Exploring a Novel Gathering Method for Finite Element Codes on the Cell/B.E. Architecture

Mohammad Jowkar, Raúl de la Cruz and José M. Cela
Department of Computer Applications in Science & Engineering
Barcelona Supercomputing Center
Barcelona, Spain
{mohammad.jowkar, raul.delacruz, josem.cela}@bsc.es

Abstract—Indirect addressing is known for being slow on conventional architectures, due to the extra step of gathering data before computations can be done. There have been proposed many methods for optimizing indirect addressing. However, these almost exclusively, merely try to change the order in which data is accessed, so as to better utilize the cache. Furthermore, vector instructions can not be used, since data is not accessed continuously, and therefore valuable processing power can not be exploited.

The Cell/B.E. architecture has multiple powerful DMA engines, suitable for gathering scattered data. Unfortunately, at fine data granularity, they have several constraints which make them inefficient. In this paper, a novel solution called DMA list Interlacing (DLI) is explored, which overcomes the DMA constraints and enables the usage of vector instructions, without any extra effort. It is shown that DLI can achieve speedups of several orders of magnitude, compared to conventional processors.

Index Terms—Cell/B.E., Cell Broadband Engine Architecture, DMA List Interlacing, DLI, DMA Transfer, DMA Engine, Indirect Addressing, Scattered Data, Sparse Data, Sparse Gather, Finite Element.

I. INTRODUCTION

The cornerstone of many scientific codes and algorithms is indirect addressing. Instead of accessing an array directly using an index, indirect addressing uses an auxiliary array which contains the indexes to be used to access the main array. Indirect addressing is often used when certain data is shared among several data structures. A typical application area where indirect addressing is used extensively, is in Finite Element (FE) codes.

The disadvantage with indirect addressing is, however, that the order in which data is accessed, is not predictable for traditional caches and this usually incurs a penalty. Furthermore, since data is not accessed continuously, it is not possible to use SIMD (Single Instruction, Multiple Data) instructions to achieve data parallelism.

There have been many attempts to solve this performance bottleneck in the past. Most common methods include reordering of the auxiliary array to reduce cache misses or elimination of the auxiliary array altogether by using data duplication. Although such methods can give a significant boost in performance, they are still quite inefficient, due to the inherent cache based architecture of traditional processors.

There have now appeared, however, processor architecture types which diverge from the above traditional cache based architectures. The most unconventional of these is perhaps the heterogeneous Cell/B.E. architecture [1], with cores without any caches. The burden of predicting data access patterns is instead put on the programmer. This can be used to ones advantage in cases such as indirect addressing.

In this paper, a novel method for doing indirect addressing on the Cell/B.E. architecture is presented, which is as efficient as doing direct sequential addressing. An additional beneficial side effect of this method is that data parallelism can be achieved, without any extra effort or data duplication. The method is in this paper dubbed DMA List Interlacing (DLI). Cell/B.E. has been used many times in the past, to show that very high bandwidths can be achieved by using DMA transfers [2], [3], [4], [5].

II. RELATED WORK

For many non-structured mesh based applications, like those based on FE methods, the main data structures can be efficiently represented by a graph, where nodes and edges represent data elements and interactions between the data elements respectively.

FE based applications are characterized by patterns of data and computation that are unknown until run time, and therefore data access often has poor spatial and temporal locality, which leads to inefficient use of the memory hierarchy. Improving memory system performance for such kinds of applications, requires addressing problems of both latency and bandwidth.

The two main techniques for achieving this is data and computation reordering, shown in Fig. 1, which will be covered briefly in the following subsections.

A. Data reordering

Data reordering involves changing the memory location of data elements, but not the order in which these elements are referenced, by placing data element memory addresses near each other. This has the effect that data elements are referenced together, and thus data reordering improves spatial locality. Temporal locality would not be affected, since the order in which data elements are accessed, remains unchanged during the computation. Some data reordering techniques that can be found in the literature are, Space-filling curves [6], [7], [8].
The Finite Element Method (FEM) will briefly be covered here since this was the application area where the DLI method was developed for initially.

