SOFTWARE ENVIRONMENT FOR nCUBE MASSIVELY PARALLEL SYSTEMS

Matthew Hall
nCUBE U.K. Limited

ABSTRACT

The traditional supercomputer market consists of very fast sequential systems, whose software environment is well understood. These systems however have serious limitations:

* Their speed is limited by the physics of silicon.
* They have limited scalability.
* They are very expensive.

The nCUBE system is a scalar "masssively parallel" computer. It incorperates a network of between 32 and 8,192 general purpose processing nodes. This approach offers a system that is highly scalable, has a relatively low cost, and does not approach the physical limits of silicon.

Programming an nCUBE system is very similar to that of sequential systems however to achieve scalability the implementation of the algorithm would require adaption to a parallel environment. Normally this involves partitioning and also communicating between nodes.

In many cases the "parallel code" will be almost idential to its equivalent "sequential code" this is illustrated in the text.

1. INTRODUCTION

Traditionally, the computer industry has offered ever faster, more distributed, more compact implementation of conventional, sequential computers - those which process a single instruction at a time, in a serial sequence. The programming of these machines is well understood, but their ultimate performance is limited by the speed of integrated circuits.

The low end of performance in traditional computers is represented by minicomputers and workstations. A more aggressive solution is offered by mainframes and, more recently, vector processing supercomputers - those which process ordered sets of data simultaneously.

These traditional solutions has serious limitations:

* *Limited Speed* - Maximum throughput is limited by the physics of silicon.
* *Limited Scalability* - At best, these systems have only limited ability to expand. Some of them can be expanded into tens of processors working together, but not into hundreds or thousands of processors.
* *Poor Reliability* - With few processors and tightly coupled (shared) memory, the systems have poor reliability. Component failures often have global effects.
* *High Cost* - System cost is high, measured in terms of performance. Typically, the systems have many components which frequently use exotic and expensive materials. Both initial cost and maintenance cost are high.

2. The nCUBE ARCHITECTURE

The nCUBE hardware architecture is based on a scalable network of between 32 and 8,192 general-purpose processing nodes and one or more UNIX-based hosts. The hosts are connected to the nCUBE network by I/O nodes, which are themselves built around general-purpose nCUBE processors. Both distributed and shared I/O are supported. I/O services can be provided directly by the I/O nodes or through the host.

The nCUBE system uses a multiple-instruction, multiple-data (MIMD) architecture in which each processor operates independently on the programs and data stored in its local memory. Each processing node runs like a stand-alone sequential computer, with up to 64 MBytes of local DRAM memory. The processing nodes coordinate their activity by communicating with each other and with the host. The data and control messages are passed on high-speed DMA communication channels supported by hardware routing, a routing method that is faster than store-and -forward methods. Any node can communicate via DMA with any other node on the network, including I/O nodes.

The host workstations serve as operator interfaces to the nCUBE network. Hosts can share their memory and I/O resources with the processors in the network. For large problems, a single host can use all the processors and memory in the nCUBE system. Alternatively, the nCUBE can be divided into subsets of processing nodes that are dedicated to individual hosts in a multi-host environment, in such configurations, computations and message

passing by one user are completely protected from other users on the network.

The topology of the nCUBE network is a hypercube - an *n-dimensional* cube. The processors in the hypercube can be imagined to lie at the nodes (vertices) of the cube, and neighbouring nodes are linked along the edges of the cube by the message-passing communication channels. Each node is therefore linked to its n neighbouring processors, and the entire hypercube contains 2^n processors.

nCUBE systems are expandable. Two nCUBE networks of the same dimension can be merged to form an nCUBE network of the next higher dimension. A single system clock drives all the processors.

3. STRUCTURING PARALLEL APPLICATIONS

The application algorithms that run on each nCUBE processing node are identical to those that run on sequential machines. To achieve efficiency, however, some parts of the programs that implement these algorithms need adaptation to the parallel environment. Typically, this involves two things:

* Partitioning the data and/or code among the nCUBE processors.
* Communicating between nCUBE processors and with the host.

