user. This means that the programmer does not need to worry about the communication and synchronization control between different nodes, but it is required to rewrite the program to use special mechanisms of the C* language. Another advantage of the C* is the access to the vector facilities of the nodes. The languages used in the MP method does not allow the use of vector facilities.

5. The Data Parallel Implementation

The \textit{for} loop described below was written in C* and it gives an idea of the implementation process in this language.

\begin{verbatim}
shape [2048] shapeA;
shapeA int: ptAUX_SP[1000];
shapeA float: ptGRID_SP[1000]
for (ii = 0; ii < max_grid_dimen; ii++)
{    where (ptGRID_SP[ii] >= INVALID_VALUE) ptAUX_SP[ii] = 0x1001;
    else where (ptGRID_SP[ii] >= isovalue) ptAUX_SP[ii] = 1;
    else
        ptAUX_SP[ii] = 0;
}
\end{verbatim}

Considering a 32 nodes configuration, \textit{shapeA} is a data type that will distribute 2048 components in this way: [0-63] in the node 0, [64-127] in the node 1, and sequentially following this scheme until the components [1984-2047] are mapped in the node 31.

\textit{ptAUX_SP} and \textit{ptGRID_SP} are respectively an integer and a real vector. Each element of these vectors (\textit{ptAUX_SP[ii]} or \textit{ptGRID_SP[ii]}) has 2048 components. In this case we use the term \textit{element} rather than component in order to avoid confusions between vector elements and shape components.

The \textit{where} command works similarly to the \textit{if} command in a sequential program. It defines a selection condition for a shape component as it is exemplified below.

\begin{verbatim}
ii = n, 0 <= n <= 1000, 0 <= j <= 2047
if [j]ptGRID[ii] equal to INVALID_VALUE then [j]ptAUX_SP[ii] = 0x1001
else if [j]ptGRID[ii] equal to isovalue then [j]ptAUX_SP[ii] = 1
else [j]ptAUX_SP[ii] = 0
\end{verbatim}

Program writing using \textit{where} is not a difficult job, we encountered a serious performance problem using \textit{where} statements. Despite the importance of the \textit{where} command, Thinking Machines informed us that this problem would be solved only in the next version of operating system and unfortunately the available guide of the language did not have this information. We only discovered the reason for the poor performance when another guide became available [CM92]. The reasons for the low performance of the \textit{where} command are explained in this guide and reproduced below.

a) C* blocks of code are loaded in each node of the machine and all nodes are executing the same code synchronously;
b) at the end of the processing of each block of code, all nodes are synchronized so that the results of the block can be used for the next block to be loaded;
c) when a \textit{where} command is found by the used compiler, a new block is generated because the compiler can not distinguish situations that requires or not a new block;
d) so, if a program has a great number of \textit{where} commands it will have a great number of blocks and consequentially a great number of synchronization operations that will decrease the performance of the program.

To solve this problem, the new guide recommended the use of logical variables or expressions rather than \textit{where} commands. Using this approach, the loop described previously changes to:
for (ii = 0; ii < max_grid_dimen; ii++)
    ptAUX_SP[ii] = 0 + (ptGRID_SP[ii] >= isovalue) +
        (ptGRID_SP[ii] == INVALID_VALUE)*0x1000

In this case, the logical expression ptGRID_SP[ii] >= isovalue works like a logical parallel variable. Considering temp_SP the name of this variable, then:

    [j]temp_SP will be "1" if "[j]ptGRID[i] >= isovalue"
    [j]temp_SP will be "0" if "[j]ptGRID[i] < isovalue"

The second loop version is more efficient than the first one. Optimizing all the program using this method it was possible to increase the performance of the algorithm routines in the following rates: Flags 5.78, Marching 1.41, Normals 3.25 and Triangles 1.91. The routines Flags and Normals, with larger percentages of vectorized loops, yielded larger rates of optimization.

Another important factor for efficiency is the size of the subgrid shape that is mapped to each node or vector processor. When the machine has vector facilities, each node is equipped with 8 vector processors. A minimum of 8 components of the shape type is mapped in each vector processor. Then, in a configuration of 32 nodes, the mapping is made in the following way:

| shape [2048] shapeA
2048 64 components/node in a 32 nodes machine configuration
64 components of type ShapeA mapped in each node
8 vector processors per node
8 components per vector processor |

<table>
<thead>
<tr>
<th>Routine</th>
<th>CM-5 (seconds)</th>
<th>SG Onyx (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flags</td>
<td>0.17</td>
<td>0.21</td>
</tr>
<tr>
<td>Marching</td>
<td>1.34</td>
<td>0.17</td>
</tr>
<tr>
<td>Normals</td>
<td>0.46</td>
<td>0.19</td>
</tr>
<tr>
<td>Triangles</td>
<td>1.30</td>
<td>0.07</td>
</tr>
</tbody>
</table>

The C* compiler always maps a number of shape type components that is a multiple of 8 to each vector processor. Then, if a shape type is defined with 2050 components, 4096 components will be mapped in the vector processors. The extra components do not cause errors in the output but, as the mapping of the components changes, do have an impact on the program performance. This is especially true in more complex cases such as shape types with two dimensions (example: shape [8][12] shapeA).

Unfortunately, even after the conclusion of all mentioned optimizations, the global performance was still unsatisfactory. Table 1 has a comparison between the time spent to run the routines on a CM-5 with 32 processors and vector facilities, and on a Silicon Graphics Incorporated (SGI) Onyx. These runs used a data grid with dimension 50x50x50 with a spherical isosurface.

The two implementations have a few differences in terms of algorithm process. The main difference is the calculation of normals and the elimination of polygons in the extra border in the CM-5 process. As the SGI Onyx does not have a vector processor, the processing time of the highly vectorized routine Flags is worse than the CM-5 time. On the other hand, the other routines run much faster on the SGI Onyx.

Some reasons for the low performance of the CM-5 implementation are:

a) The CM-5 C* compiler had many problems associated with code optimization, so any implementation on this machine requires many tests to check the performance of code sections. These problems were detected only in the end of the implementation.

b) The CM-5 implementation of the algorithm was based in the assumption that “each component of a shape type processed in a different node would be related to a different Dsubgrid”. For example, in the case of a 32 node configuration, a shape type would have 32 components and consequently it would have 32 Dsubgrids. But if the machine has vector processors, the mapping of the components in the processors must obey the rule of multiples of 8 and then the number of Dsubgrids became 2048.