Approaches to Architecture-Aware Parallel Scientific Computation

James D. Teresco*[*], *Joseph E. Flaherty*[†], *Scott B. Baden*[‡], *Jamal Faik*[§], *Sébastien Lacour*[¶], *Manish Parashar*[‖], *Valerie E. Taylor*[**] *and Carlos A. Varela*[††]

Modern parallel scientific computation is being performed in a wide variety of computational environments that include clusters, large-scale supercomputers, grid environments, and metacomputing environments. This presents challenges for application and library developers, who must develop architecture-aware software if they wish to utilize several computing platforms efficiently.

Architecture-aware computation can be beneficial in single-processor environ-

[*]Department of Computer Science, Williams College, Williamstown, MA 01267 USA, (terescoj@cs.williams.edu).

[†]Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY 12180 USA, (flaherje@cs.rpi.edu).

[‡]CSE Department, UC San Diego, USA, (baden@cs.ucsd.edu).

[§]Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY 12180 USA, (faikj@cs.rpi.edu).

[¶]IRISA/INRIA, Rennes, France, and Northern Illinois University, USA, (Sebastien.Lacour@irisa.fr).

[‖]The Applied Software Systems Laboratory, Department of Electrical and Computer Engineering, Rutgers, The State University of New Jersey, Piscataway, NJ 08855 USA, (parashar@caip.rutgers.edu).

[**]Department of Computer Science, Texas A&M University, College Station, TX 77842 USA, (taylor@cs.tamu.edu).

[††]Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY 12180 USA, (cvarela@cs.rpi.edu).

ments. It takes the form of something as common as an optimizing compiler, which will optimize software for a target computer. Application and library developers may adjust data structures or memory management techniques to improve cache utilization on a particular system [21].

Parallel computation introduces more variety, and with that, more need and opportunity for architecture-specific optimizations. Heterogeneous processor speeds at first seem easy to account for by simply giving a larger portion of the work to faster processors, but assumptions of homogeneous processor speeds may be well-hidden. If all processor speeds are the same, differences between uniprocessor nodes and symmetric multiprocessing (SMP) nodes may be important. Computational and communication resources may not be dedicated to one job, and the external loads on the system may be highly dynamic and transient. Interconnection networks may be hierarchical, leading to non-uniform communication costs. Even if targeting only homogeneous systems, the relative speeds of processors, memory, and networks may affect performance. Heterogeneous processor architectures (e.g., Sparc, x86) present challenges for portable software development and data format conversions. Some operating systems may provide support for different programming paradigms (e.g., message passing, multithreading, priority thread scheduling, or distributed shared memory). Resources may also be transient or unreliable, breaking some common assumptions in, e.g., applications that use the Message Passing Interface (MPI) standard [37]. Finally, scalability is a concern, in that what works well for a cluster with dozens of processors will not necessarily work well for a supercomputer with thousands of processors, or in a grid environment [34] with extreme network hierarchies.

Many decisions that software developers can make may be affected by their target architectures. The choices can be algorithmic, such as when choosing a solution method that lends itself better to shared memory/multithreading or to distributed memory/message passing, as appropriate. The choice of parallelization paradigm affects portability and efficiency. The single-program multiple-data (SPMD) with message passing approach is often used because MPI is widely-available and highly portable. However, this portability may come at the expense of efficiency. Other options include shared memory/multithreading [16, 55], a hybrid of SPMD with multithreading [8], the actor/theater model [1], and the Bulk Synchronous Parallel (BSP) [63] model. Parallelization can be achieved by a "bag-of-tasks" master/worker paradigm, domain decomposition, or pipelining. Computation and/or communication can be overlapped or reordered for efficiency in some circumstances. A programmer may choose to replicate data and/or computation to eliminate the need for some communication. Small messages can be concatenated and large messages can be split to achieve an optimal message size, given the buffer sizes and other characteristics of a particular interconnect [58]. Communication patterns can be adjusted. The computation can be made to use an optimal number of processors, processes, or threads, given the characteristics of the application and of the computing environment [20]. Partitioning and dynamic load balancing procedures can make tradeoffs for imbalance *vs.* communication minimization, or can adjust optimal partition sizes, and can partition to avoid communication across the slowest interfaces [29, 84].