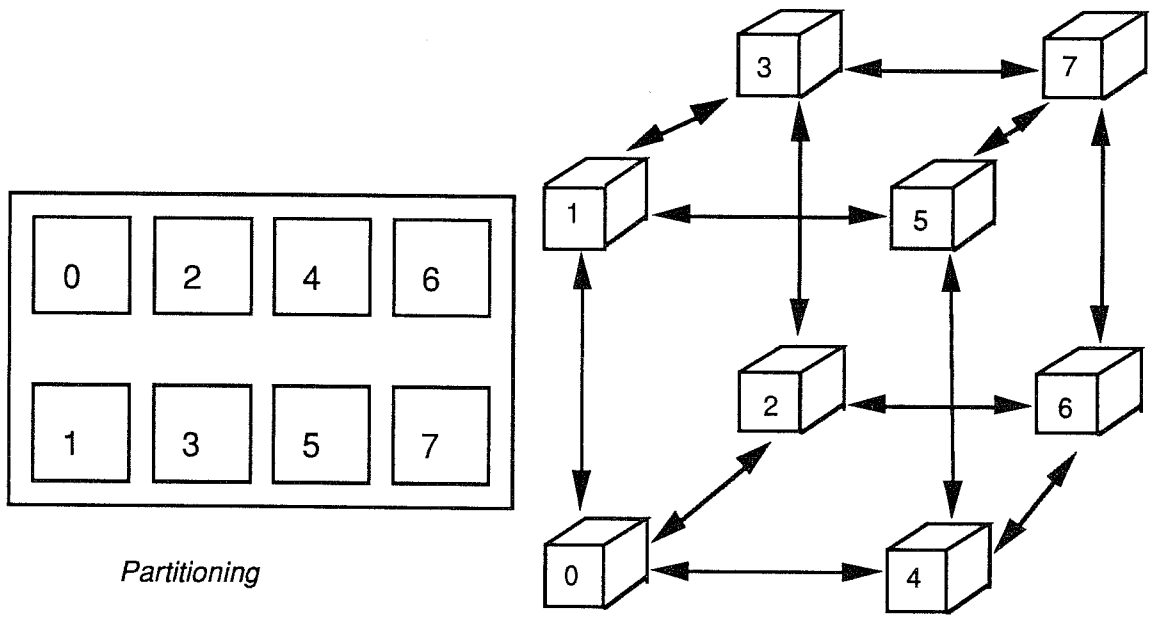

Figure 1

Before processing begins code and data need to be downloaded to the local memory in the processing nodes. During processing, partial results obtained at the boundaries of partitioned data or code in each node may need to be communicated to neighbouring nodes.

nCUBE library functions automate or simplify partitioning and communication. In applications such as matrix and finite-element problems, these library functions automatically perform the appropriate partitioning and interprocessor communication. Downloading of code and data to the local processors is also performed automatically. The examples to follow demonstrate a few of these nCUBE library functions. All standards UNIX, FORTRAN, and C functions can be used. nCUBE also provides extensions, including cross-compilers, high-level runtime environments, and support for the automated code-producing tools from third-party vendors such as ParaSoft and Pacific Sierra Research (PSR).

You can combine sequential and parallel operations. Portions of your application that cannot conveniently be partitioned and do not require maximum processing efficiency can be left intact and run on a single nCUBE processor without modification. Such an approach may minimize software development time at only a modest cost in processing time.

3.1 Partitioning

To run efficiently in a parallel environment, a sequential application must be partitioned (or decomposed). This involves dividing the data and/or code among the available or allocated nCUBE processors. It may also involve changing DO-loop limits, array dimensions, and subroutine parameters so that each processor can operate on a subset of the data.

There are two basic approaches to partitioning:

Partition the Data - Divide the data for a program among the available nCUBE processors so that each processor's subset of data is processed by identical copies of the program. This is also referred to as *parallelism by data*. Examples of data partitioning include:

* Each processor handles fixed sub-region of the total data region.
* Each processor handles a subset of the total-data, wherever the data are located in the data region.