Finite Element (FE) is a numerical method used to approximate the solution of Partial Differential Equations (PDEs) that arise in engineering and scientific contexts. The method is based on the integration of the terms in the equation to be solved, instead of point discretization schemes like the Finite Difference (FD) method. FE is used to solve a wide range of problems, and permits physical domains to be modeled directly using unstructured meshes, composed of geometrical elements like triangles or quadrilaterals in 2-D, and tetrahedrals or hexahedrals in 3-D.

FEM works by discretizing a continuous domain, in so called elements, to form a mesh. The domain can represent anything from a car to fluids in a tube. An element in turn approximates a very small subset of the total domain. Each element is composed of multiple nodes, called nodal points, and each node has several attributes representing physical properties, attached to it. This could be pressure, density, temperature, etc. Finally since each element can have one or more neighboring elements, nodes are commonly shared between multiple elements.

Fig. 2 shows a simple mesh composed of only four elements, the associated data structures, and the process of gathering the nodes of one of the elements. Similarly, the pseudo code below shows how the above data structures can be accessed, so as to go through each element, gather the associated nodes using indirect addressing and performing a computation on the nodes.

```
for(int m=0; m < nr_elements; ++m)
{
    // Gather nodal values for element m
    // in local arrays using indirect addr
    for(int n=0; n < nr_element_nodes; ++n)
    {
        // Physical properties of node n
        int idx = connectivities[m][n];
        pres[n] = pressure[idx];
        dens[n] = density[idx];
        temp[n] = temperature[idx];
    }

    // Do computation on gathered nodal values
    compute(pres, dens, temp, ...)
}
```

The algorithm below shows the main operations of a Finite Element code.
Algorithm 1: General control flow of a generic Finite Element code. The element loop involves data intensive (D) and compute intensive (C) operations.

1: Domain decomposition of the global mesh
2: for time = 0 to time_end do
3:   for ielem = 1 to nelem do
4:      Gather element node variables (D)
5:      Isoparametric and Cartesian derivatives (C)
6:      Compute variables at Gauss points (C)
7:      Build $A_{ielem}$ matrix and $RHS_{ielem}$ (C)
8:      Apply boundary conditions (C)
9:      Assembly into global $A$ matrix and $b$ (D)
10: end for
11: Solver (Implicit / Explicit) $Ax = b$
12: Write output (each n steps)
13: end for

In general, the two most time consuming kernels in FE codes are the assembly of the element matrix into the algebraic system ($A$ & $b$) (tasks 4 to 9 in Algorithm 1) and the execution of the solver (task 11). This paper is about task 4 of Algorithm 1, which is the gathering of node values.

Although this paper uses FE to show how an efficient gathering can be done on the Cell/B.E. architecture, most of the discussion can be related to other methods, which also have irregular data access patterns, such as Finite Volume, Multigrid or Particle-In-Cell.

IV. INTRODUCTION TO THE CELL/B.E.

A brief introduction will be given here about the Cell Broadband Engine (Cell/B.E.) processor, since it is necessary to know the Cell/B.E. architecture, to understand the DLI method presented in this paper.

Cell/B.E. is a heterogeneous architecture with two types of cores, each specialized for certain workloads. Fig. 3 gives an overview of the major elements in the Cell/B.E. processor. The two processing cores are the PowerPC Processor Element (PPE) and the Synergistic Processors Elements (SPEs). The PPE is a traditional cache hierarchical core, suitable for context switching, logical codes etc., whereas the SPEs are SIMD optimized cores without caches, suitable for crunching numbers. Instead of caches, the SPEs have software based

scratchpad memory, called Local Store (LS), which is the only memory the SPEs can access directly.

In addition the SPEs also have dedicated DMA controllers for transferring data to the LS asynchronously. These DMA controllers can be used to overlap data transfers with computations, and can therefore potentially completely eliminate the memory latencies normally associated with traditional cores. This can be achieved by using multi buffering techniques. The DMA controller supports an additional useful feature, called DMA lists transfers. Each DMA list element consists of an address and a transfer size. The DMA controller can read the DMA list and transfer each element in the list and store them continuously in the SPE LS. Fig. 4 shows a DMA list and the corresponding gather. This feature enables the gathering and scattering of data, but as will be shown later, it is not immediately applicable to the general FE case, due to the many constraints imposed by the Cell/B.E. architecture.

