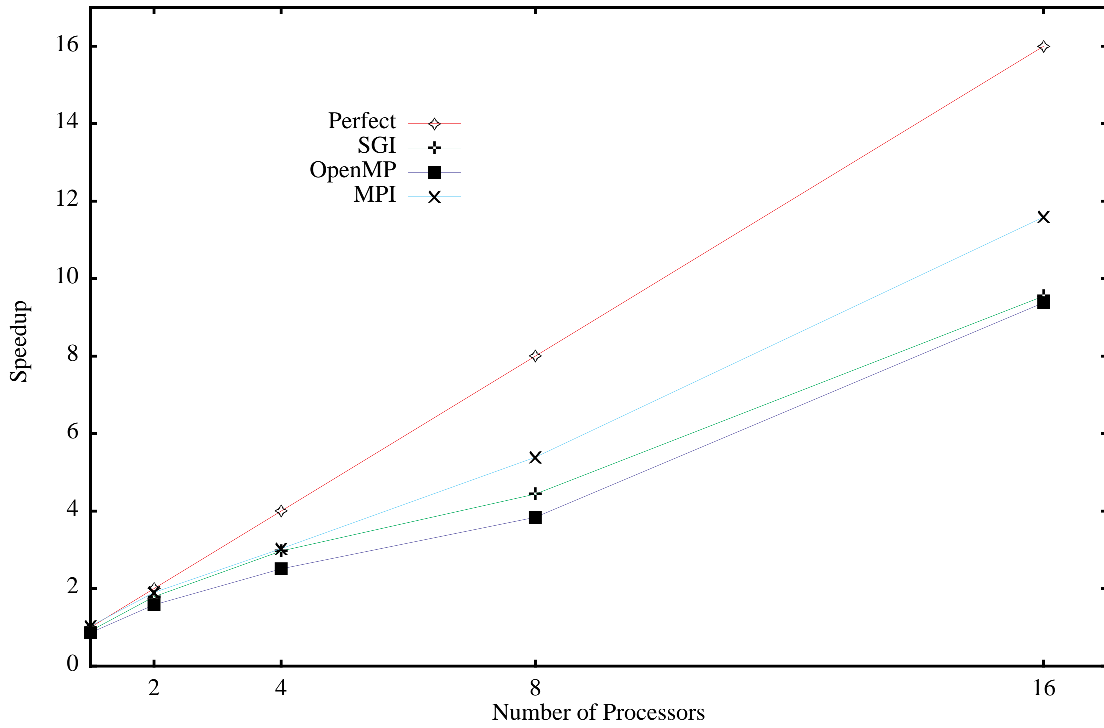

Figure 12: Performance for a 512x512 Game of Life grid on the Origin2000

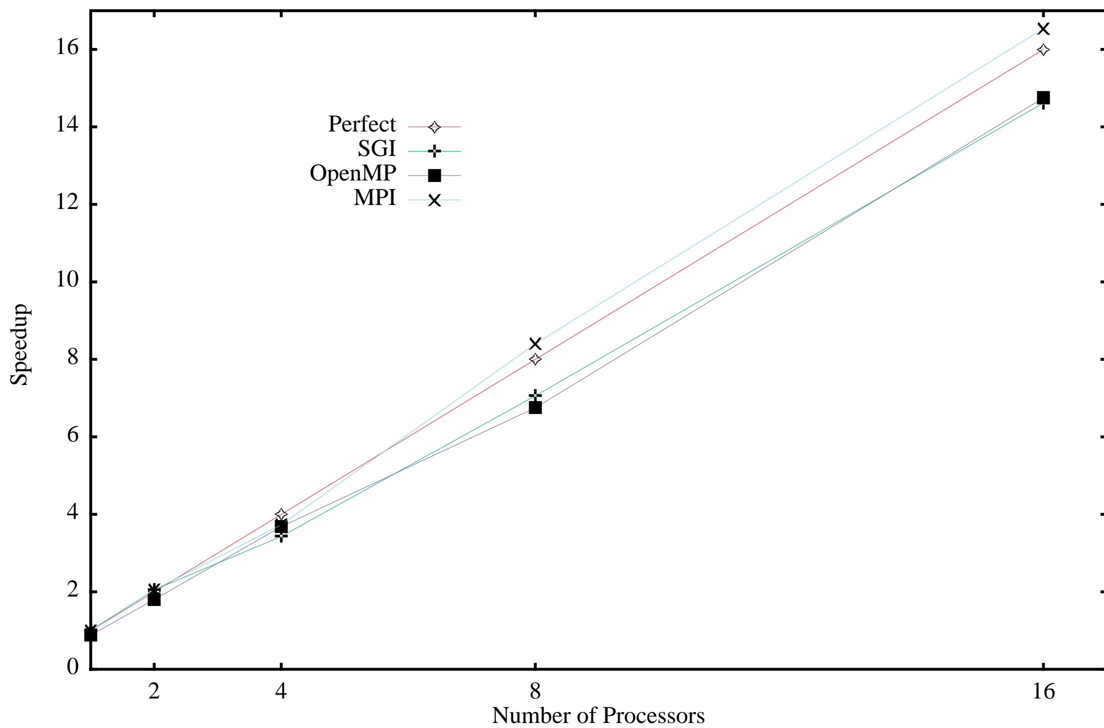

Figure 13: Performance for a 1024x1024 Game of Life grid on the Origin2000

7 References

1. X3H3 Committee. *Parallel Extensions for Fortran*. Technical Report X3H5/93-SD1-Revision M, Accredited Standards Committee X3, April 1994.
2. Klaas Jan Wieranga. *Survey of Compiler Directives for Shared Memory Programming*. EPCC TEC-WATCH report, March 1997 (`http://www.epcc.ed.ac.uk/epcc-tec/documents.html`).
3. The HPF Forum. *HPF-2 Information*. (`http://www.vcpc.univie.ac.at/information/mirror/HPFF/versions/hpf2/`)
4. The OpenMP Architecture Review Board. *OpenMP: A Proposed Industry Standard API for Shared Memory Programming*. White Paper, October 1997 (`http://www.openmp.org/openmp/mp-documents/paper/paper.html`)

8 Acknowledgements

Thanks to :

- Barbara Früh of the Johannes Gutenberg University of Mainz, Insitute for Atmospheric Physics and Elspeth Minty of EPCC for the MPI version of the Game of Life.
- Bob Kuhn of Kuck & Associates, Inc.
- Gavin Pringle of EPCC for assistance with the HPF code.

OpenMP is a trademark of the OpenMP Architecture Review Board. Portions of this publication may have been derived from the OpenMP Language Application Program Interface Specification.