* Each processor handles a variable-sized sub-region of the total data region, with the same number of data in each region.

Partition the Program - Divide the instructions in a program among the available nCUBE processors so that each processor's subset of instructions processes all of the data. This is also referred to as *parallelism by function.* Examples of program partitioning include:

* Some processors handle only I/O.
* Some processors handle only the user interface.
* Some processors handle only database searches or data reformatting.

The nCUBE library functions follow general rules for partitioning. These rules apply to all programs, including non-standard programs whose partitioning cannot be automated by library routines.

Balance the Computation Load at Each Processor - Efficient parallel applications use approximately the same processing time on all processors. By partitioning problems so that each processor has a similar processing load, some processors will not be idle while others are processing.
Balance the Memory Usage of Large Data Structures - Look at the large read/write data structures. If these structures are the main objects of computation, if they can be partitioned so as to give a balanced share of processing to each processor, and if they can fit within the memory available at each nCUBE processor, then an efficient parallel application can usually be based on their partitioning.

3.2 Communication

Communication is often required to :

* Download applications from the front-end processor to each nCUBE processing node.
* Exchange data between the processing node at data-partition boundaries.
* Exchange messages between the processing nodes at function-partition boundaries.
* Upload results from the processing nodes to the front-end processor for final assembly and presentation.

As in partitioning, nCUBE library functions automate or simplify these programming tasks. For example, nCUBE library functions perform such things as global summations across all processors, using a logarithmic broadcast tree which is transparent to the programmer. All nCUBE processors are networked together by DMA channels with hardware routing. Gray codes are used for processor addressing although nCUBE library functions make these details transparent to the programmer.

nCUBE systems are well-suited to the *loosely synchronous* method of communicating between processors. In this method, the activities of all processors are synchronized through paired writes and reads. A source processor writes to the destination processor, and the destination processor then reads the message. The read is *blocking*; it prevents further processing until it is finished, thereby ensuring that the application runs in an orderly manner. The synchronization occurs when processors are both senders and receivers, as in regular matrix applications where submatrix boundary data is exchanged.

Not all parallel applications need interprocessor communication. Our first example in the next section does not use it, because the partitioned data sets are not interdependent - that is, they do not have boundary conditions that must be communicated to neighbouring processors.

The nCUBE library functions follow general rules for communication. These same rules also apply in complex applications whose communication routines cannot be fully automated:

* *Minimize the Communication-to-Computation Ratio* - Keep interprocessor communication time small, relative to computation time. One way to do this is with the proper choice of partitioning. Depending on the application, this may imply partitioning in which each partitioned part of the data or code involves a large amount of computation but the number of partitions is small (to minimize communication between partitions). It may also imply partitions which are simple in their geometry - forming whole objects or whole procedures rather than parts of objects or procedures. The Laplace equation example shown in the next section can achieve a communication-to-computation ratio of approximately 1% by making each data partition a submatrix of at least 30 x 30 data points. Rations considerably higher than

this can still achieve very efficient performance compared with conventional sequential processing, but lower ratios usually in better performance.

Keep Closely Related Objects on Neighbouring Processors - Messages passed between neighbouring processors move somewhat faster than messages between non-neighbouring processors. This is because neighbouring processors are directly connected by a dedicated communication link. nCUBE library functions can automatically map data onto processors so that neighbouring partitions are located in neighbouring processors.

4. PROGRAMMING EXAMPLE

4.1 Laplace Equation

This example solves the Laplace equation using a simple finite-difference algorithm. In the parallel version. the data matrix is partitioned into submatrices and loaded into the appropriate nCUBE processors by a library routine. This example is loosely synchronized by interprocessor communication at the boundaries between submatrices. Again, a library routine handles the communication.

There are, of course other ways to partition data. For example, the matrix could have been partitioned among the nCUBE processors one-dimensionally, as a stack of rows. Alternatively, the boundary dedicated only to that aspect of the calculations, while the remaining processors handle only interior points.

