IMPLEMENTATION OF PARALLEL MESH PARTITION AND GHOST GENERATION FOR THE FINITE ELEMENT MESH FRAMEWORK

BY

SAYANTAN CHAKRAVORTY

B.Tech, Indian Institute of Technology, Kharagpur, 2002

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

Abstract

The Finite Element Method framework allows the user to develop scalable parallel finite element applications easily. During initialization it reads in an input mesh and partitions it into a large number of chunks that are distributed among different processors. This partition process is sequential and memory intensive. Thus the partition algorithm is a bottleneck for running finite element applications with large meshes on a large number of processors. This thesis parallelizes the partition algorithm to remove the memory bottleneck on one processor. It also presents an algorithm to generate ghost elements in parallel.

To my parents for teaching me to think for myself.

Acknowledgments

I would like to thank my advisor Prof. L. V. Kale for his guidance, advice and patience. He has given me freedom to pursue my research interests and provided a wonderful environment for research. I am very thankful to Orion Sky Lawlor and Gengbin Zheng, my colleagues at the Parallel Programming Laboratory, for providing insights and advice that helped greatly with the work in this thesis. Everyone at the laboratory has helped make my stay here a thoroughly enjoyable and educational one. Last but not the least, I would like to thank my friends and parents for always being encouraging and supportive. I am thankful to the METIS research group at the University of Minnesota for the use of their software.

Table of Contents

List of Tables

List of Figures

Chapter 1

Finite Element Framework

1.1 Introduction

A large number of engineering applications with irregular domains are solved using the Finite Element Method (FEM) approach. The FEM approach decomposes the domain into a large number of disjoint finite elements. A finite element is defined by a set of nodes, that specify its shape and size. We refer to this set of nodes as the *connectivity* of that element. Most applications use simple finite elements such as straight lines for 1D, triangles or quadrilaterals for 2D and tetrahedra or hexahedra for 3D. A single domain can contain elements of different types. All the elements and nodes in a domain together make up a *mesh*. An example of a 2D mesh with 3 triangular elements and 5 nodes is shown in Figure 1.1. The connectivity of the elements of the example mesh is shown in Table 1.1. A typical FEM application performs some element based calculations, which update adjacent nodes and then some node based calculations.

Many interesting FEM applications want to perform accurate simulations over large

Element	Adjacent Nodes		
e1	n1	n3	n4
e2	n1	n2	n4
e3	n2	n4	n5

Table 1.1: Connectivity table for the mesh in Figure 1.1

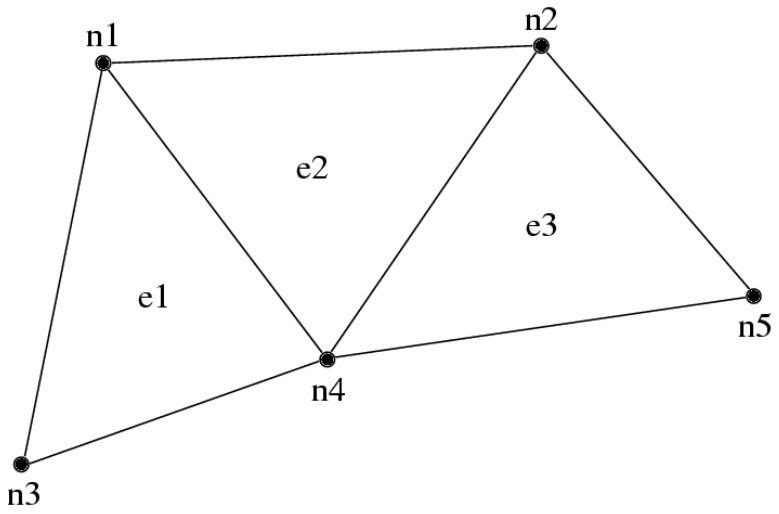

Figure 1.1: A 2D mesh with 3 triangular elements and 5 nodes

domains. These applications have meshes with millions of elements and nodes. These simulations take huge periods of time to run on single processors. Some meshes cannot even be loaded into the memory of a single processor. Parallelizing such applications provides a way to run these simulations for a large number of steps within a practical amount of time. However parallelizing these applications requires a significant effort on the part of the programmer.

The FEM framework [1] developed in the Parallel Programming Laboratory, reduces the programming effort of parallelizing the application. It allows the user to write sequential pieces of code that specify the mathematical model and leave the details of parallelization to the FEM framework. The FEM framework partitions the original mesh into a large number of smaller parts or *chunks*. During partitioning we give nodes and elements new local numbers. Each element belongs to a particular chunk. Nodes shared among elements within a particular chunk belong to that chunk. Nodes that are shared among elements on different chunks are duplicated in those chunks. We refer to such nodes as *shared nodes*. We have partitioned the original mesh of Figure 1.1 into two chunks, A and B in Figure 1.2. The connectivity of the elements of the two chunks is shown in Table 1.2. The framework provides methods to periodically synchronize the duplicate copies of a shared node on different chunks.

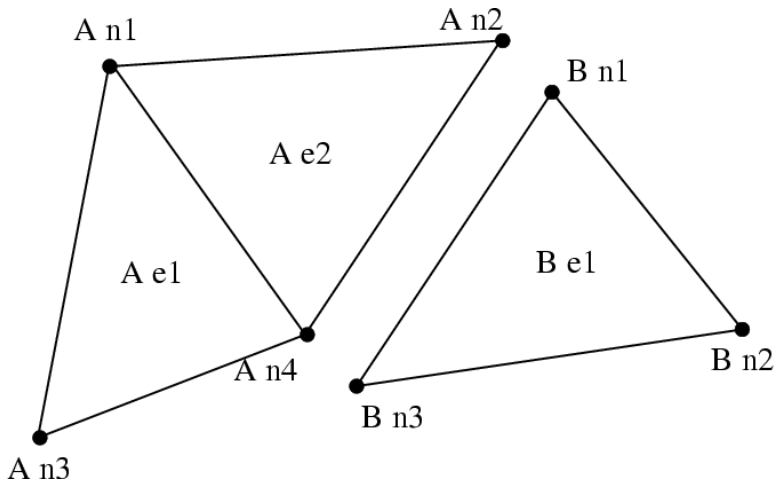

Figure 1.2: Partition of the mesh in Figure 1.1 into 2 parts A and B

Connectivity data of Chunk A

Element	Adjacent Nodes		
e1	n1	n3	n4
e2	n1	n2	n4

Connectivity data of Chunk B

Element	Adjacent Nodes		
e1	n1	n2	n3

Table 1.2: Connectivity table for chunks A and B of the partitioned mesh in Figure 1.2

The current version of the FEM framework uses MPI[5, 4] for its communication needs. Although it can be used with any implementation of MPI, using it with AMPI [6] allows it to reap the benifits of virtualization [9] such as overlap between communication and computation and run time load balancing.