A MPI version of the Game of Life

```
      program game
      implicit none

      include   'mpif.h'
c-------------------------------------
c variables for the iteration
      integer   iter,maxiter
      parameter (maxiter=100)

c-------------------------------------
c variables for the basic information
      integer   errcode,rank,size

c---------------------------------------------
c variables for dim create and cart. topology
      integer   ndims, TWODIM_COMM
      parameter (ndims=2)
      integer   dims(2)
      logical   periods(2),reorder
      integer   nx,ny, xglob,yglob

c-----------------------
c variables for derived datatype and file access
      integer   numberblocks,blocklength,stride,SUBDOM,ROW,COLUMN
      integer   sizes(2), subsizes(2), starts(2), MEMTYPE
      integer   fh, fstat(MPI_STATUS_SIZE)

c-----------------------
c variables of the field
      integer   i,j,xsize,ysize,maxgrey
      parameter (xsize=8000,ysize=8000,maxgrey=1)
      integer cell(0:xsize+1,0:ysize+1) ! enlarged array for the halo
      integer numneigh(xsize,ysize)

c---------------------------------------------
c variables to determine the pos. of the proc.
      integer   coords(ndims)

c------------------------
c variables for the shift
      integer   direction,disp,left_neighb,right_neighb,above_neighb,
     &          below_neighb

c----------------------------------
c variables for the send and receive
      integer       count,tag,dx,dy
      integer       request(4)
      integer   status(MPI_STATUS_SIZE, 4)
      character picfile*10

      integer num_alive, total_alive
      logical output
      parameter (output=.false.)

c--------------------------------------------------------------------
```

```
c basic information
      call MPI_INIT (errcode)

      call MPI_COMM_RANK (MPI_COMM_WORLD,rank,errcode)
      call MPI_COMM_SIZE (MPI_COMM_WORLD,size,errcode)

c------------------------------------------------------------------
c create dimensions
      dims(1) = 0
      dims(2) = 0
      call MPI_DIMS_CREATE(size,ndims,dims,errcode)
      nx = dims(1)
      ny = dims(2)
      dx = xsize/nx
      dy = ysize/ny

c------------------------------------------------------------------
c create 2-dimensional, periodic, cartesian grid
      do i=1,ndims
         periods(i) = .TRUE.    ! cyclic boundaries
      enddo
      reorder    = .TRUE.       ! not yet distributed

      call MPI_CART_CREATE(MPI_COMM_WORLD,ndims,dims,periods,reorder,
     &     TWODIM_COMM,errcode)

c------------------------------------------------------------------
c determination of the cartisian coordinates of the processor (x,y)

      call MPI_CART_COORDS(TWODIM_COMM,rank,ndims,coords,errcode)

c------------------------------------------------------------------
c initialisation

      xglob = coords(1)*dx
      yglob = coords(2)*dy

      do j=0,dy+1
         do i=0,dx+1
            if (xglob+i.eq.xsize/2  .or.
     &          yglob+j.eq.ysize/2 ) then
               cell(i,j) = maxgrey ! white
            else
               cell(i,j) = 0       ! black
            endif
         enddo
      enddo

c------------------------------------------------------------------
c create derived datatype
c file layout
      sizes(1) = xsize
      sizes(2) = ysize
      subsizes(1) = dx
      subsizes(2) = dy
      starts(1) = coords(1) * dx
      starts(2) = coords(2) * dy
      call MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts,
     &       MPI_ORDER_FORTRAN, MPI_INTEGER, SUBDOM, errcode)
      call MPI_TYPE_COMMIT(SUBDOM,errcode)

c memory layout
      sizes(1) = dx+2
```

```
      sizes(2) = dy+2
      starts(1) = 1
      starts(2) = 1
      call MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts,
     &       MPI_ORDER_FORTRAN, MPI_INTEGER, MEMTYPE, errcode)
      call MPI_TYPE_COMMIT(MEMTYPE,errcode)

c row
      numberblocks = dy+2
      blocklength  = 1
      stride       = xsize+2
      call MPI_TYPE_VECTOR(numberblocks, blocklength, stride,
     &                     MPI_INTEGER, ROW, errcode)
      call MPI_TYPE_COMMIT(ROW,errcode)

c column
      numberblocks = 1
      blocklength  = dx+2
      stride       = xsize+2
      call MPI_TYPE_VECTOR(numberblocks, blocklength, stride,
     &                     MPI_INTEGER, COLUMN, errcode)
      call MPI_TYPE_COMMIT(COLUMN,errcode)

c--------------------------------------------------------------------
c find the nearest neighbour in the x-direction
      direction = 1             ! x-direction