V. A NOVEL GATHERING METHOD USING DMA LIST INTERLACING (DLI)

As mentioned in the previous section, DMA lists are used to gather and scatter data. Since FE nodes are scattered, DMA lists seem like a good approach to gathering element nodes. However, the Cell/B.E. processor has several constraints which limit the usefulness of DMA lists to very specific cases. This is especially true when the DMA list transfers are less than 16 bytes. This section will cover these constraints and will show how to overcome them so as to implement an efficient gathering of data for FE codes. It is assumed throughout this paper, that the FE node data is double precision and thus 8 bytes. Also to simplify the explanation of methods and figures in this paper, it will be assumed that we only have elements of one type and that each element only has four nodes. It is trivial to extend the ideas presented here to single precision and to other element types.

A. GATHERING ELEMENT NODES USING DMA TRANSFERS

The aim of this paper is to use the DMA list feature of the Cell/B.E. processor, to gather scattered element nodes into the SPE local stores, in a way so the SPEs can use their SIMD units to do vector computations.
Ideally what we want is a DMA list, which gathers two elements at a time, so that they are stored in mixed AOS (Array of Structures) and SOA (Structure of Arrays) form as shown here:

\[
\{ [n1_{ei},n1_{ej}], [n2_{ei},n2_{ej}], [n3_{ei},n3_{ej}], [n4_{ei},n4_{ej}] \} \tag{1}
\]

where \( n_{X_{ey}} \) is node \( X \) of element \( Y \).

The above layout would allow us to vectorize the FE computations very easily. To achieve such a layout would mean that we create a DMA list such that the first DMA list entry contains the address of the first node of element \( ei \), the second entry the address of the first node of element \( ej \), the third entry the address of the second node of element \( ei \) and so on, as shown in Fig. 5. However, achieving the layout in equation 1 above, using DMA lists, is more difficult than might seem at first, due to the many constraints imposed by the Cell/B.E. processor.

All Cell/B.E. DMA transfers must meet the following constraints:

- **L1** The transfer size can only be 1, 2, 4, 8 or 16 bytes, or be a multiple of 16 bytes to a maximum of 16 KB.
- **L2** Transfers less than 16 bytes must be naturally aligned\(^1\).
- **L3** Transfers bigger than 16 bytes must be 16-byte aligned.
- **L4** Peak bandwidth is achieved when source and destination addresses are 128-byte aligned and transfer size a multiple of 128 bytes.

If all of the above constraints are satisfied the DMA controller can write 128-bytes to the LS every 8 cycles. At 3.2 GHz this equals 51.2 GB/s per SPE.

In addition to the above constraints, there are two additional constraints for DMA list transfers:

- **L5** Transfers must end at 16-byte aligned LS addresses, otherwise the LS address is incremented automatically to satisfy this, before starting the next transfer in the DMA list.

\(^1\)Address must be divisible by transfer size.

L6 Transfers must be aligned within a 16-byte boundary.

Since each individual FE node data is 8 bytes and DMA list transfers will be used to gather the node data, all of the above six constraints apply. The constraints which pose the greatest challenge, however, are L5 and L6.

The L5 constraint means that if we create a DMA list like in Fig. 5, with 8 byte transfers, we will get gaps in the LS. This could be acceptable since we could have two DMA list transfers: an initial one and then a second one to fill the gaps. However, due to the L6 constraint the gaps depend on the indirect addressing of each element, and are therefore not consistent.

Fig. 6 shows an example of how a gather of data can occur in the LS, due to the above constraints, when using DMA lists. The next two sections will cover how this variability in the gaps can be overcome, by using a method dubbed in this paper *DMA List Interlacing* (DLI). The solution is specific for FEM codes but can potentially be extended to other types of codes, with some modifications.

### B. Element Alignment Type

Recall that it was not possible to use a single DMA list, to transfer the nodes of two different elements in the local store, as in equation 1. The reason for this was that the L5 constraint results in gaps in the LS, since the individual DMA list transfers are less than 16 bytes. It was also not immediately possible to split the DMA list up in two separate DMA lists, so that the first transfers the nodes of one element, and the second fills the gaps of the first DMA transfer with the nodes of a second element. This was because the indirect addressing and the L6 constraint, made it inconsistent where element nodes were written, thus resulting in LS gaps, as shown in Fig. 6.