1.2 Partitioning

The FEM framework reads in the entire input mesh on processor 0. It then partitions the input mesh into multiple smaller meshes or chunks. The user code for each such chunk executes in a separate MPI process. It uses the graph partitioning tool METIS [12, 14, 13] to obtain a mapping from elements to chunks. It uses the mapping to construct each chunk

Shared Nodes on A	Shared Nodes on B
n2	n1
n4	n3

Table 1.3: Shared node lists for the chunks A and B of the partitioned mesh in Figure 1.2

with the connectivity and user data associated with its elements and nodes and then sends the chunk to its corresponding MPI process. It also sets up a mapping between different copies of a shared node. Each pair of chunks that share nodes map their shared nodes through a pair of lists of node indices. Each list contains the local node indices of the nodes shared with the other chunk. The lists are sorted such that corresponding entries in the list of each chunk refer to duplicates of the same shared node. The lists of shared nodes for the chunks A and B in Figure 1.2 are specified in Table 1.3.

The FEM framework currently partitions and builds all the chunks on processor 0. Since partitioning is fairly memory intensive, it is not always possible to partition a big mesh into a large number of pieces on a single processor. The calculation does not fit into the memory available on a single processor on non-shared memory clusters. Moreover building all the chunks on processor 0 as FEM does currently means that at some instant processor 0 not only stores the entire input mesh, but also the chunks for all the other processors. This increases the memory consumption on the already loaded processor 0 even more. Thus the sequential partition method becomes a bottleneck for the FEM framework. We measure the maximum memory used on processor 0 by the partition algorithm while dividing meshes of different sizes into varying number of chunks.

Figure 1.3 shows the memory consumption on processor 0 while dividing a small mesh of 20 thousand elements into different numbers of chunks. We can see that memory consumption increases with increasing number of chunks. When the number of chunks is of similar order to the number of elements the memory consumption increases almost linearly. However this is a somewhat extreme case. Figure 1.4 shows the memory consumption for a more substantial mesh of 200 thousand elements. This plot shows a trend similar to the

smaller mesh. We also partitioned a large mesh with 2 million elements into varying number of pieces. When we tried to partition it into 12 thousand chunks, the processor 0 ran out of memory and the program crashed. The results for smaller number of chunks are shown in Figure 1.5. The three graphs show that although the serial partition algorithm is suitable for small meshes, it runs out of memory on a single processor while trying to partition large meshes. The increase in memory consumption with increasing number of elements is demonstrated in Figure 1.6.

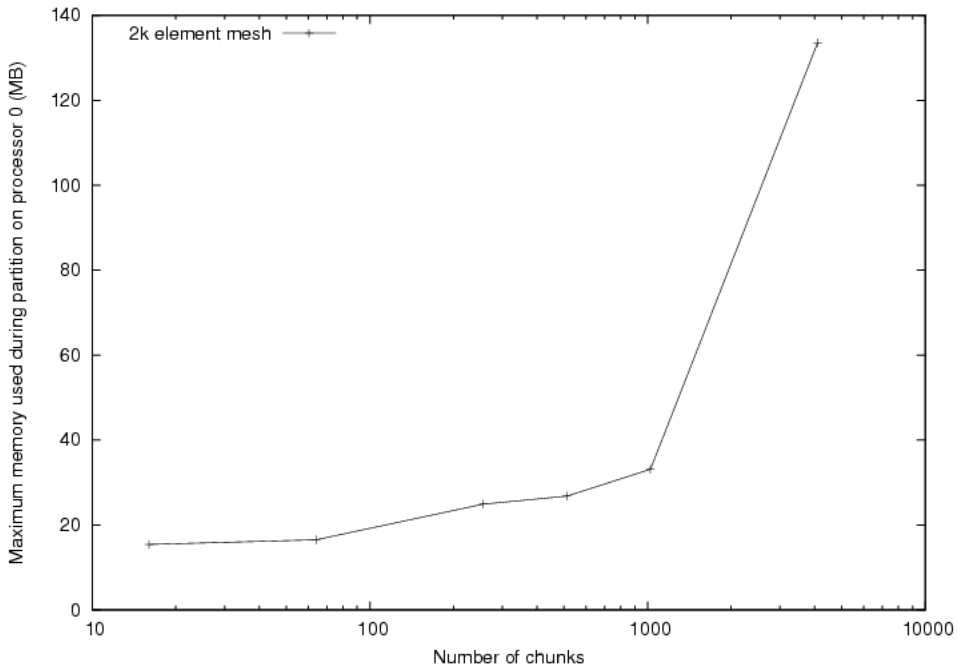

Figure 1.3: Maximum memory used by the serial algorithm while partitioning a 20k element mesh into different numbers of chunks

It is possible to work around this problem by partitioning the mesh into the required number of chunks on a shared memory machine with lots of memory and writing out the partitioned chunks into files. Each of these files can be read in by a MPI process running on a non-shared memory machine. The FEM framework provides support for this. However this process has a lot of drawbacks. A large shared memory machine might not be available to the user. Partitioning the mesh on one machine and then shipping them to another before the application can be run is tedious. Moreover, if the user wants to change the number of