Any architecture-aware computation must have knowledge of the computing environment, knowledge of software performance characteristics, and tools to make use of this knowledge. The computing environment may come from a manual specification or may be discovered automatically at run time. Computing environment performance characteristics can be discovered through *a priori* benchmarking, or by dynamic monitoring. Software performance characteristics can be based on performance models or on studies that compare performance.

Software can use such knowledge of the computing environment at any of a number of the common levels of abstraction. Compiler developers and low-level tool developers (e.g., MPI implementers) can make architecture-aware optimizations that are applicable to a wide range of applications. Other tool developers, such as those designing and implementing partitioners and dynamic load balancers or numerical libraries, can make their software architecture aware and benefit all users of the libraries. Middleware systems can make architecture-aware adjustments to computations that use them. Application programmers can make high-level decisions in an architecture-aware manner, e.g., through their choice of programming languages and parallel programming paradigm, by adjusting memory management techniques, or by adjusting the parameters and frequency of dynamic load balancing.

This paper describes several efforts that were presented at a minisymposium on architecture-aware parallel computing at the Eleventh SIAM Conference on Parallel Processing for Scientific Computing (San Francisco, 2004). The first approach, the Prophesy framework by Taylor, et al., analyzes the performance of applications running in parallel and distributed environments (Section 1). Section 2 describes Baden's work on canonical variant programming and on computation and communication scheduling. Next, the work of Lacour, et al., on topology-aware collective communication in the grid-enabled MPI implementation, MPICH-G2, is described (Section 3). Dynamic load balancing for heterogeneous and hierarchical systems is described next, including work by Faik, et al. (Section 4), Teresco, et al. (Section 5), and Parashar, et al. (Section 6). Finally, Varela's approach to "worldwide computing" shows how a middleware layer can help manage a computation in a widely distributed and highly dynamic and transient computing environment (Section 7).

1 Prophesy: A Performance Analysis and Modeling System for Parallel and Distributed Applications

[1]

Today's complex parallel and distributed systems require tools to gain insight into the performance of applications executed on such environments. This section presents the web-based Prophesy system[2] [95, 96], a performance analysis and modeling infrastructure that helps to provide this needed insight. Prophesy automatically instruments application software, records performance data, system features and application details in a performance database, and provides automated

[1] Primary section author: Taylor, with Xingfu Wu and Rick Stevens

[2] `http://prophesy.cs.tamu.edu`

modeling techniques to facilitate the analysis process. Prophesy can be used to develop models based upon significant data, identify the most efficient implementation of a given function based upon the given system configuration, explore the various trends implicated by the significant data, and predict software performance on a different system.

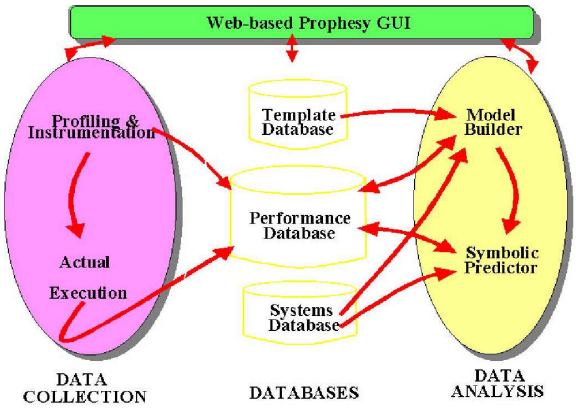

Figure 1. *Prophesy framework.*

Prophesy consists of three major components: data collection, data analysis, and central databases (Figure 1). The data collection component focuses on the automated instrumentation and application code analysis at the granularity levels of basic blocks, procedures, and functions. Execution of the instrumented code gathers a significant amount of performance information for automated inclusion in the performance database. Manual entry of performance data is also supported. Performance data can then be used to gain insight into the performance relationship among the application, hardware, and system software.