Luckily, it is possible to predict such LS gaps, before issuing a DMA list transfer. The idea is to split elements up in types,
The initial problem posed with gathering FE nodes was that given a random element, it was not apparent how its nodes would be gathered in the local store. However, since the alignment type of each element can now be determined, this unpredictability can be overcome. We can as before create two separate DMA lists, however this time with compatible element types, say $1010_2$ and $0101_2$, respectively. Issuing these two DMA transfers concurrently to the same starting address in the LS, will result in perfect overlapping/interlacing of two elements. This data can then be used for computation in SIMD fashion. Fig. 8 shows an example of interlacing an element, compatible with the element from Fig. 6.

**D. Strategy**

Using the two previous concepts of element alignment type and DMA List Interlacing, the following strategy can be used to gather scattered element nodes by using all eight SPEs:

1) Go through each entry of the connectivity table and assign each element a type, based on the alignment of its nodes in main memory.
2) Split elements up in buckets depending on their element alignment type.
3) Assign two compatible bucket subsets to each SPE.
4) For each iteration of the element loop, each SPE selects an even amount of elements from each of its assigned buckets.
5) Create two DMA lists, one for each bucket.
6) Use DLI to gather elements of compatible types.
7) The gathered element types will be continuous in the LS and can be computed using SIMD.

In the above, two buckets are compatible if the elements they contain are of compatible types. The number of buckets will be two to the power of the number of element nodes.

One possible concern with the above strategy is the relative sizes of the buckets. It is assumed that given a bucket, there will be enough elements in another compatible bucket, so that an SPE can use DLI to interlace all elements in the buckets. However, if two compatible buckets are not the same size, there will be elements which an SPE can not interlace. These are called bucket residuals and need to be computed by the PPE instead. Since the PPE is very slow at computing, it can become a bottleneck, if the total residual of the compatible buckets are too big. However, as we will see later, this poses no problem.

**VI. ANALYSIS OF THE DLI METHOD**

It is interesting to analyze the performance of DLI using a micro-benchmark, to see how much time is being spent on the different stages of the method and how well it scales. The main stages of DLI are (in the order they are done):

- **Assign each node a type based on its alignment**
  - Type of $e_i$ is $1100_2 = 12$

**Fig. 7.** The process of assigning an element an alignment type. A node is assigned a 1 if it is 16-byte aligned and 0 otherwise. In this example the data in the memory is assumed to be double precision (8 bytes).
1) DMA transfer of the connectivity table
2) Creation of the two DMA lists to be used for DLI
3) DMA list transfer gathering of scattered data using the DLI lists
4) Computation

Recall that the DMA controller runs independently of the SPE pipelines. This means that DMA transfers are asynchronous and hence computations can be overlapped with data transfers. Therefore the best performance is achieved when the time spent on doing DMA transfers (steps 1 and 3), are equal to the time for creating the DLI lists and doing computations (steps 2 and 4). Thus to test the efficiency of DLI, a generic computation needs to be defined for step 4 above.

The generic computation used in this paper is the following triple loop:

```c
double res = 0;
for (int m=0; m < nr_elements; ++m)
{
    for (int n=0; n < nr_element_nodes; ++n)
    {
        for (int q=0; q < nr_iterations; ++q)
        {
            res += node_data[m][n] * multiply_data[q];
        }
    }
}
```

The node_data array is the DLI gathered data and its type is vector double, which means that each element of the array is a vector containing two nodes from two different elements. This is a result of the DLI gathering. So in fact the array contains 2 \cdot nr_elements \cdot nr_element_nodes. Notice that the indirect addressing from Section III is no longer needed. multiply_data contains random data and is merely used to increase the computational intensity.

The two above outer loops go through each node of each element. The inner loop does a multiplication and accumulates the result. This is the basic unit of work. The end result is the accumulation of all node values. If needed, this work can be done more compute intensive by repeating the inner loop several times by increasing nr_iterations.