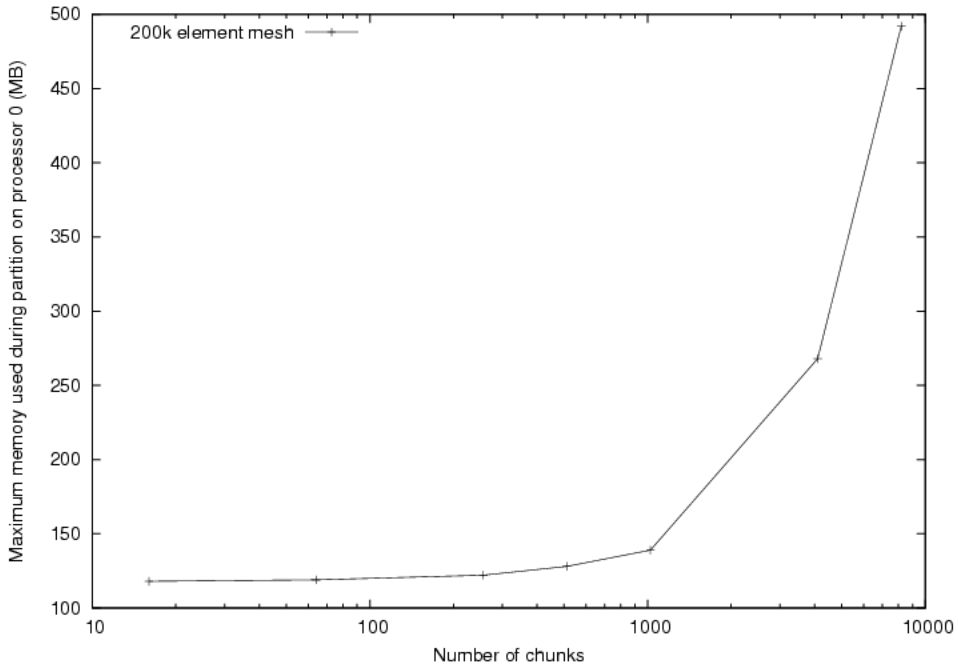

Figure 1.4: Maximum memory used by the serial algorithm while partitioning a 200k element mesh into different numbers of chunks

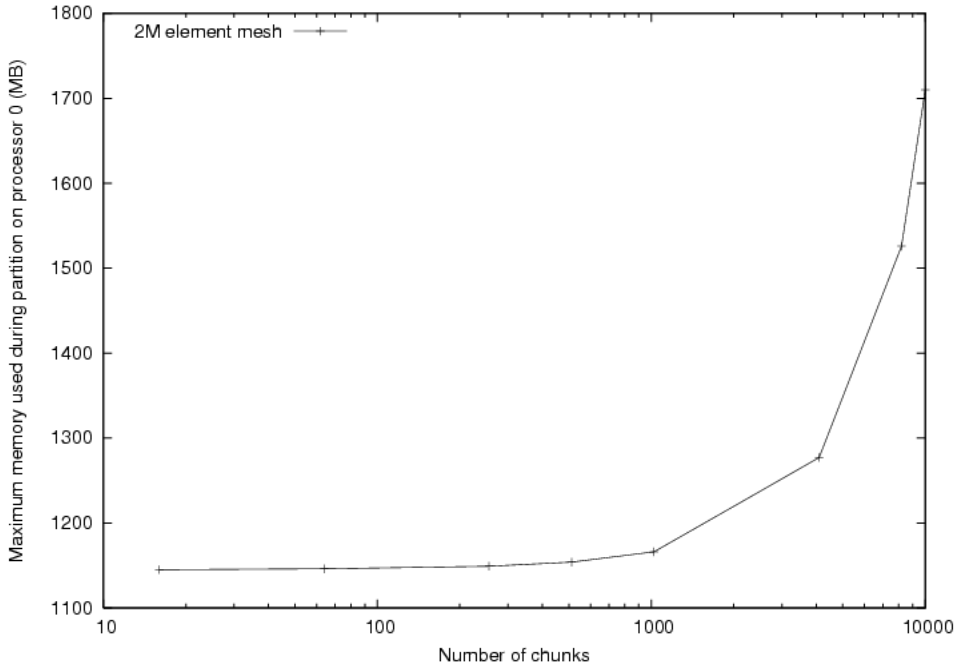

Figure 1.5: Maximum memory used by the serial algorithm while partitioning a 2 M element mesh into different numbers of chunks

chunks, he has to go back to the shared memory machine and repartition the mesh into the new number of chunks.

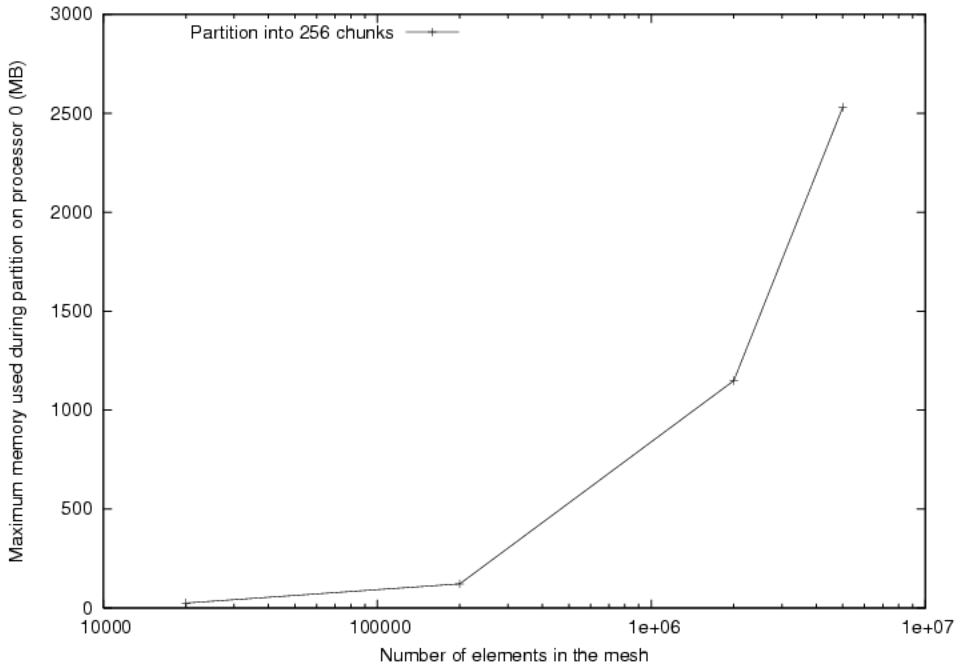

Figure 1.6: Maximum memory used by the serial algorithm while partitioning meshes of different sizes into 256 chunks

Parallelizing the partition and chunk building process seems to be the obvious way of reducing the memory bottleneck at processor 0. We describe the implementation in Chapter 3.

