Using CMSSL with Split-C *

Steven Steinberg

December, 1994

1 Introduction

The difficulty of programming the CM-5 vector units is well known. Although a number of languages have been designed in an attempt to hide the complexity of the vector units (such as AC and CDPEAC), obtaining good performance with the VUs still requires an extremely detailed understanding of the hardware and an almost obsessive attention to data layout. The result has been that most programmers forgo using the vector units altogether.

Thinking Machines partially addressed this issue with the CM Scientific Subroutine Library (CMSSL), a library of mathematical routines such as matrix-multiply and Fast-Fourier Transform that have been carefully engineered to take full advantage of the CM-5 and its vector units. Unfortunately, the library is intended only for use with CMF and, to a lesser extent, C*, data-parallel languages with sophisticated run-time systems that support the notion of "parallel arrays" that live in vector space.

In this appendix I describe how CMSSL can be called from two different SPMD languages, C with message passing and Split-C. While conceptually a simple task, the details of getting CMSSL and non-data-parallel languages to cooperate proved to be difficult. I will describe some of the subtle problems but will primarily give a high-level overview of the method; those interested in the details can consult the source code of the examples. In section two of this appendix I sketch the architecture of the vector units and explain how arrays are mapped into their memory. Then, in section three I describe how to call nodal CMSSL and point out a few of its limitations. I conclude by drawing a few lessons and pointing toward future work that could be done.

2 VU Architecture

Each node of the CM-5 has six main components: the CPU (currently a SPARC processor), the network interface, and four vector units (figure 1). The vector units act as both memory controllers and floating-

*This report appeared as an appendix to the author's Master's report, titled, "Parallel Cell Simulation: Analysis, Abstraction and Portability," December 1994.

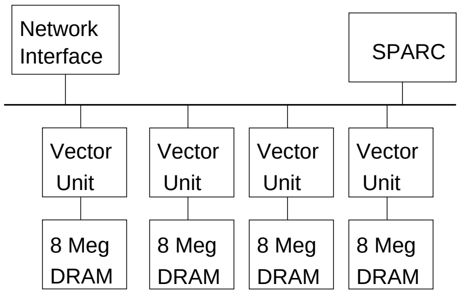

Figure 1: One node fo the CM-5

point units. This scheme offers very high memory bandwidth, about 500 MB/sec per node, but is also the cause for much of the difficulty inherent in programming the VUs. Because each VU can only address its associated memory unit, data layout becomes critical. Also, because each access is uncached, scalar performance of the VUs is poor. This leads to time-consuming swapping of data between scalar and vector memory.

Vector memory is mapped into virtual memory as shown in figure 2. A single physical address can be accessed by either its instruction address, which triggers the execution of an instruction in the associated vector unit, or by its data address. For now we will only be concerned with data addresses. The data address space can be broken down further: there are heap and stack segments (each grows upwards), and these in turn are broken down into regions for each VU and combination of VUs. The segments shaded gray can only be written to, this allows a write to be broadcast to multiple VUs with a single instruction.

Given this memory architecture, the problem now becomes how to best map an arbitrary array into vector space. In order to achieve peak performance we are faced with two main constraints: we must divide the array as evenly as possible so that each VU has an equal amount of data and we must minimize communication between VUs. The latter is necessary because data movement within a VU is faster than between VUs, which in turn is faster than data movement between nodes. The CM Run Time System (CMRTS) adds a further constraint: the size of the data owned by each VU must be a multiple of eight.

