LB_Migrate: A Dynamic Load Balancing Library

Rohit Chaube[1], Ricolindo L. Cariño[2], Ioana Banicescu[3]
rohit@erc.msstate.edu, rlc@erc.msstate.edu, ioana@cse.msstate.edu
Mississippi State University

[1,2]Department of Electrical and Computer Engineering
[2]Center of Computational Sciences -ERC
[2,3]Department of Computer Science and Engineering

Abstract

The design of a general-purpose dynamic load balancing library for a vast variety of parallel applications is more challenging than the design of a static partitioning library. The dynamic load balancing library needs to be implemented in parallel with the application and must utilize memory efficiently, so that the application scalability is not affected. This paper studies the need for a dynamic load balancing library and looks at the related work. We propose a new dynamic load balancing library called `LB_Migrate`, targeted for large scientific applications with parallel loops as a major source of concurrency. The library is designed to be independent of any application. Hence, it must be supplied with a routine in the application that encapsulates the computations for a chunk of loop iterates. The library consists of various scheduling methods and load balancing strategies for performing both scheduling and data migration. Performance evaluation on a Linux cluster indicates that the library reduces the cost of executing an irregular loop without load balancing by up to 68%.

1. **Introduction**

The efficient use of parallel computers requires two objectives to be achieved [1]. First, the processors must be kept busy doing useful work. And second, the amount of interprocessor communication must be kept to a minimum. In many scientific problems, the above objectives are achieved by the single assignment of task to processors, and that does not change over the course of a simulation. Such a "static" distribution includes applications like the traditional finite element and finite difference methods, dense linear solvers, and most iterative solvers. There are several static partitioning tools that have been developed to support static task distribution, such as Sandia's Chaco [5] and METIS from the University of Minnesota [7].

A number of important applications such as Adaptive Mesh Refinement (AMR), adaptive physics models, particle simulations, multiphysics simulations, crash simulations, etc., have workloads that are unpredictable and vary over the course of computation. For such type of applications, high performance can only be obtained if the work load is distributed among processors in a time-varying "dynamic" fashion (i.e., at runtime, during the execution of the application). These applications impose different requirements upon dynamic load balancing algorithms and software. Some applications like particle simulations are fundamentally geometric in nature while others such as AMR are described using mesh connectivity. Thus, a complicated tradeoff needs to occur between the cost of the load balancer, the quality of partition it produces and the amount of data needed to be redistributed [1].

A number of large scientific applications typically have parallel loops as major source of concurrency. These loops have no dependencies among iterates and can be executed in any order, or even simultaneously without affecting the correctness of the computations. To achieve high performance, these loops should be scheduled to execute in parallel to minimize the completion time and also maximize processor utilization [4]. However, achieving this goal of high performance is not simple. Parallelization strategies have to account for algorithmic and systemic performance degradation factors. These factors which may be predictable or unpredictable give rise to unbalanced processor workloads. Hence, dynamic load balancing is an ideal solution for such type of applications.

The designing of dynamic load balancing algorithms poses significant challenges as compared to the design of static partitioning algorithms. Along with the issues of well balanced distribution of work and minimum amount of interprocessor communication, a dynamic load balancer should satisfy a few more requirements [1]. When integrated within the application it should run atleast as fast as its straight forward parallelization, and memory usage should be kept modest. Moreover, a small change in the problem

induces only a small change in the decomposition and it should have a deterministic communication pattern.

Generally, dynamic load balancing algorithms are implemented directly with the application resulting in a close coupling between the application and the load balancing algorithm's data structure. Such an implementation cannot be used with other applications due to the specificity of the data structure shared both by the application and the algorithm. Also, the application developer may not have expertise in optimizing the partitioning algorithm, and implementing the load balancing algorithms takes time away from the developer's primary interest – the application.

Software toolkits can provide solutions to these problems. The toolkits consist of sets of data partitioning algorithms and data migration strategies, thus providing an application developer the flexibility to choose the most suitable algorithm and optimize its results. The toolkits must have function call interface to obtain information from the application, and at the same time it should be independent of the application data structure.

This paper describes the design and implementation of a toolkit for applications with parallel loops. In Section 2, we look at the related work done for designing dynamic load balancing tools targeted to specific group of applications. In section 3, we review the load balancing tool, `LBtool` [4] and further develop it into a dynamic load balancing library, `LB_Migrate`. In Section 4 an analytical description of the performance evaluation is given. Section 5 gives results of preliminary performance tests of the toolkit. Section 6 concludes with possible future work.