1.3 Ghost Generation

Some finite element calculations perform element-based calculations in which each element requires data from its neighboring elements. This means that elements on the boundary of a chunk require data from elements on another chunk. The FEM framework solves this problem by adding ghost elements to the boundary of a chunk. A ghost entity (element or node) is a local, read-only copy of an entity on another chunk. This allows the elements at the boundary to access data from elements on another chunk in a seamless manner. The FEM framework provides mechanisms to keep the read-only copies of elements synchronized with their original version.

The FEM framework allows the user to specify what 'neighboring' exactly means for a particular calculation. In a particular application two tetrahedra that share an edge might be considered neighbors, whereas in another case only tetrahedra that share faces might be considered neighbors. The user can also have multiple layers of ghosts for applications that need neighbors of neighbors. The user describes the 'neighboring' relationship by specifying a *face*. An element will be added as a ghost to your chunk if it shares a face with at least one of your elements. The user specifies the number of nodes that make up a face, the number of faces per element and the list of faces in an element. The user might choose to create read-only copies of nodes of ghost elements that do not already exist on a chunk. Such nodes are referred to as *ghost nodes*.

Figure 1.7 shows a mesh broken up into two chunks. Figure 1.8 shows each chunk with ghosts added for boundary elements that share an edge. Since the shared face in this case is an edge, each face has 2 nodes and each element has three faces. Figure 1.9 shows the same chunks with ghosts added for boundary elements that share a node. In this case the shared face is a single node and each element has 3 faces that they might share. As expected, the number of ghost elements is significantly higher when the shared face is a node instead of an edge.

After partitioning the input mesh, the FEM framework generates ghost layers for each chunk according to the user's specification. At the moment this is implemented as a sequential method running on processor 0. The first step in adding a layer of ghosts for a chunk is to find out all the elements on other chunks that have a face common with elements on it. For millions of elements, this looks like a very computationally expensive process. However the cost can be significantly reduced by observing that only elements that have at least one node on a chunk boundary might require ghosts or possibly be ghosts on another chunk. We refer to elements that have at least one shared node as *interesting* elements.

The problem now reduces to finding out all pairs of intersecting sets among a list of sets where each interesting element is a set and the faces in those elements are the members of

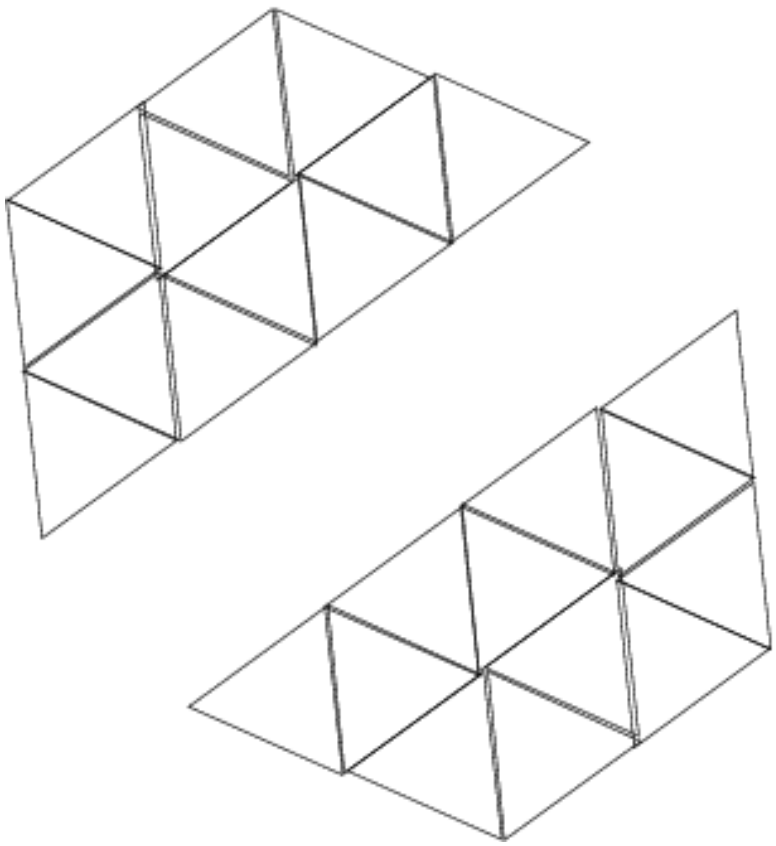

Figure 1.7: A mesh with two chunks to which we shall add ghosts

the corresponding set. A hashtable is created in which the members of the sets (faces of the elements) are the keys. Each set is put into the hashtable once for each of its members. When there is a collision, the new entry is queued after the old contents. After all the sets have been added to the hashtable, each location in the hashtable contains a short list of sets (elements) that might intersect with each other. An intersection test among the sets at each location and collecting the result across all locations yields the list of intersecting sets for each set. Once we know the elements on different chunks that share faces, the data for the ghost elements are added to the appropriate chunk.

If there are multiple layers, the FEM framework starts with the first (innermost) layer of ghosts. After having added the first layer, it adds the second layer on top of it by treating the first layer of ghosts as part of the mesh for the second layer. For the second layer it also marks all elements that share a node with the interesting elements of the first layer as