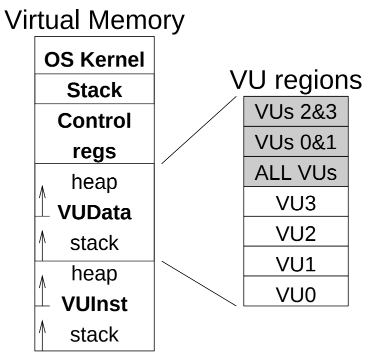

Figure 2: Vector memory

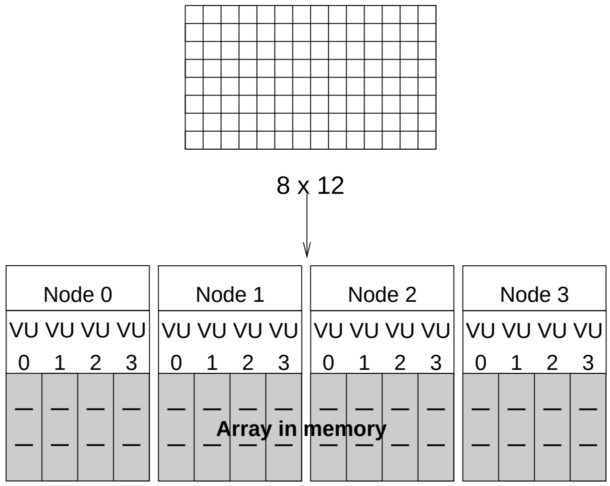

Figure 3: Mapping an array

Suppose we have an 8 x 12 array that we wish to map onto a machine with four nodes, as shown in figure 3. We must divide the array into sixteen equal subgrids, each with a size that is a multiple of eight. This is impossible to do for an 8 x 12 array so we must add garbage positions (array elements that do not contain useful data) to pad axis one to 16. The resulting array could then be laid out a number of ways, two of which are shown in figure 4. Notice that in the right example axis 0 is serial — all positions along the axis are located on the same VU — so communication will be fastest along this axis.

Another layout decision must be made concerning the numbering of the vector units. In figure 4 the vector units could be numbered so that the horizontal axis varies fastest (so node 0 would own the first top-four subgrids), or so the vertical axis varies fastest (node 0 would own the left-most four subgrids). CMRTS will attempt to make these layout decisions optimally, but to achieve peak performance with CMSSL the programmer must be prepared to control the layout decisions explicitly. Fortunately, this can be easily done and is described in the following section.

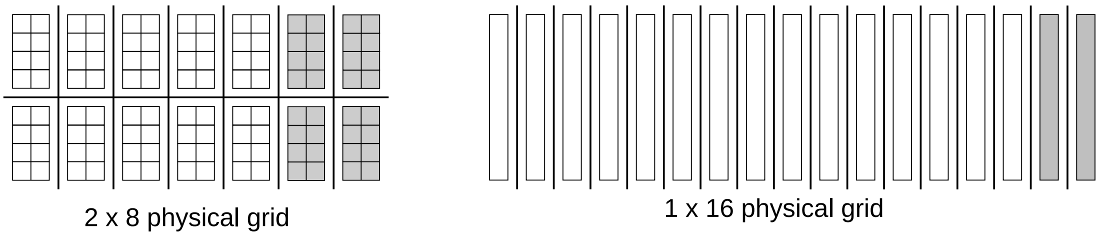

Figure 4: Mapping an array

3 Nodal CMSSL

Nodal CMSSL provides a model slightly different than the one I have described. Operation still follows a data-parallel model, but now each node acts as the host while the vector units are the PEs. This allows each node to operate independently which more closely resembles the SPMD model of Split-C and C with message passing. A nodal CMSSL routine is called with arrays that live entirely in the associated four vector units. For some applications, where each node operates independently, this model works very well. For other applications, where the size of the data is large enough so that it makes sense to use the power of the entire machine, this model is less appropriate. But in some cases, such as matrix-multiply or 2-D FFT's, it is possible to use blocking to effectively perform global operations with nodal CMSSL. We have used this last method in a Navier-Stokes solver.

3.1 Calling nodal CMSSL

There are three main steps to using nodal CMSSL from C: allocation of arrays, manipulating those arrays, and then actually calling the CMSSL routine. Allocation of vector arrays is done using the primitives provided by the CMRT library. After a few required steps to initialize the run-time system, a geometry that describes the desired layout of the array is created and then passed to the allocation routine. This routine returns a pointer, of type `CMRT_desc_t`, that completely describes the layout and position of the array. A few of the fields in this descriptor (`geometry_order_offsets_ptr`, `axes_layout_maps_ptr`, and `axes_extents_ptr`) must be modified so that it can be understood by CMSSL. I have implemented a library of routines that allows allocation of parallel arrays to be done from C while hiding most of details. These routines support a range of control over the array geometry, from basic to very detailed.

Reading and writing parallel arrays is awkward for two reasons: first, determining the address for an element depends on the layout and can be complex to calculate. Second, because vector memory is uncached, each access is two to four times more expensive than a normal memory access. Therefore it is best to read and write in bulk using the CMRT routines:

```
CMRT_read_array( CMRT_desc_t dest, double *src )
CMRT_write_array( CMRT_desc_t dest, double *src )
```

These allow an entire array to be copied out of or into vector memory.