An application goes through three stages to generate an analytical performance model: (*i*) instrumentation of the application, (*ii*) performance data collection, and (*iii*) model development using optimization techniques. These models, when combined with data from the system database, can be used by the prediction engine to predict the performance in a different computing environment. The Prophesy infrastructure is designed to explore the plausibility and credibility of various techniques in performance evaluation (e.g., scalability, efficiency, speedup, performance coupling between application kernels, etc.) and to allow users to use various metrics collectively to bring performance analysis environments to the most advanced level.

The Prophesy database must accommodate queries that lead to the development of performance models, allow for prediction of performance on other systems, and allow for one to obtain insight into methods to improve the performance of the application on a given distributed system. Hence, the database must facilitate the following query types:

- Identify the best implementation of a given function for a given system configuration (identified by the run-time system, operating system, processor organization, etc.). This can be implemented by querying the database for comparison of performance data on different systems.
- Use the raw performance data to generate analytical (nonlinear or linear) models of a given function or application; the analytical model can be used to extrapolate the performance under different system scenarios and to assist programmers in optimizing the strategy or algorithms in their programs.
- Use the performance data to analyze application-system trends, such as scalability, speedup, I/O requirements, communication requirements, etc. This can be implemented by querying the database to calculate the corresponding formula.
- Use the performance data to analyze user-specific metrics such as coupling between functions.

The Prophesy database has a hierarchical organization, consistent with the hierarchical structure of applications. The entities in the database are organized into four areas: (*i*) Application information. This includes entities that give the application name, version number, a short description, owner information and password. Data are placed into these entities when a new application is being developed. (*ii*) Executable information. This includes all of the entities related to generating an executable of an application. These include details about compilers, libraries and the control flow, and are given for modules and functions. Data are placed into these entities when a new executable is generated. (*iii*) Run information. This includes all of the entities related to running an executable. These are primarily details about the program inputs and the computing environments used. Data are placed into these entities for each run of a given executable. (*iv*) Performance statistics information. This includes all of the entities related to the raw performance data collected during execution. Performance statistics are collected for different granularities (e.g., application, function, basic unit, and data structure performance).

The Prophesy automated model builder automatically generates performance models to aid in performance analysis and evaluation of a given application or execution environment. Prophesy supports two well-established modeling techniques: curve fitting and parameterization, plus a composition method developed by the Prophesy research group [79, 80]. Curve Fitting uses optimization techniques to develop a model. The model builder uses a least squares fit on the empirical data in the Prophesy database specified by the user to generate the model. The models it generates are generally a function of some input parameters of the application and the number of parameters. The system performance terms are clustered together with the coefficients determined by the curve fitting; such parameters are not exposed to the user. The advantage of this method is the fact that only the empirical data is needed to generate the models; no manual analysis is required. The parameterization method combines manual analysis of the code with system performance measurements. The manual analysis requires hand-counting the number of different operations in kernels or functions that are generally 100 lines of

code or less. The manual analysis is used to produce an analytical equation with terms that represent the application and the execution environment, allowing users to explore different application and execution environment scenarios with parameterized models. The manual analysis step is the only drawback, but this step is done only once per kernel. The composition modeling technique attempts to represent the performance of an application in terms of its component kernels or functions. These kernel performance models are combined to develop the full application performance models. It is extremely useful to understand the relationships between the different functions that compose the application, determining how one kernel affects another (i.e., whether it is a constructive or a destructive relationship). Further, this information should be able to be encapsulated into a coefficient that can be used in a performance model of the application. In [79], the advantages of using the coupling values to estimate performance are demonstrated using the NAS Parallel Benchmarks [10]. For BT (Block Tridiagonal) dataset A, the four kernel predictor had an average relative error of 0.79%, while merely summing the times of the individual kernels resulting in an average relative error of 21.80%.