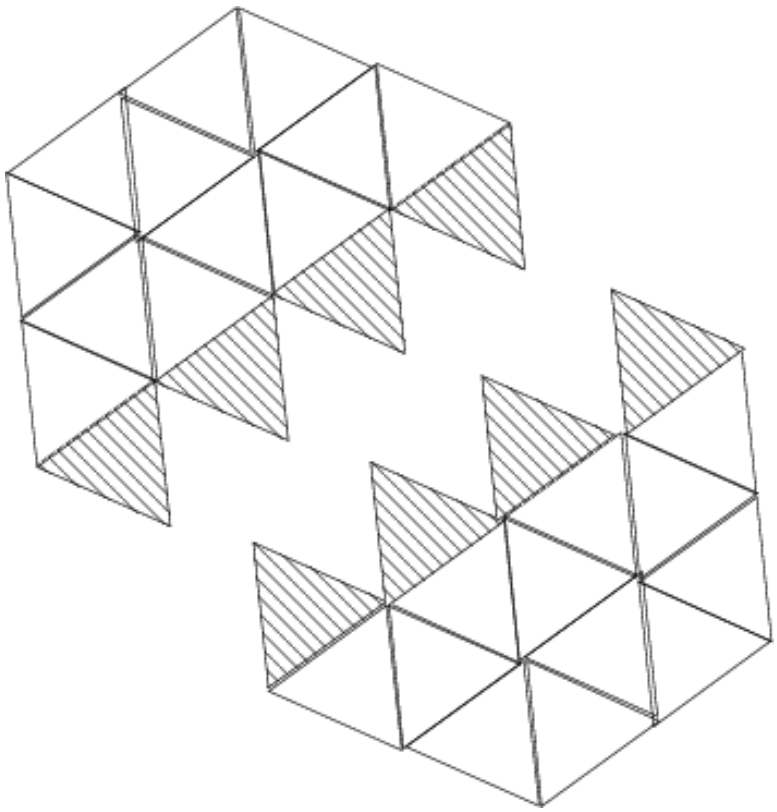

Figure 1.8: The mesh in Figure 1.7 with ghosts added for boundary elements that share edges. The shaded elements are the ghosts

interesting as well. Care is taken that the same element does not get added as a ghost on the same chunk multiple times.

As mentioned earlier this mechanism is entirely sequential. Since the current implementation requires all the chunks to be present on processor 0, it would negate a large part of the benefit of parallel partition. We solve this problem by parallelizing the ghost generation algorithm.

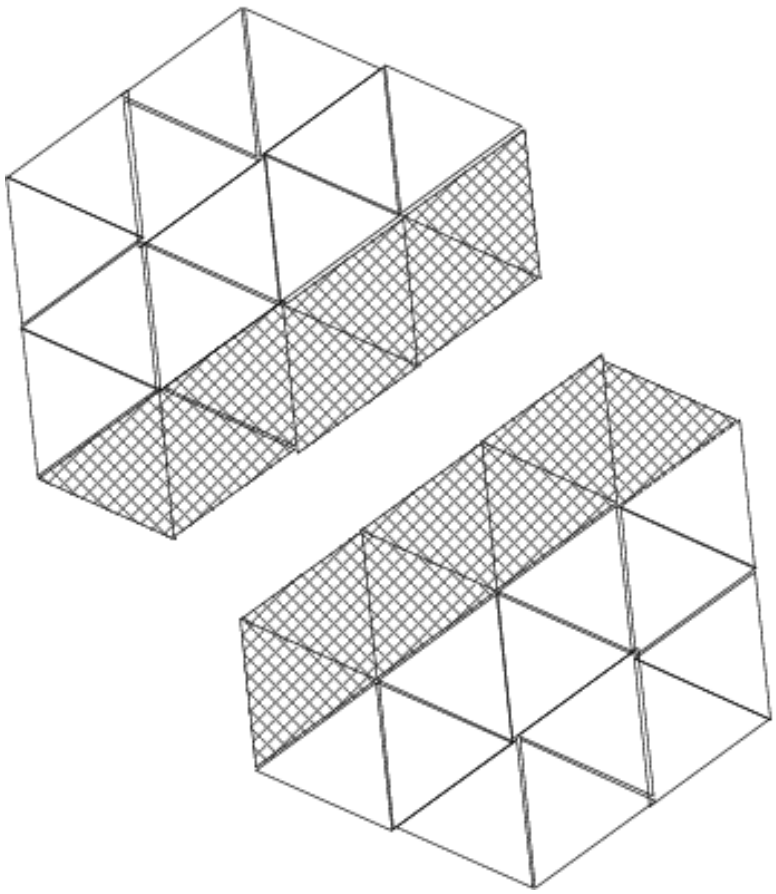

Figure 1.9: The mesh in Figure 1.7 with ghosts added for boundary elements that share nodes. The shaded elements are the ghosts

Chapter 2

Charm++ and MSA

2.1 Charm++

Processor virtualization [9] is the key idea behind Charm++[7, 8]. The user breaks up his computation into a large number of objects without caring about the number of physical processors available. These objects are referred to as *virtual processors*. The user views the application in terms of these virtual processors and their interactions. The Charm++ runtime system allows the virtual processors to interact among each other through asynchronous method invocation. Figure 2.1 shows the users view of an application and the system view of it.

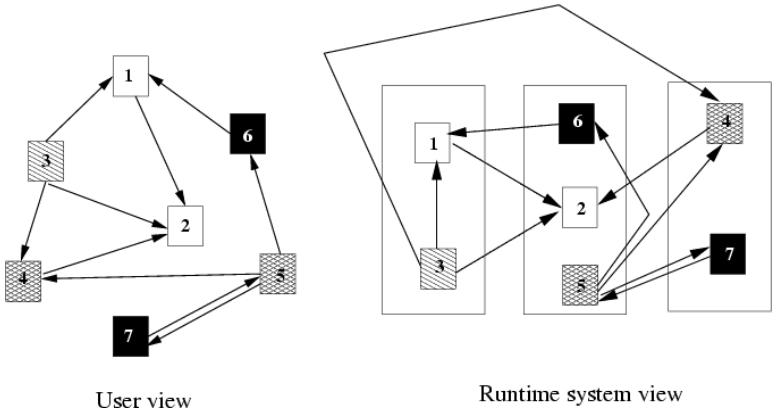

Figure 2.1: The Charm++ runtime system maps virtual processors to physical processors. The small squares are objects or virtual processors. The large ones refer to physical processors.

The Charm++ runtime system also maps the virtual processors to physical processors. The user code is independent of the location of a virtual processor. This allows the runtime system to change the mapping of virtual processors to physical processors in the middle of an execution. It *migrates* one virtual processor on one physical processor to another physical processor at runtime. The Charm++ runtime system allows for message delivery and collective operations such as reductions and broadcasts in the presence of migrations. So the runtime system can migrate objects away during an execution to adapt to the changing load characteristics of a problem. If at some point in the execution some objects on one processor start doing more work, the runtime system can distribute them uniformly among all the other physical processors. Thus processor virtualization enables us to perform measurement-based run-time loadbalancing. Distributed and centralized load balancers can be developed to remap virtual processors to physical. Runtime load balancing has been used to scale molecular dynamics to thousands of processors [10].