With the computation defined, it is possible to profile DLI. For this purpose Paraver [24] has been used, which is a performance visualization and analysis tool. For the profiling, a real world FE dataset of blood vessels in a brain, consisting of \sim 19 million elements with \sim 4 million nodes was used. The dataset is shown in Fig. 9. The DLI method was used to gather all mesh nodes and to perform the above computation. The value of nr_iterations was set to 1 to start with. Fig. 10 shows graphs obtained from a Paraver trace of such a run. As can be seen the most time consuming parts of the DLI method are the DLI gathering (step 3) and the computation itself (step 4), taking up 45% and 39% of the total execution time, respectively. The computational load is thus not enough to overlap step 3. Step 1, however, is insignificant since it is overlapped with the creation of the DLI lists (step 2), which takes 12% of the execution time. Therefore one way of improving the performance even further, would be to increase the amount of computations. This will not decrease the execution time, but it will remain constant, meaning that more work can be done for the same amount of time.

Fig. 10 also shows a trace of a run with nr_iterations set to 80. The execution time remains the same, however, the relative execution times have changed. Now the DLI gathering time has been reduced to 7% and the computation time has increased to 79%, effectively overlapping most of the memory transfer time.

The graph in Fig. 11 shows the execution time as nr_iterations is increased. Until a value of approximately 90, the execution time remains more or less constant. However, after that, the execution time begins to increase linearly. This is approximately the point where the DLI gathering time is equal to the computation time. This is consistent with the graph trace from before. At around 80 iterations, the DLI transfer is almost completely overlapped with computation.

A. Bucket Residuals

In section V-D, there was a concern that the bucket residuals could be big and since the PPE has to compute these, it could become a bottleneck. However, for all datasets which were used in this paper, the residuals were very small compared to...
the dataset sizes. Below is a table showing the bucket residuals, for the 19 million element brain dataset, for each compatible bucket pair.

<table>
<thead>
<tr>
<th>Bucket pair</th>
<th>Residual</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-15</td>
<td>135</td>
</tr>
<tr>
<td>1-14</td>
<td>33</td>
</tr>
<tr>
<td>2-13</td>
<td>604</td>
</tr>
<tr>
<td>3-12</td>
<td>43</td>
</tr>
<tr>
<td>4-11</td>
<td>205</td>
</tr>
<tr>
<td>5-10</td>
<td>782</td>
</tr>
<tr>
<td>6-9</td>
<td>155</td>
</tr>
<tr>
<td>7-8</td>
<td>246</td>
</tr>
<tr>
<td><strong>Total residual</strong></td>
<td><strong>2203</strong></td>
</tr>
</tbody>
</table>

One can, however, create pathological examples where the node distribution in the buckets are uneven. The worst scenario is when all elements are placed in a single bucket. This can only happen in cases where nodes are not stored continuously in memory or if e.g. only every fourth node is referenced in the connectivity table and so it is not realistic. Another pathological example is a mesh with a single center node, which is shared by many elements (like a pie cut in slices). This will result in a bucket distribution, where none of the buckets are compatible. However, such meshes are not realistic, and although they can happen locally, if the number of nodes is large in a general mesh, such examples occur several times and most will cancel each other out. In the worst case, nodes can be reordered specifically to achieve a small residual.

With the above in mind, the next section will explore the performance of DLI on the Cell/B.E., compared to indirect addressing on several different processor architectures commonly used in HPC systems.

VII. PERFORMANCE COMPARISON OF DLI

This section compares the performance of the DLI gather method on the Cell/B.E. processor, with the performance of the traditional indirect addressing gather method, from section III, on several conventional processors, commonly used in HPC systems (shown in Table I). The performance results in this section are for the micro-benchmark defined in section VI and are for a single processor chip only.

The table below shows the main datasets used to do performance runs. The datasets were all reordered to achieve the best performance on the architectures they were run on. Reordering methods have no impact on the performance of the DLI method whatsoever.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Number of elements</th>
<th>Number of nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brain (realistic)</td>
<td>18,999,594</td>
<td>3,807,528</td>
</tr>
<tr>
<td>Building (realistic)</td>
<td>4,724,778</td>
<td>833,731</td>
</tr>
<tr>
<td>Mold filling (realistic)</td>
<td>3,543,706</td>
<td>706,945</td>
</tr>
<tr>
<td>Cube</td>
<td>user variable</td>
<td>user variable</td>
</tr>
<tr>
<td>Spheroid</td>
<td>user variable</td>
<td>user variable</td>
</tr>
</tbody>
</table>

The execution times in this section are for the brain dataset, since it was used for the analysis in section VI and contains the biggest mesh, however, the relative results are similar for the other datasets listed above.