2. **Related work**

There are a number of efforts being made to address dynamic load balancing issues. These efforts can be distributed into two categories depending upon whether the data migration is carried out by the application or by the library. A class of dynamic load balancing libraries like DRAMA [6] is specific to certain applications which are graph based, so the library can perform data migration. Another class of libraries like Zoltan [1][2] are designed to be data structure neutral making them suitable to large number of applications but cannot perform data migration.

The DRAMA [6] project created a dynamic load balancing library specifically for parallel unstructured finite element applications with changing work load and communication requirements. The library consists of dynamic mesh re-partitioning algorithms. These algorithms must interact with the application program, take inputs from the application program consisting of the current distributed mesh, calculation and communication cost parameters and time. Based on the output of the mesh re-partitioning, the application must decide whether the gain of the re-distribution will outweigh the cost of data migration and if so, migrate the application data according to the new partitioning.

Aiming to a wider range of applications the Sandia National Laboratory developed a dynamic load balancing library called Zoltan [1][2] . Zoltan simplifies load balancing, data movement, unstructured communication and memory usage difficulties that arise in dynamic applications such as adaptive finite-element methods, particle simulations and crash simulations. The Zoltan library is a collection of data management services for unstructured, adaptive and dynamic applications [2].

Data migration involves gathering data from an overloaded processor and sending it to an underloaded processor, removing it from the original processor. A general purpose load balancing library like Zoltan cannot perform all the operations for data migration. However, it assists an application with the communication required in data migration. Zoltan knows where data must be sent to establish the new decomposition and perform all the communication using the communication tools needed within the library. Then, the application must specify means of gathering data and migrating it into a new processor's data structure.

To cater to applications with computationally intensive parallel loops, there is a general-purpose tool `LBtool` [4] that has been developed at Mississippi State University. The tool is based on the MPI library and is suitable when the loop data is already partitioned among the participating processes. The tool dynamically schedules the execution of chunks of loop iterates and directs the transfer of data between processes if needed to achieve better performance. Load balancing schemes based on dynamic loop scheduling such as factoring (FAC), fractiling (FRAC), adaptive weighted factoring, etc. have been used for scheduling loops on various processes. The tool works independently of any application and may be used to parallelize sequential application with parallel loops or as an alternate load balancing strategy for

existing parallel application. A drawback of `LBtool` is the necessity for the user to provide routines for migrating data and results. We address this drawback in the next section

3. LB_Migrate: the dynamic load balancing library

Based on the load balancing tool, `LBtool`, we propose a dynamic load balancing library called `LB_Migrate`. The library is designed for large scientific applications with parallel loops. The library assumes that the application uses MPI and the application data is stored in arrays. The data may be partitioned among the processors or replicated on all processors. The library uses dynamic loop scheduling to execute iterates on different processors. Along with scheduling it also performs data migration. The library not only makes the application totally orthogonal of scheduling and load balancing but also provides more flexibility like the ability to choose different load balancing strategies.

3.1 Load balancing schemes

`LB_Migrate` is based on the Master/Slave approach. One of the processor is designated as, the master or the scheduler processor which assigns chunks of iterates to other processors, keeping track of their performance, directing data movement if required, and detecting loop completion. All processors including the master processor participate in executing chunks of loop iterate. Thus the overall library schedules work and then migrate data to minimize the load imbalance.

Load balancing in `LB_Migrate` uses on loop scheduling schemes like factoring [8] (FAC), fractiling [9] (FRAC), weighted factoring [10] (WF), adaptive weighted factoring [11] (AWF), adaptive factoring [11] (AF). Based on probabilistic analysis, these schemes schedule the execution of iterates in chunks with decreasing sizes. The chunk sizes are determined during loop runtime, such that chunks have a high probability of being completed before the optimal time. In addition `LB_Migrate` also uses older non-adaptive schemes like static chunking (STAT), self-scheduling (SS) and fixed size chunking (FSC).

3.2 The data migration strategy

The `LB_Migrate` provides two data migration strategies the GIVE strategy [4] used in `LBtool` and the GET strategy(fig 1), giving the application developer the option to choose the most suitable migration strategy.