Another major benefit of processor virtualization is the automatic adaptive overlap between computation and communication. If one virtual processor on a processor is waiting for a message, another one can run on the same physical processor.

2.2 AMPI

Adaptive MPI (AMPI) [6, 2] is an implementation of the Message Passing Interface (MPI)[4, 5] on top of Charm++. Each MPI process is a user-level thread bound to a Charm++ virtual processor. The MPI communication primitives are implemented as communication between the Charm++ objects associated with each MPI process. Traditional MPI codes can be used with AMPI with slight modification. These codes can also take advantage of automatic load balancing and adaptive overlap of communication and computation.

The FEM framework is written on top of MPI. Its structure was found to be a very good fit for the MPI style of programming. Although it works on all MPI versions, using it

with AMPI provides the advantages of processor virtualization. Load balancing is especially useful for applications with refinement where the amount of computation on a particular chunk can change significantly as the number of elements on it varies. It is also helpful in applications where the computation load for some elements increases during the simulation.

2.3 MSA

Multiphase shared array (MSA)[3] is a distributed shared memory model in which the data is accessed in phases. In each phase all the data elements in a particular array are accessed in the same mode by all participant threads. The mode of an array can be changed between phases. The different phases of access of a MSA are separated by synchronization operations on that array. The different modes supported by MSA are

1. **read-only**: The participants can only read the data elements in a MSA. If a participant reads a page that is not local, it gets blocked. Prefetch calls can be issued by the participants to efficiently read remote data.

2. **write-many**: All participant threads are allowed to write to the MSA. However only one participant can write to any particular data element. It is the user code's responsibility to make sure that multiple threads do not write to the same element. The runtime throws an exception when it detects a violation.

3. **accumulate**: Multiple participant threads can accumulate data for the same data element. The data contributed by different array elements is combined using a commutative-associative operation. The commutative-associative operator to be used is specified when a MSA is set-up. It can also be changed dynamically in between phases. The user is allowed to define his own operations apart from the built-in operations like addition and maximum.

The data elements in a MSA can be either C++ built-in types or user defined types that have to support a particular interface.

MSA is implemented on top of Charm++. Data elements are grouped together into *pages*, each of which is a Charm++ virtual processor. These pages can be load balanced at run time on the basis of computation and communication load. This allows a page to move to a processor where most of its accesses originate. MSA also allows us to set a cap on the memory used per processor. Thus it can be used to read in a huge amount of data on one processor but store it simply and efficiently across other processors.

Chapter 3

Parallel Partitioning

We discuss our method of parallely partitioning the input mesh into a large number of chunks. Section 3.1 specifies the various conditions that we would like our parallel partitioning algorithm to meet.

3.1 Specifications

The primary requirement for the parallel algorithm is that it should not have a memory bottleneck on one processor like the serial version. This means that not only should the process of calculating the mapping of elements to chunks be parallel but also that we should avoid storing the whole mesh on one processor. So we should also parallelize the process of building chunks: sending node and element data to the corresponding chunk, figuring out the shared nodes and setting up the communication lists for the shared nodes.

3.2 Algorithm

The parallel partitioning algorithm can be broken up into three parts.

1. Calculate a mapping of elements to chunk that tries to minimize the number of elements on the boundaries between different chunks

2. Use the mapping from the previous step to create chunks that contain all the data for the elements and nodes mapped to them.

3. Find out the nodes that are shared between different chunks and set up the communication lists.

We use ParMETIS [11] to calculate the mapping from elements to chunks. ParMETIS works by having each processor call it with the connectivity data for some elements. In order to distribute the computation load more or less equally among the processors, it should be called with the connectivity data for equal number of elements on each processor. This means that we should partition the mesh before we call ParMETIS. This is something of a chicken and egg problem. How do we partition the mesh if we need to partition it before we call the partitioning algorithm?

We solve this by using a trivial strategy to initially break up the input mesh into chunks that contain the same number of elements and nodes. If the number of elements per chunk is k, we send the first k elements to the first chunk, the i^{th} chunk receives elements numbered from $(i-1)k$ to $ik-1$. We divide up the nodes in the same way. This trivial partition probably produces chunks that are not contiguous and have lots of boundary elements. So these partitions can not be used for the application but allows us to break up the mesh fast into equal sized chunks. It also allows us to pipeline the process of reading in the mesh and carrying out the trivial partition. So we do not need to store the whole mesh on one processor at the same time. Each process is deemed to be *responsible* for the chunk that it gets during the trivial partition. After the ParMETIS call, each processor knows the mapping of each element that it is responsible for.

The next step in the partitioning algorithm requires us to find out the nodes and elements that belong to each chunk. Each processor knows the chunk to which each of the elements it is responsible for is mapped (henceforth referred to as the *owner* of that element). Each processor sends all the elements on it to their owners.

However sending the nodes to their owners is complicated by two factors: each node might be owned by multiple chunks and after the call to ParMETIS each processor knows the owner for the elements but not the nodes it is responsible for. As a first step to sending the nodes to their owners, each processor finds out the owners of the nodes that are present in the connectivity of the elements it is responsible for. The owner of an element is one of the owners of all the nodes in the connectivity of that element. This data is then collected over all processors. From this each processor extracts the ownership information for all the nodes that it is responsible for. It then sends the node to all the chunks that own it.

At this point each chunk has all the nodes and elements it owns. Now it needs to set up the communication lists for shared nodes as mentioned in Section 1.2. Each chunk needs the list of nodes it shares with all its adjacent chunks. For this it uses the global node ownership information calculated in the previous step. A chunk finds out all the other chunks that also own a node owned by it. Then it finds out all the nodes that a chunk shares with another chunk and uses this information to set up the communication lists.

3.3 Implementation Issues

A number of steps in the parallel partitioning require us to collect data scattered across multiple processors. As an example, after the call to ParMETIS all the elements owned by a particular processor are scattered across different processors and need to be brought to that processor. The set of processors containing elements owned by a processor is dynamic and depends on the input mesh. An operation like this seems well suited for a distributed shared memory model.