Prophesy includes automatic instrumentation of applications, a database to hold performance and context information, and an automated model builder for developing performance models, allowing users to gain needed insights into application performance based upon their experience as well as that of others. Current research is focused on extending the tool to different application communities.

2 Canonical Variant Programming and Computation and Communication Scheduling

[3]

Application performance is sensitive to technological change, and an important factor is that the cost of moving data is increasing relative to that of performing computation. This effect is known as the "memory wall." A general approach for desensitizing performance to such change remains elusive. The result can be a proliferation of program variants, each tuned to a different platform and to configuration-dependent parameters. These variants are difficult to implement owing to an expansion of detail encountered when converting a terse mathematical problem description into highly tuned application software.

Ideally, there would exist a *canonical program variant*, from which all *concrete* program variants unfold automatically, altering their behavior according to technological factors affecting performance. There has been some progress in realizing the notion of canonical program variants through "self tuning software." Self tuning software has proven highly successful in managing memory hierarchy locality and includes packages such as ATLAS [93], PhiPac [12], and FFTW [35]. The general approach is to generate a search space of program variants, and to solve an optimization problem over the search space. The crucial problem is how to optimize the search space. Architecture cognizant divide and conquer [36] explored the notion of separators for pruning search trees; these enable different levels of the mem-

[3] Primary section author: Baden

ory hierarchy to be optimized separately. A related approach is to customize the source code using semantic level optimizations, including telescoping languages [51], Broadway [38], and ROSE [70]. Lastly, DESOBLAS takes a different approach: it performs delayed evaluation using task graphs [57].

Another manifestation of the memory wall is the increased cost of interprocessor communication in scalable systems. Reformulating an algorithm to tolerate latency is a difficult problem owing to the need to employ elaborate data decompositions and to solve a scheduling problem. Because an overlapped algorithm requires that communication and computation be treated as simultaneous activities, communication must be handled asynchronously [7, 9, 31, 76]. The resultant split phase algorithms are ad-hoc and prone to error [8], even for the experienced programmer, and require considerable knowledge about the application. The resultant difficulties have led to alternative actor-based models of execution including: Mobile Object Layer [23], Adaptive MPI [41], and Charm++ [45]. These models support shared objects with asynchronous remote method invocation. Data motion is implicit in method invocation. Other relevant work includes DMCS [22], which supports single sided communication and active messages, and SMARTS, which uses affinity to enhance memory locality by scheduling related tasks "back to back" [87].

A new project called *Tarragon* has been started at UCSD. Tarragon supports a non-bulk-synchronous, actor model of execution and is intended to simplify communication tolerant implementations. As with the other actor-based models, Tarragon employs data driven execution semantics [43], e.g., coarse grain dataflow [6], to manage communication overlap under the control of a scheduler. The data-driven model is attractive because it does not require the programmer to hardwire scheduling decisions into application code in order to manage communication overlap. As with traditional data flow [5, 25, 44] parallelism arises among tasks which are independent. Interdependent tasks are enabled according to the flow of data among them. A scheduler – rather than the application – determines an appropriate way to order computations.

A Tarragon Task Graph is interpreted as a logical grouping of an iteration space, along with a partial ordering of that grouping. The tasks are not objects as in Charm++ or Mobile Object Layer, but an abstract description of computation to be carried out. This abstraction enables Tarragon to realize pipelining optimizations across time-stepped simulations and to capture elaborate computational structures such as timestepped adaptive mesh hierarchies, and distinguishes it from other actor-based models. Tarragon also differs in another fundamental way. Whereas the other models support shared objects with asynchronous remote method invocation, Tarragon does not support shared objects, and methods may be invoked only locally. Data motion is explicit, reflecting the Tarragon philosophy of exposing such expensive operations to the programmer.