The communication and computation among the processors can be explained with the messages in Fig 1. The WRK_MSG message from the scheduler to other processors and itself gives the start and size of the chunk to be executed. The REQ_MSG message to the scheduler, from other processors is sent to request for more work when they are about to finish with their current chunk of data. The GET_MSG message is sent from the scheduler to the processor which has finished executing its own chunk of data. The message gives the source processor from where the data needs to be obtained and the start and size of data it needs to work on. The GIV_MSG message from the source processor to the faster or destination processor gives the start and size of the data to be worked upon. If the data distribution is distributed then the GIV_MSG also has a parameter asking the source processor to send the data to be worked on.

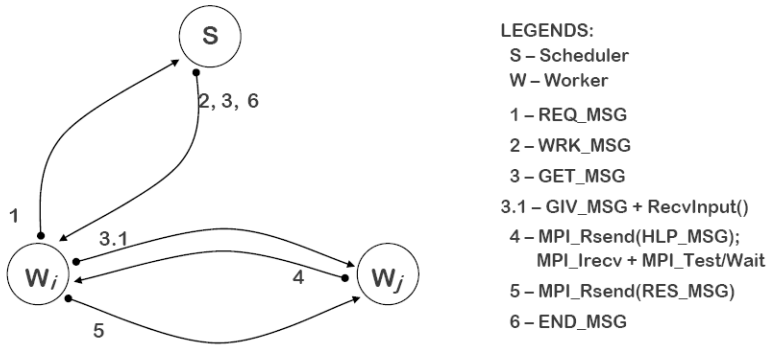

Fig 1: GET load balancing strategy

Once the migrated chunk of data has been worked on then the RES_MSG message is sent by the destination processor to send back the results. This message is sent to a source or overloaded processor if the data distribution is distributed. Otherwise it is sent to the scheduler, who broadcasts all the results once the END_MSG is send as signal termination.

```
main loop
{
…
 inType:: x[maxN];
outType:: y[maxN];
...
do {
  WorkProc (x[i], y[i], i)
  i++
  } while i<N
….

}
…

void WorkProc(inType inputs,
outType output , i)
     output[i] = f(input[i])
}
```

Application with parallel loop

```
main loop
{
…
inType::  x[maxN];
outType:: y[maxN];
myRank = ...

MPI_Init();
...
GetMyInputs (myRank, myShare, x)
…
LB_Migrate (WorkProc, xLoc,
Debug_level,
Data_migration_method, foreman,
Scheduling_method, minChunk,
breakAfter, requestWhen,
MPI_COMM_WORLD);
…
MPI_Finalize();
}
…

void WorkProc(inType inputs,
outType output, start, size) {
for (i=start, i<start+size-1,i++)
            outType[i] = f(inType(i))

}
```

Integration of LB_Migrate in the application

Fig 2: Pseudo code of the application with the parallel loop and with the library

3.3 The LB_Migrate parameters

The work parameter of the library is specified by the variable `WorkProc`, which encapsulates the routine that does all of the computation for the application. The library assumes that an initial distribution of data is available. If no such distribution is given, then a STATIC distribution is applied, where the parallel loops are equally divided among all the processors. Each processor executes `WorkProc` on its own data. Once each processor is done with its data, the data is migrated from other processors and the `WorkProc` is executed on them.

The data distribution parameter `xLoc` specifies whether the data is distributed or replicated in the application. If the data is distributed then during data migration, results are sent back to the source processor. However, if the data distribution is replicated, then every processor has complete data. In this case, during data migration the master processor needs to specify the start and size of the chunk of data to

the destination processor, rather than asking the source or slow processor to migrate the data. Once the results are computed, they are sent back to the master processor. The END_MSG message is issued to processors once all the results have been computed. Now, the master processor broadcasts computed results to all the processors.

The load balancing parameters of the library are specified by the variables `method`, `minChunk`, `breakAfter` and `requestWhen`. The application developer, or the user of the library, must specify values for these parameters. However, if it is not provided then the library determines the values heuristically. The `breakAfter` parameter specifies the number of iterates to be executed by a processor before it checks and responds for messages. If `breakAfter` is not given, then the processors check for messages *P* times in the course of executing a chunk (*2*P* if it is the scheduler). A processor sends the REQ_MSG to the scheduler when the number of iterates remaining in the current chunk falls below `requestWhen` for the first time. The processor will continue executing the rest of the chunk thus interleaving communication and computation.

The parameter `Debug_Level` is the debugging parameter specified by the library. The application developer can set this parameter at different levels which have predefined action to be taken. When the debug parameter is set to the lowest level, then no communication messages are printed out. However, error messages will be printed in case of errors. At its highest level, the debug parameter will print all the communication messages among the processors while scheduling and during data migration. The intermediate levels print out various communication messages, each level details are provide in table 1.