One could declare a global array of chunks, one for each processor. Each processor goes through the list of elements it is responsible for and adds it to the chunk of its owner. After all the processors have finished writing in their elements, each processor reads its chunk in the global array to obtain the list of elements it owns. The phased nature of the calculation

makes it an ideal match for the MSA described in Section 2.3. We create a MSA of chunks. A chunk is just a list of elements owned by that processor. Each processor adds elements to the chunks in the accumulate mode. The accumulate operation takes two lists of elements owned by the same processor and merges them. It removes any duplicate elements and sorts them in increasing order of global element number.

We used MSA because it not only simplified the implementation, but also provided additional features such as collecting the result of the accumulate calls locally before sending it out to the destination processor. An MPI implementation would have been far more compilcated and we would have had to implement all the additional features that MSA offers for free. The performance of the parallel partitioning algorithm is evaluated in Chapter 5.

Chapter 4

Parallel Ghost Generation

We discussed the sequential algorithm for generating ghosts in Section 1.3. We develop a parallel version of that in this chapter and describe some of its important implementation issues.

4.1 Algorithm

The input for the ghost generation algorithm on each processor is

1. The list of nodes and elements in the chunk on that processor.
2. The list of nodes shared with other chunks.
3. The number of ghost layers and the specification of each layer as described in Section 1.3.
4. A global numbering for the nodes and the elements.

As in the sequential case, the parallel algorithm adds multiple layers of ghosts by adding each new layer of ghosts on top of the older ones. So we first take a look at how a single layer of ghosts is added. The first step of the algorithm creates a distributed hashtable with the faces of elements as keys. For each face of an element, the hashtable can store a list of elements to which it might belong and the chunks on which those elements exist.

Each processor enumerates all the faces for each of its elements. It adds an element to the hashtable for each face of the element in which all the nodes are shared. It uses the face as the key while adding the element to the hashtable. It also maintains a list of tuples of faces and their elements that were added to the hashtable. It waits for all the other processors to finish adding to the hashtable.

Then the processor considers each face and element tuple that it added to the hashtable. For each such tuple, it reads from the hashtable the list of other elements that might share the tuple's face with the tuple's element. Since multiple faces might be mapped to the same slot in the hashtable we are not sure that all the elements in the list actually share the same face. So the list is pruned down to elements that actually share the tuple's face with its element. The tuple's element gets sent as a ghost element to the chunks that contain these elements. If ghost nodes are to be added for this layer then, the nodes of the ghost elements are sent along with the element itself. If those nodes already exist on the receiving chunk, they are simply discarded.

When a second layer of ghosts is to be added on top of an existing layer, we treat the existing layer as part of the mesh. We also treat all the nodes of elements that got sent out as ghosts for the previous layer as shared nodes.

4.2 Implementation Issues

The distributed hashtable was implemented as a MSA. Each data element in the global array is a list. Each object in the list represents a face of an element. It stores the face, the element identifier and the chunk on which that element is found. The face is stored as a tuple of node identifiers. It is also used as the key for the distributed hashtable. A hash function takes in the tuple of nodes as argument and returns an index into the MSA. The hash function is defined such that the order of nodes in the tuple does not make any difference to the index returned.

In its current implementation a MSA has a fixed number of slots that have to be specified when the MSA is setup. So a hashtable built on top of a MSA can not be resized. This means that we need to choose a suitable number of slots for our hashtable when it is created. If the number of slots were too low then too many faces would share the same slot and the check to find out the actual elements that share a face would become too costly. In the worst case it makes the whole hashtable redundant. If the number of slots were too large then most of the slots would be empty and waste a large amount of memory. We decided to use as many slots as there are shared nodes in the whole mesh since the number of shared faces is dependent on the number of shared nodes. Prior experience with the sequential version had shown us that the relationship is more or less linear.

Chapter 5

Results and Conclusion

5.1 Performance Evaluation

The purpose of parallelizing the partition and ghost generation process is to reduce memory consumption on processor 0. It distributes the calculation among all the processors. It should remove the memory bottleneck on processor 0 and allow the FEM framework to run with larger meshes and on a larger number of processors. We evaluate whether our implementation matches these goals.

We verify that memory consumption during partition on processor 0 has indeed been reduced by parallelizing the partition procedure. A 200 thousand element mesh is partitioned into 256 chunks on varying number of processors. Figure 5.1 shows the memory consumption on processor 0 for different numbers of physical processors. On 1 physical processor the parallel partition algorithm consumes nearly 4 times the serial partition algorithm. However the memory consumption on processor 0 decreases with increasing number of physical processors. The break even point for this data set is around 3 processors, at which the parallel implementation consumes less memory than the sequential one. This means that the parallel algorithm should be used with this dataset for runs with more than 3 processors. The memory consumption flattens out for higher number of processors as the decrease due to fewer chunks per processor is offset by more replication in MSA because of the higher number of physical processors.

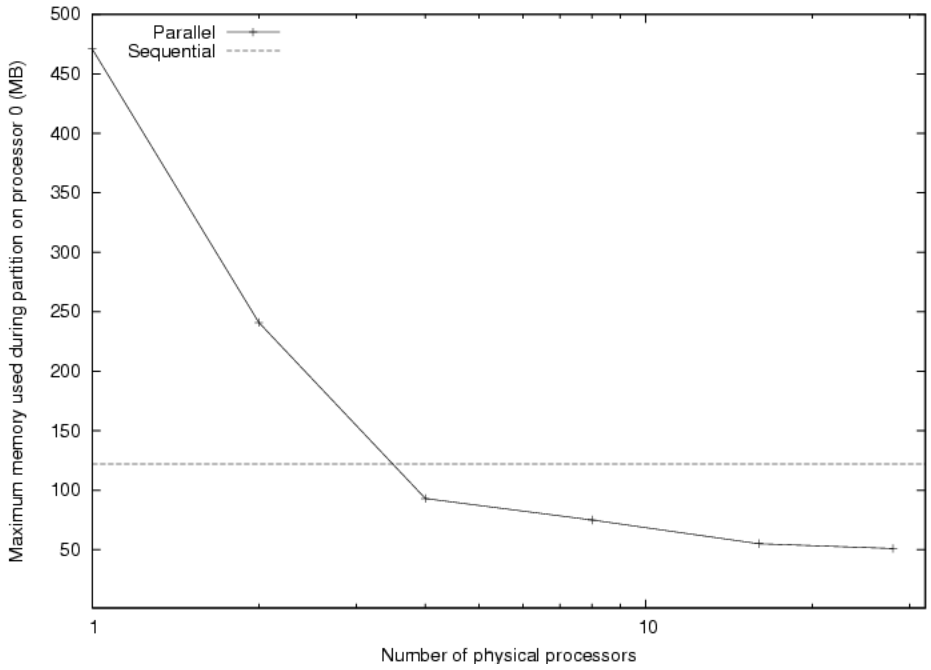

Figure 5.1: Maximum memory used on processor 0 while partitioning a 200k element mesh into 256 chunks on different numbers of physical processors