Fig. 12 shows the performance of Cell/B.E. plus DLI, compared to the other four processor architectures in Table I, running the traditional gather method, which was parallelized using OpenMP. As can be seen, the DLI method on Cell/B.E. is slower than all other architectures for low number of iterations, however, it quickly becomes faster as the number of iterations are increased. DLI begins to become faster at around 45 iterations, thanks to its initial constant execution time, as discussed in section VI. The overall closest competitor is the Nehalem architecture. A more close-up comparison of the Nehalem and Cell/B.E. is shown in Fig. 13.

The table below shows the execution times for \( nr\_iterations = 1 \), and the speedups of DLI compared to the traditional gathering method on other processors:
Fig. 13. Close-up performance comparison between the Nehalem and Cell/B.E. This is the same run as in Fig. 12.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Time (sec)</th>
<th>DLI speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell/B.E. DLI</td>
<td>0.44</td>
<td>1</td>
</tr>
<tr>
<td>Power6</td>
<td>0.06</td>
<td>0.14</td>
</tr>
<tr>
<td>Nehalem</td>
<td>0.07</td>
<td>0.16</td>
</tr>
<tr>
<td>Opteron AMD64</td>
<td>0.08</td>
<td>0.18</td>
</tr>
<tr>
<td>BlueGene/P</td>
<td>0.23</td>
<td>0.52</td>
</tr>
</tbody>
</table>

The above table is the worse case scenario for Cell/B.E. since there is a lot of wasted bandwidth in the EIB. Furthermore, there is very little computation to overlap the memory transfers. In other words the SPEs are stalling for data. However, the computational intensity of $nr\_iterations = 1$ is very unrealistic for FE codes, which tend to be much higher. As shown in Fig. 12 and 13, better performance is possible when $nr\_iterations$ is increased.

The table below shows the execution times for $nr\_iterations = 90$, which was the limit where computation and transfer time was found to be more or less equal, in the previous section.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Time (sec)</th>
<th>DLI speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell/B.E. DLI</td>
<td>0.44</td>
<td>1</td>
</tr>
<tr>
<td>Power6</td>
<td>2.17</td>
<td>4.93</td>
</tr>
<tr>
<td>Nehalem</td>
<td>0.91</td>
<td>2.06</td>
</tr>
<tr>
<td>Opteron AMD64</td>
<td>2.76</td>
<td>6.27</td>
</tr>
<tr>
<td>BlueGene/P</td>
<td>1.85</td>
<td>4.20</td>
</tr>
</tbody>
</table>

As can be seen much better speedups can be achieved when the computational intensity is increased. The speedups can be improved further for even higher values of $nr\_iterations$. Thus at $nr\_iterations = 500$, Cell/B.E. is 3.51x faster than Nehalem.

Although the results in this section show that the DLI method on Cell/B.E. can outperform the other architectures, the results are nevertheless only theoretical, since no useful computation is done. Therefore it would be practical to also verify the performance of DLI in a real application. The next section will thus cover the usage of the DLI method to improve Alya, which is a real FE application.

VIII. ALYA, A HPCM CODE

Alya [26], [27] is a High Performance Computational Mechanics (HPCM) code based on FE. Among the problems it solves are: Convection-Diffusion-Reaction, Incompressible Flows, Compressible Flows, Turbulence, Bi-Phasic Flows and Free Surface, Excitable Media, Acoustics, Thermal Flow, Quantum Mechanics (TDFT) and Solid Mechanics (large strain). Alya was developed specifically with parallelization in mind and is capable of running on large scale supercomputing facilities.

The initial target platform for Alya was conventional architectures, but it has recently been ported to Cell/B.E., where it has shown good performance compared to conventional cache based processors [28]. However, performance traces showed clearly that indirect addressing of data is a significant overhead of the total execution time of the element loop. Algorithm 1 showed the main loop of FE codes, which is composed of six tasks (lines 4 to 9). The computational intensive tasks on lines 5, 6, 7 and 8 (marked as C) are straightforward to port to Cell/B.E. and only require vectorization to fully exploit the SPE accelerators. On the other hand, tasks 4 and 9, are data intensive tasks (marked as D) and therefore suitable candidates for introducing the DLI method. This paper only covers task 4, gathering of element nodes, however, work is also being done to improve task 9, scattering of data.