Level	Details
0	No output unless an error or warning
1	Trace all the SEND statements
2	Trace all the RECV statements
3	Trace all the SEND and RECV statements and statements made during data migration
4	Trace calls to the WORK_ROUTINE

Table 1: Debugger parameters and output description

The debugging parameter is strictly a tool for identifying the errors in computation and communication while using the library. It cannot interact with the application to detect errors in the application.

The `Data_mig_strategy` parameter gives flexibility to the application developer to make a choice between the data migration strategies, the GIVE strategy or the GET strategy. It has been observed in preliminary tests for simple data types and a small amount of data, the GIVE strategy works better than the GET strategy. However, for large applications involving large amounts of data migration and having user defined complex data types, the GET strategy works better.

Initially we dealt with applications which had single input and single output arrays, even the tool had been tested for such type of application. This limited the usage of the library to deal with applications having multiple input and output arrays. To overcome this problem, we adopted the flexible parameters for the `WorkProc`. All the input and output arrays, data types, starting addresses, and size of each element of each of the arrays were stored in arrays specified by the library parameters of `get_start` and `get_type`. Thus, the library works with reference to the data specified in the arrays independent of the application data type. All this provides the flexibility to work with a large number of applications for the library.

This is made possible by casting the application data as void pointer types and then passing to the library, thus making any data type referred to by its starting address and the size of the data to be transferred. The GIV_MSG uses the MPI send message with the void function specifying the size of the buffer with reference to the starting point of the data distribution and the parameter which specifies the size of each element of the input array. The GET_MSG receives the data and using the reference specified to processes the data using the `WorkProc` and results are sent back to the source processor or the scheduler, depending upon the data distribution.

4. Performance Evaluation

`LB_Migrate` performance is evaluated by empirical and analytical methods. Empirically, when the library is used to execute a parallel loop, the times utilized by each participating processor by the library `LB_Migrate` routines, and by the `WorkProc` are easy to collect. These times are denoted by t_r and u_r for processor r, respectively. Let

W = N, the amount of work, or the loop size and
$T_1 = \sum_{r=0}^{P-1} u_r$, the total useful work time for the loop.

Then, the following performance metrics can be computed: parallel time $T_p = max\{t_r\}$; $cost = P*T_p$; performance = W/ T_p, the ratio of total work to parallel time; the aggregate time used by the parallel system to execute the loop; *effectiveness* $= (W/ T_p)/(P*T_p)$, the ratio of performance to cost; *speedup* $= T_1/ T_p$ and *efficiency* $= (T_1/*T_p)/P$. When using these metrics to compare algorithms to solve the same problem on the same computation setup, the "better" algorithms are those with lesser T_p and cost, or those algorithms with greater performance, effectiveness, speedup and efficiency.

The load balancing overhead T_o is easy to compute by the usual formula, $T_o = P * T_p - T_1$. Analytically, the overhead depends on the number of chunks generated by the loop scheduling methods. A chunk triggers the arithmetic to compute the size, and the sending/receiving of messages: REQ_MSG and WRK_MSG, or REQ_MSG, GET_MSG, GIV_MSG, HLP_MSG and RES_MSG. Table 2 gives expressions for the number of chunks generated by most of the methods. Since T_o does not asymptotically exceeds the problem size N, the loop scheduling schemes are scalable.

Method	T_O	Method	T_O
STAT	$O(P)$	FAC	$O(P \log N)$
SS	$O(N)$	WF	$O(P \log N)$
FSC	$O(\frac{N}{K_{opt}})$	FRAC	$O(P \log N)$
GSS	$O(P \log \frac{N}{P})$	AWF	$O(P \log N)$

Table 2: Load balancing overhead

5. Experiments and Results

The library has been tested on an application that considers the problem of generating profiles of automatic quadrature routines (AQR). An AQR is designed to approximate an integral $I = \int_D f(x)dx$, where D is the domain of the integration, and $f(x)$ is the integrand. Here, x could be a single variable or a vector. Typically, the inputs to such a routine are: a description of the domain D, the code for the integrand $f(x)$, absolute and relative error tolerances (EPSA,EPSR), a limit to the number of function evaluations(MAXNFE) in case the error tolerances are not achievable by the AQR, and the quadrature rule (RULE) to be used. The AQR returns RESULT, hopefully satisfying $|RESULT - I| \leq max(EPSA, EPSR*|I|)$, an estimate ERREST of the absolute error in RESULT, the actual number of function evaluations used (NFE), and a termination condition indicator(IER). Since the AQR does not know the value of I, it attempt to satisfy $ERREST \leq max(EPSA, EPSR*|RESULT|)$.