Finally the CMSSL routine may be called, but the programmer must be careful to use FORTRAN calling conventions. This means that an underscore is appended to the function name, all parameters must be passed by reference, arrays must be passed in column-major order, and a length must be passed following any string. The following shows how the FFT routine is called:

```
fft_(a_desc, "CTOC", &type, &fftsetup, &ierr, 4);
```

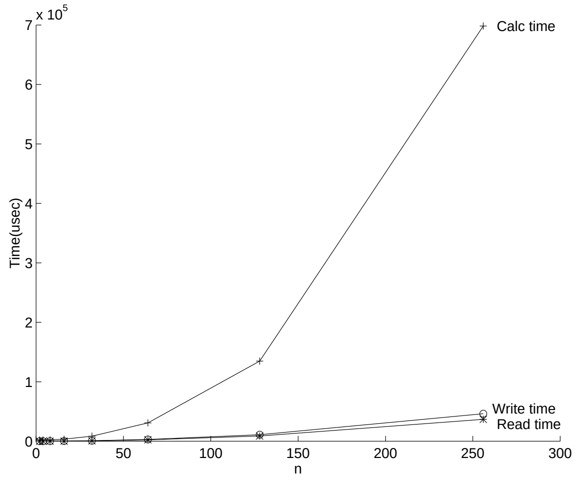

Figure 5: Runnin time for CMSSL matrix multiply

3.2 Performance of nodal CMSSL

Nodal CMSSL offers the same performance whether called from CMF or Split-C, the only difference lies in the overhead costs of manipulating vector memory. This overhead is implicit and unavoidable in CMF but is explicit when CMSSL is called from Split-C. The magnitude of the overhead determines when it makes sense to call CMSSL.

Figure 5 and table 1 show the break down of time spent manipulating vector memory and time spent performing a N x N matrix multiply. It is important to realize that these times are averages over many iterations — the first occurrence will be significantly more expensive due to initialization and set-up time.

The lesson from these numbers is clear: manipulating vector memory is a significant cost and should be minimized. We can think of vector memory in terms of a new level in the memory hierarchy, and the same rules that apply to optimizing programs to extract maximum cache reuse apply here as well.

One use of nodal CMSSL where the time spent performing memory manipulation is particularly important is when nodal CMSSL is used for global operations. This is done using *blocking*, an array is split into blocks that are then transferred between nodes using message passing. I implemented blocked

N	Write (μsec)	Calculate (μsec)	Read (μsec)	% overhead
2	33	2024	42	3.57%
4	48	2135	65	5.03%
8	85	2342	126	8.26%
16	211	3524	302	12.71%
32	671	8468	877	15.46%
64	2243	30763	2942	14.42%
128	8814	134791	10912	12.77%
256	36723	698328	46262	10.62%

Table 1: Breakdown of time for CMSSL matrix multiply

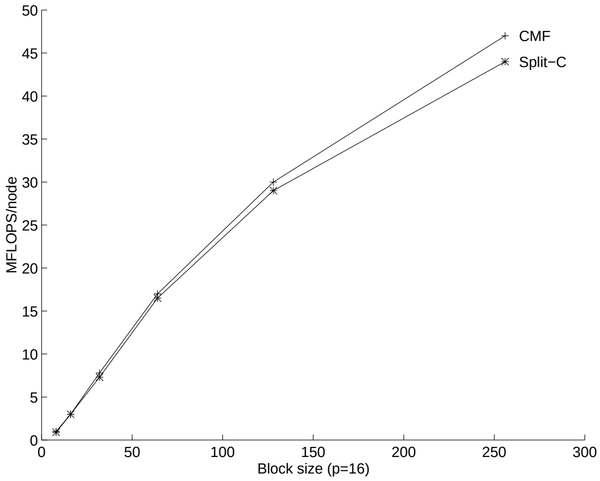

Figure 6: Comparison between CMSSL called from CMF and Split-C

matrix multiply using nodal CMSSL and initially found it to be fairly inefficient. The problem was that CMMD is extremely slow at transferring portions of vector arrays (it first creates a new array, copies the original portion in, and then sends the data.) However, Jean-Marc Adamo modified the code so that the lower-level CMAML library was used and this dramatically improved running time.

Figure 6 compares the speed of this blocked CMSSL matrix multiply against the global CMSSL routine called from CMF and shows that the global CMSSL version is faster by only about 10%.

4 Conclusion

This appendix has described how to call CMSSL from Split-C, as well as showing why *and when* it is a good idea. Using CMSSL is an easy way to harness the power of the vector units and thereby achieve significantly higher floating-point performance on common mathematical routines, from matrix-multiply to LU to FFT.

Nodal CMSSL, where each node operates independently, can be used quite easily with the help of the library I have built. There are some initial costs, but for problems of moderate size calling CMSSL should result in far higher performance than could be achieved with the SPARC CPU alone, even with hand tuned assembly. However, the programmer must be aware of the expense of transferring data between vector and scalar memory.

Further work could be done on making CMSSL even easier to use from C and Split-C. Wrappers could be written for each function that hide the details of FORTRAN calling conventions. And, ideally, Split-C should be able to call global CMSSL as well (although TMC claims this is impossible due to memory policies). But in general, I believe this work marks a significant advance toward routinely attaining super-computer performance from Split-C on the CM-5.