The next section covers the gathering strategies explored for Alya on Cell/B.E., namely DLI and packing.

A. Gathering Strategies

Data intensive operations are the main concern for Alya since Computational Mechanics (CM) codes based on FE, have sparse data structures, due to graphs of elements and nodes of unstructured space discretizations. Sparse data structures are difficult to port to the SPEs, due to the many alignment constraints imposed by the Cell/B.E. architecture.

In this paper two different strategies are presented, for tackling the Alya data intensive gathering operations on the Cell/B.E. architecture. One is a packing based method, where the PPE gathers data in a conventional way and packs it into a sequential buffer in main memory for the SPEs to fetch, which is a bag-of-tasks parallelism. This strategy has the advantage that indirect addressing is very simple to do on the PPE due to its conventional architecture. Fig. 14 shows an overview of the ‘packing based’, bag-of-tasks parallelism strategy.

Although the results in this section show that the DLI method on Cell/B.E. can outperform the other architectures, the results are nevertheless only theoretical, since no useful computation is done. Therefore it would be practical to also verify the performance of DLI in a real application. The next section will thus cover the usage of the DLI method to improve Alya, which is a real FE application.
<table>
<thead>
<tr>
<th>Platform name</th>
<th>Architecture</th>
<th>Initials</th>
<th>IBM Power6</th>
<th>IBM BlueGene/P PowerPC405</th>
<th>AMD Opteron</th>
</tr>
</thead>
<tbody>
<tr>
<td>MarCeL</td>
<td>IBM PowerXcell 8i</td>
<td>Intel Xeon Core i7 X5570 (Nehalem-EP)</td>
<td>IBM Power6</td>
<td>IBM BlueGene/P PowerPC405</td>
<td>AMD Opteron</td>
</tr>
<tr>
<td>Inti</td>
<td>Intel Xeon Core i7-2600</td>
<td>IBM Power6</td>
<td>2.7 GHz</td>
<td>43.2</td>
<td></td>
</tr>
<tr>
<td>Sara</td>
<td>IBM Power6</td>
<td>4.7 GHz</td>
<td>13.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jugene</td>
<td>IBM Power6</td>
<td>13.6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loubi</td>
<td>IBM Power6</td>
<td>2.7 GHz</td>
<td>86.4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Chips per blade | 1 (PPE) + 8 (SPEs) | 2 | 2 |
| Cores per chip  | 2 (PPE) + 1 per SPE | 2 | 2 |
| Threads per core| 2 (PPE) + 1 per SPE | 2 | 2 |
| Clock           | 2.93 GHz | 75.2 |
| SP GFlop/s†     | 204.8 (SPEs) + 25.6 (PPE) | 32 KB + 32 KB | 46.88 |
| DP GFlop/s†     | 102.4 (SPEs) + 6.4 (PPE) | 32 KB + 32 KB | 46.88 |
| L1 Cache (D+I) | 32 KB + 32 KB | 64 KB + 64 KB |
| L2 Cache        | 512 KB per core | 4096 KB per core |
| L3 Cache        | 8 MB (shared) | 32 MB (shared) |
| Local Store     | 256 KB per SPE | 2 MB (shared) |
| Main memory     | 8 GB | 128 GB |
| Bandwidth       | 25.6 GB/s | 51.2 GB/s |
| Watts           | 95 | 100 |
| Compiler        | IBM XL Compiler Multicore Intel Compiler IBM XL Compiler | Intel Compiler IBM XL Compiler |
|                | Acceleration for Linux (v10.01/v11.01) | Acceleration for Linux (v11.1) |

**TABLE I**

Overview of the architectures used in this paper for benchmarking. The platforms are all PRACE [25] prototypes. †GFlop/s performance is calculated per chip.