The profile of an AQR is an empirical study which attempts to highlight the accuracies(EPSA,EPSR) achievable by the routine and the costs (NFE) involved, for various types of integrands $f(x)$. The integrands are chosen such that the answers are known analytically to facilitate the computation of the true error in RESULT, as well as the accuracy of the error estimate.As an application, profiling an AQR is embarrassingly parallel and computationally intensive. A high level description is given below in algorithm 1. The variables that determine the number of parallel tasks are as follows: NFAM- the number of integrand families, NDIF-the number of difficulty levels for each integrand family, DIF- the number of difficulty levels for each integrand family, NEPS- the number of relative accuracy requirements, NRUL- the number of quadrature rules to use and NSAMP- the number of samples to compute for each combination of integrand family, difficulty level, accuracy requirements, and quadrature rule. The total number of integrals to be evaluated is NFAM*NDIF*NEPS*NRUL*NSAMP. The

granularity of the tasks can be set form one integral per task (GRPSIZE=1) up to NSAMP integral per task (GRPSIZE=NSAMP). Due to the differences in integral families, difficult levels, accuracy requirements, and quadrature rule settings, the task execution times are highly variable even if the same number of integrals are evalauated per task.

Algorithm 1: Profiling a quadrature routine

```
Generate N,DATA(0…N-1)
do{
Evaluate integrals defined by DATA(index); compute
        accuracy and cost; accumulate some statistics
}
while (index < N)
Output Statistics
```

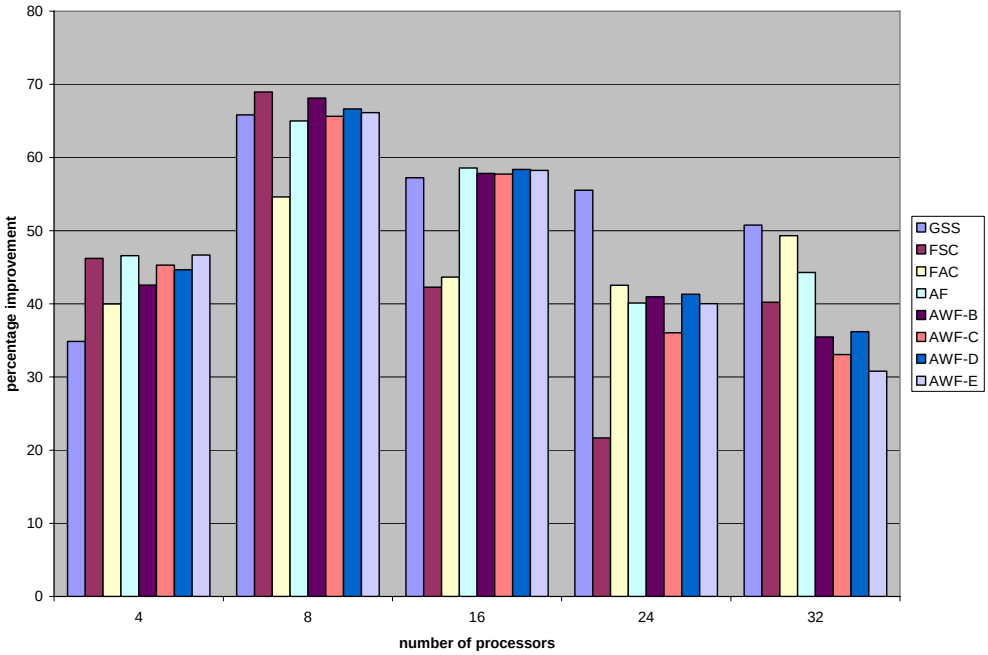

Fig 3: Percentage improvement in cost with load balancing for family 6, N=1080

The loop sizes (N) were 1040, 2600, 5200 and 10400 while the processor counts (P) were 4, 8, 16, 24 and 32. The data consists of 8 different families. The test runs were performed on the Linux cluster of the Mississippi State University Engineering Research Center. The cluster has 1038 processors consisting of 225 IBM x330 compute nodes, each with dual 1.266 GHz Intel Pentium III processors and 1.25 GB of RAM; 128 IBM x330 compute nodes, each with dual 1 GHz Intel Pentium III processors and 1 GB of RAM; and 1 SGI 1200 head node with dual 800MHz Intel Pentium III processors and 1 GB of RAM. The cluster runs the Red Hat Linux operating system and is connected via fast Ethernet switched with gigabit Ethernet uplinks. The cluster queuing nodes to a job, but this is not guaranteed. Other jobs are also running in the cluster along with the test, thus network traffic volume may have varied from test to test. In order to reduce timing measurement biases, each test was repeated three times; the t_r and u_r used in the performance evaluation of the tool are averages of the three runs.