A major problem with the serial partition algorithm was that memory consumption on it increased as the same mesh was broken into increasing number of chunks. So the FEM framework could run a mesh on a small number of processors but not on a large number of processors. This defeated the primary purpose of the framework, which is to allow applications to scale to large number of processors. The parallel partition does not suffer from this problem. We partitioned the same mesh as above on different number of processors. We partitioned it into as many chunks as there are physical processors. We did this for both the serial and parallel implementations. Figure 5.2 shows that memory consumption decreases with increasing number of processors in the parallel case, whereas it increases in the serial case. This is in line with expectations.

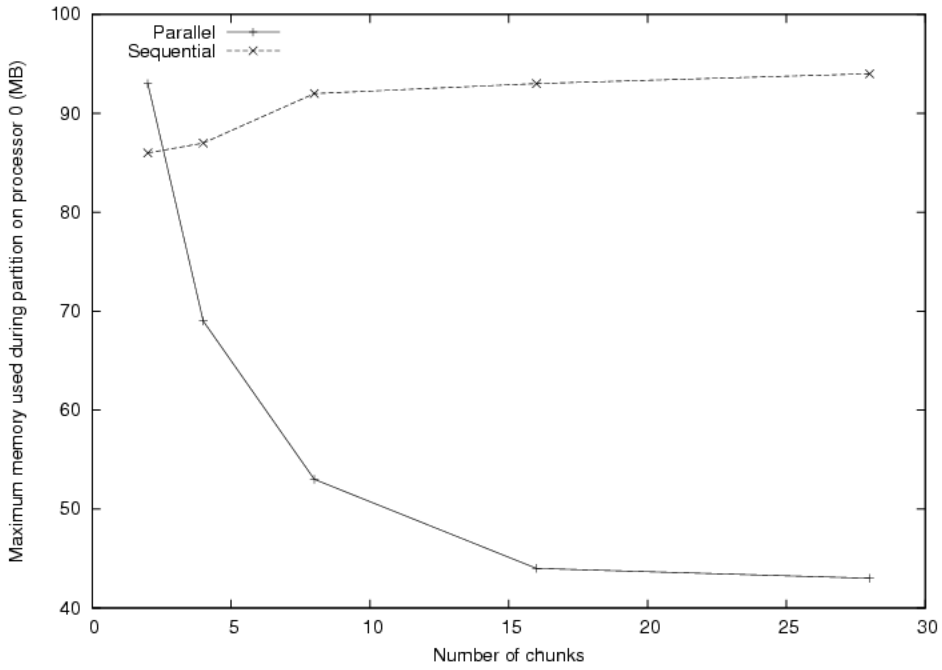

Figure 5.2: Maximum memory used on processor 0 while partitioning a 200k element mesh into different number of chunks. The number of physical processors and chunks is the same

5.2 Conclusions

The parallel partition algorithm relieves the memory bottleneck at processor 0. For the same dataset and number of chunks, the memory consumed on processor 0 decreases with increasing number of physical processors. Although the parallel algorithm consumes more memory than the sequential one when used on one processor, its memory consumption falls below the sequential one when sufficient number of processors are used. In the example in Figure 5.1 the parallel algorithm on 3 processors consumes less memory on processor 0 than the sequential one. So runs with large data sets and number of chunks, which would have been impossible with the sequential algorithm become possible with the parallel alogrithm.

5.3 Future Work

The memory consumption of the parallel algorithm should be studied and decreased so that the break-even point can be lowered. A more detailed study of the break-even points for

different meshes and different numbers of chunks should be performed. This data can be used to let the FEM framework choose the serial or parallel algorithm for partitioning a particular mesh into a certain number of chunks.

References

[1] Milind Bhandarkar and L. V. Kalé. A Parallel Framework for Explicit FEM. In *Proceedings of the International Conference on High Performance Computing (HiPC 2000)*, volume 1970, pages 385–395. Springer Verlag, December 2000.

[2] Milind Bhandarkar, L. V. Kale, Eric de Sturler, and Jay Hoeflinger. Object-Based Adaptive Load Balancing for MPI Programs. In *Proceedings of the International Conference on Computational Science, San Francisco, CA, LNCS 2074*, pages 108–117, May 2001.

[3] Jayant DeSouza and Laxmikant V. Kalé. MSA: Multiphase specifically shared arrays. In *Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, Indiana, USA, September 2004.

[4] Message Passing Interface Forum. MPI: A message-passing interface standard, 1994. http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html.

[5] Message Passing Interface Forum. MPI-2: Extensions to the message-passing interface, 1997. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[6] Chao Huang, Orion Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03)*, College Station, Texas, October 2003.

[7] L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[8] L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[9] Laxmikant V. Kalé. Performance and productivity in parallel programming via processor virtualization. In *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, Madrid, Spain, February 2004.

[10] Laxmikant V. Kalé, Sameer Kumar, Gengbin Zheng, and Chee Wai Lee. Scaling molecular dynamics to 3000 processors with projections: A performance analysis case study. In *Terascale Performance Analysis Workshop, International Conference on Computational Science(ICCS)*, Melbourne, Australia, June 2003.

[11] George Karypis and Vipin Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 35, 1996.

[12] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[13] George Karypis and Vipin Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–13, Washington, DC, USA, 1998. IEEE Computer Society.

[14] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48:96 – 129, 1998.