---

The second strategy is DLI-driven and is based on the DLI method, presented in detail in the previous sections. In this method, elements are assigned an alignment type and two separate DMA list transfers are used to interlace elements of compatible alignment types, resulting, as shown before, in an efficient gathering of data.

The disadvantage of the DLI-based strategy is that it is more complex and requires more implementation effort, compared to the simple packing-based strategy. However, as shown, the performance improvements are very good and can be orders of magnitude better than conventional cache-based processors.

It is important to note that the only difference between the two strategies above is the way data is gathered. The computational kernels are identical for both strategies and all computations are done on the SPEs in parallel.

The next section will compare the performance of the two gathering strategies above, using Alya and a real world dataset.

---

**B. Performance Evaluation of Alya and DLI**

The following result shows the performance of Alya for task 4 of Algorithm 1, gathering of node data, which can represent a significant portion of the total execution time of the element loop. This task together with task 9, represent the main execution times of Alya.

To measure the performance of the two gathering strategies from the previous section, and to compare the performance with other architectures, the brain dataset from section VII was used to simulate the transport of red blood cells.

Fig. 16 shows the execution times of only task 4 (gathering), of Algorithm 1, for a single time-step iteration of the element loop of Alya. As can be seen, the DLI implementation is...
about 13x times faster than the packing based version. The execution time for the packing based version is composed of the time to pack data for the SPEs and the time for the SPEs to fetch the packed data into their LS. The execution time for the DLI based strategy is composed of the steps covered in section VI.

Since the above speedup is only for task 4 of Algorithm 1, it is interesting to also see the impact of DLI when considering the remaining element loop tasks. Below are the execution times for executing all of the element loop (lines 2 to 10 of Algorithm 1), for one iteration in Alya, on the platforms in Table I, from Section VII.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Time (sec)</th>
<th>DLI speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell/B.E. (DLI)</td>
<td>5.37</td>
<td>1</td>
</tr>
<tr>
<td>Cell/B.E. (Packing)</td>
<td>22.1</td>
<td>4.12</td>
</tr>
<tr>
<td>Power6</td>
<td>36.38</td>
<td>6.77</td>
</tr>
<tr>
<td>Nehalem</td>
<td>10.91</td>
<td>2.03</td>
</tr>
<tr>
<td>Opteron AMD64</td>
<td>22.87</td>
<td>4.26</td>
</tr>
<tr>
<td>BlueGene/P</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

The above table shows that the DLI method is the fastest method, follow once more by the Nehalem. Since BlueGene/P only has 2GB of RAM per node, it was unfortunately not possible to execute Alya on that platform, due to insufficient memory. The DLI method is around twice as fast as Nehalem, however, this time the comparison is against an actual FE application, running a real-world dataset, and is thus more fair.

IX. CONCLUSIONS

This paper has presented a technique called DMA List Interlacing (DLI), which is a novel method for gathering sparse data, at fine data granularity, on the Cell/B.E. architecture. Although DLI is not a general approach which can be used on all architectures, it was shown, using a micro-benchmark, that it is able to outperform the traditional method of indirect addressing by several orders of magnitude. The comparison was done between Cell/B.E. and several modern conventional architectures. The above speedups, however, require that the computational loads are sufficiently high enough, to hide the large DMA transfer times.

Another benefit of DLI is that the execution time remains constant, independent of the ordering of the mesh. This is due to the user controlled local stores and asynchronous DMA engines. One limitation of reordering methods over that of the DLI method, is that if the graph connectivity changes, then a new reordering step has to be carried out. This is not the case for DLI and therefore will have an additional advantage over reordering methods, for cases where the connectivity table changes frequently, such as S-adaptivity[29], Chimera[30] and Meshless[31] methods. This could be a direction for future research, along with using DLI for other problems with sparse data.

Finally, DLI was applied to Alya, which is a real FE application. It was shown that DLI improved the gathering performance of Alya by 13x compared to the traditional method, and performance comparisons against other common HPC architectures showed that it achieves speedups of over 2x. This is noteworthy, considering that the comparison was against several recent architectures and the Cell/B.E. architecture is several years old.

Although the DLI method presented in this paper is specific to Cell/B.E., other DMA engines which have gather/scatter capabilities, often have similar constraints as Cell/B.E., and therefore DLI could potentially also apply to other hardware.

REFERENCES