Figure 3 illustrates the percent improvement in the cost that is obtained on family 6 when the loop type has a Back distribution. The percentage improvement is computed as 100 * *(CostNoLB-CostLB)/CostNoLB* where *CostNoLB* is the cost when there is no load balancing (i.e. method is STAT) and *CostLB* is the cost with load balancing. The figure demonstrates that up to 68% cost reduction is achieved using the adaptive loop scheduling methods.

6. Conclusion and Future Work

We have developed a load balancing library catering to large scientific applications that have parallel loops and utilize MPI. The library provides more flexibility to the load balancing tool, `LBtool`. Preliminary tests on a Linux cluster indicate that the cost of executing a simulated irregular parallel loop without load balancing can be reduced by up to 68% when using the library. It is also been observed that the adaptive techniques perform better than the non-adaptive techniques irrespective of number of processors.

The library has been tested upon simulated application of AQR and current tests are being performed for QTM application in Fortran 90 and on graphics and visualization computationally intense applications like Mandelbrot plot . The future work proposed for the library is to make it data structure neutral and at the same time be able to perform data migration.

References:

[1] B. Hendrickson, and K. Devine, "Dynamic Load balancing in Computational Mechanics," Comp. Methods Appl. Mech. Engg., 184(2000), pp 485-500.

[2] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan , "Zoltan Data Management Services for Parallel Dynamic Applications," *Computing in Science and Engineering*, Vol. 4, No. 2, Mar/Apr 2002, pp. 90-97.

[3] K. Devine, B. Hendrickson, E. Boman, M. St.John, and C. Vaughan. ``Design of Dynamic Load-Balancing Tools for Parallel Applications." *Proceedings of the International Conference on Supercomputing*, Santa Fe, May, 2000.

[4] R. L. Carino, and I. Banicescu, "A load balancing tool for distributed parallel loops," *Proc. International Workshop on Challenges of Large Applications in Distributed Environments*" Seattle WA, June 2003, pp. 39-46.

[5] B.Hendrickson, and R.Leland "The Chaco user's guide, version 2.0,"tech.rep.SAND94-2692, Sandia National Laboratories, Albuquerque, New Mexico,1994.

[6] B.Maerten, D.Roose, A.Basermann, J.Fingberg and G.Lonsdate, " DRAMA: A library for parallel dynamic load balancing of finite element application, " Proceeding Euro-Par '99,1999, pp 313-316.

[7] G.Karypis, and V.Kumar,"ParMETHIS: parallel graph partitioning and sparse matrix ordering library ," tech rep. CORR 95-035, Department of Computer Science, Univ. of Minnesota, 1997.

[8] S.F. Hummel, E. Schonber, and L. E. Flynn, "Factorng: A Method for Scheduling Parallel Loops," *Communications for the ACM*,vol. 35 no.8, 1992, pp90-101.

[9] I. Banicescu, and S. F. Hummel, "Balancing Processor Loads and Exploitng Data Locality in N-body Simulations," *Proceedings for Supercomputing '95*,San Diego California, 1995.

[10] S. F. Hummel, J. Schmidt, R. N. Uma, and J. Wein, "Load-sharing in Heterogenous Systems via Weighted Factoring," *Proceedings of Symposium on Parallel Algorithms and Architecture*, Padua, Italy, 1996,pp318-328.

[11] I. Banicescu, and V. Velusay, "Load Balancing Highly Irregular Computation with the Adaptive Factoring," *Proceedings of the IEEE- International Parallel and Distributed Processing Symposium 2002-Heterogeneous Computing Workshop*, Fort Lauderdale, Florida, 2002.

[12] R. L. Carino, *Numerical Integration Over Finite Regions Using Extrapolation by Non-linear Sequence Transformations,* doctoral dissertation, La Trobe University Australia, May 1992.