      disp      = 1             ! direct neighbour
      call MPI_CART_SHIFT(TWODIM_COMM,direction,disp,
     &     left_neighb,right_neighb,errcode)

c--------------------------------------------------------------------
c find the nearest neighbour in the y-direction
      direction = 0             ! y-direction
      disp      = 1             ! direct neighbour
      call MPI_CART_SHIFT(TWODIM_COMM,direction,disp,
     &     above_neighb,below_neighb,errcode)

      count       = 1
      tag         = 1

c--------------------------------------------------------------------
      do iter=0,maxiter

c--------------------------------------------------------------------
c swap boundaries with the processor above and below

        call MPI_ISSEND(cell(dx,0),count,ROW,below_neighb,tag,
     &          TWODIM_COMM,request(1),errcode)
        call MPI_ISSEND(cell(1,0),count,ROW,above_neighb,tag,
     &          TWODIM_COMM,request(2),errcode)

        call MPI_IRECV(cell(0,0),count,ROW,above_neighb,tag,
     &          TWODIM_COMM,request(3),errcode)
        call MPI_IRECV(cell(dx+1,0),count,ROW,below_neighb,tag,
     &          TWODIM_COMM,request(4),errcode)

        call MPI_WAITALL(4,request,status,errcode)

c--------------------------------------------------------------------
c swap boundaries with the processor left and right

        call MPI_ISSEND(cell(0,1),count,COLUMN,left_neighb,tag,
     &          TWODIM_COMM,request(1),errcode)
```

```
         call MPI_ISSEND(cell(0,dy),count,COLUMN,right_neighb,tag,
      &           TWODIM_COMM,request(2),errcode)

         call MPI_IRECV(cell(0,dy+1),count,COLUMN,right_neighb,tag,
      &           TWODIM_COMM,request(3),errcode)
         call MPI_IRECV(cell(0,0),count,COLUMN,left_neighb,tag,
      &           TWODIM_COMM,request(4),errcode)

         call MPI_WAITALL(4,request,status,errcode)

c---------------------------------------------------------------------
c loop over local cells, counting neighbours

         do j=1,dy
           do i=1,dx
             numneigh(i,j)= (cell(i-1,j+1)+cell(i,j+1)+
      &                      cell(i+1,j+1)+cell(i-1,j)+cell(i+1,j)+
      &                      cell(i-1,j-1)+cell(i,j-1)+cell(i+1,j-1))
      &                      /maxgrey
           enddo
         enddo
c---------------------------------------------------------------------
c loop over local cells, updating life board

         do j=1,dy
           do i=1,dx
             if ((numneigh(i,j).lt.2).or.(numneigh(i,j).gt.3)) then
               cell(i,j) = 0       ! black
             else if (numneigh(i,j) .eq. 3) then
               cell(i,j) = maxgrey ! white
             endif
             if (cell(i,j) .eq. 1) num_alive=num_alive+1
           enddo
         enddo

c---------------------------------------------------------------------
cpg Perform a reduction operation on num_alive
          call MPI_REDUCE(num_alive, total_alive, 1, MPI_INTEGER,
      &                   MPI_SUM, 0, TWODIM_COMM)


c---------------------------------------------------------------------
c output
         IF (output .ne. .FALSE.) THEN
           write(picfile, fmt='(''life'', i2.2, ''.out'')') iter
           call MPI_FILE_OPEN (MPI_COMM_WORLD, picfile, MPI_MODE_CREATE+
      &        MPI_MODE_RDWR, MPI_INFO_NULL, fh, errcode)
           call MPI_FILE_SET_VIEW (fh, 0, MPI_INTEGER, SUBDOM,
      &        "native", MPI_INFO_NULL, errcode)
           call MPI_FILE_WRITE_ALL (fh, cell, 1, MEMTYPE, fstat, errcode)
           call MPI_FILE_CLOSE (fh, errcode)
         END IF

       enddo                      ! iteration loop
c---------------------------------------------------------------------
       call MPI_FINALIZE(errcode)
       end
```

B HPF version of the Game of Life

```
   program game_of_life
   implicit none
   integer, parameter :: n=250, maxloop=10000
   integer loop, num_alive
   integer, dimension(n,n) :: board,neigh

! Distribute data for HPF compiler
!HPF$ distribute(*,block)::board,neigh

! Initialise board
   board=0
   board(:,n/2)=1
   board(n/2,:)=1
   num_alive=sum(board)

! Perform maxloop updates of the board
   do loop=1, maxloop

! Calculate number of neighbours
   neigh=board+cshift(board,shift=-1,dim=1) &
              +cshift(board,shift=+1,dim=1)
   neigh=neigh+cshift(neigh,shift=-1,dim=2) &
              +cshift(neigh,shift=+1,dim=2)
   neigh=neigh-board

! Update board and calculate number alive
   where(neigh==3)board=1
   where(neigh<2.or.neigh>3)board=0
   num_alive=sum(board)

   end do
   end program
```