4.2 Sequential Version

The sequential version begins by declaring the parameter, 32, used in dimensioning the x and dx matrices for memory allocation. Then the *solve* subroutine is called and the program ends.

The *solve* subroutine dimensions and initialises its own working matrices, x and dx , and iteratively fills them with the finite-difference calculations. First the dx increments are calculated, then the results are used to update the x matrix.

For simplicity, the program ends by printing only one value - that of the grid point, (n/2+1,n/2), which is one grid point away from the centre of

the data matrix. A real application would output to a graphics plotting program or disk file.

```
c       Sample program to execute simple finite difference algorithm
c       SEQUENTIAL VERSION

c       Allocate memory
        parameter(n=32)
        dimension  x(n+2,n+2),dx(n,n)


c       Execute the algorithm:
        call  solve(x,dx,n)
        end

        subroutine  solve(x,dx,n)

        dimension  x(0:n+1,0:n+1),dx(0:n,0:n)
        alpha = 0.25
        do 30 iter=1,10

        do 10 j=1,n
        do 10 i=1,n
10      dx(i,j) = x(i-1,j) + x(i,j+1) - 4. * x(i,j) + x(i,j-1) + x(i+1,j)
        do 20 j=1,n
        do 20 i=1,n
              x(i,j) = x(i,j) + alpha * dx(i,j)
              if ( i .eq. n/2 .and. j .eq. n/2) then
                    x(i,j) = 255.
              endif
              if ( i .eq. n/2+1 .and. j .eq. n/2) then
                    check = x(i,j)
              endif
20      continue
30      continue
        write(*,*) 'check = ',check
        end
```

4.3 Parallel Version

In the parallel version, a new parameter, *m*, serves to dimension the *x* and *dx* matrices for memory allocation on each processor.

The *divsqu* library routine - which is specifically designed for square, two-dmensional regions - decides how to partition the *x* and *dx* matrices into submatrices and then loads these submatrices into the memories of the n processors. The *divsqu* routine also defines *ilo, ihi, jlo,* and *jhi* to be the boundaries of the data region in a particular processor. Then, the *solve* subroutine is called and the program ends.

The only changes to the *solve* subroutine are an include statement, the array indices, and the call to the *getedg* library routine. The include statement declares the *ilo, ihi, jlo,* and *jhi* array indices in a common block, *utln.hf*. The *getedg* library routine updates all of the boundary values for the local region by exchanging data between neighbouring processors in the two-dimensional mesh, as determined by the previous call to *divsqu*.

The program ends in the same way that the sequential version ends, except that a check is made for zero values. This ensures that only the processor that is computing the grid point, (n/2+1,n/2), prints its result.

```
c     Sample program to execute simple finite difference algorithm
c     PARALLEL VERSION

c     Allocate memory
            parameter  (n=32)
            parameter  ( m=32) .....added parameter
            dimension  x(m+2,m+2),  dx(m,m)

            call divsqu(n) .....library routine - partition data

c     Execute the algorithm :
            call  solve(x,dx,n)
            end
```

```
            subroutine  solve(x,dx,n)
$include(utln.hf) .....common variables
            dimensions  x(ilo-1:iho+1,jlo-1:jhi+1),  dx(ilo:ihi,jlo:jhi)
            alpha = 0.25
            do 30 iter=1,10

            call getedg(x(ilo,jlo)) ..... library routine - comm. data

            do 10 j=jlo,jhi
            do 10 i=ilo,ihi
10          dx(i,j) = x(i-1,j) + x(i,j+1) - 4. * x(i,j) + x(i,j-1) + x(i+1,j)
            do 20 j=jlo,jhi
            do 20 i=ilo,ihi
                  x(i,j) = x(i,j) + alpha * dx(i,j)
                  if ( i .eq. n/2 .and. j .eq. n/2) then
                        x(i,j) = 255.
                  endif
                  if ( i .eq. n/2+1 .and. j .eq. n/2) then
                        check = x(i,j)
                  endif
20          continue
30          continue
            write(*,*) 'check = ',check
            end
```