Tarragon supports the notion of *parameterized scheduling*, which has the property that the scheduler can read attributes decorating the task graph. These attributes come in the form of *performance meta-data*, a concept which as been explored jointly with Kelly and others in the context of cross-component optimization [50]. Performance meta-data may represent a variety of quantities, e.g., affinity, priority or other metrics. The programmer is free to interpret the meaning

of meta-data, while the scheduler examines their relative magnitudes in order to make decisions. Parameterized scheduling differs from application-level scheduling because application-dependent behavior can be expressed via meta data alone, i.e., without having to change the scheduler. The flexibility offered by parameterized scheduling significantly enhances the capability to explore alternative scheduling policies and metrics.

As with Charm++ and other efforts [82] Tarragon virtualizes computations. That is, the workload is split such that each processor obtains many pieces of work. Early results with a 3D elliptic solver using red-black relaxation have been positive, with only modest overheads observed for virtualization factors of up to nearly an order of magnitude [72]. Virtualization enhances the ability to overlap communication via pipelining.

Tarragon is currently under development, and is being applied to a variety of applications. We are also investigating compiler support for Tarragon using the ROSE [70] compiler framework developed by Quinlan. ROSE is a tool for building source-to-source translators realizing semantic level optimizations of C++ class libraries, and can extract semantic information from libraries such as Tarragon. The resultant software will in effect be a domain specific language.

3 A Multi-Level Topology-Aware Approach to Implementing MPI Collective Operations for Computational Grids

[4]

Computational grids have a potential to yield a huge computational power. Their utilization has been made a reality by grid access middleware like the Globus Toolkit[5], so more and more computational grids get deployed, as exemplified by such projects as NASA IPG, European DataGrid, NSF TeraGrid[6]. In a typical grid, several sites are interconnected over a Wide-Area Network (WAN). Within each site, a number of computers are connected over a Local-Area Network (LAN). Some of those computers may be gathered in clusters equipped with a very high-performance network like Myrinet. Thus, computational grids raise many issues, like *heterogeneity* in terms of computing resources and network links. As a grid is made of geographically distributed resources, possibly spread across continents, grid *networks* are inherently *multi-layered*, showing large network performance gaps (bandwidth, latency) between every communication network level.

In this context, some MPI applications need to achieve high performance. To reach that goal, an efficient MPI implementation must take into account the multi-layered nature of the grid network. This is particularly true for the implementation of MPI collective operations like `MPI_Bcast`. Those functions involve several processes running on a number of computers interconnected over various networks with different performance characteristics.

[4] Primary section author: Lacour, with Nicholas Karonis and Ian Foster

[5] `http://www.globus.org`

[6] `http://www.teragrid.org/`

Topology-*unaware* implementations of broadcast often make the simplifying assumption that the communication times between all process pairs are equal. Under this assumption, the broadcast operation is often implemented using a *binomial tree*. In the example of Figure 2, the broadcast operation from process 0 to nine other processes is completed in only four steps. This implementation is efficient in terms of performance and load balancing as long as it is used within a *homogeneous* network with uniform performance.

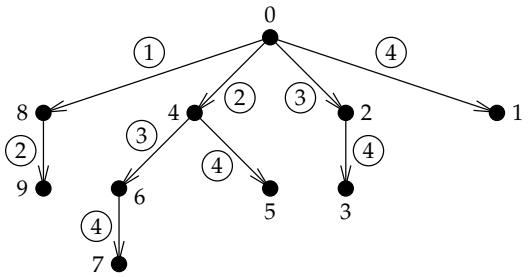

Figure 2. *Broadcast using a binomial tree: processes are numbered from 0 (root) through 9, communication steps are circled.*

If the ten processes are split into two clusters (processes 0 through 4 on one cluster and processes 5 through 9 on another), then a topology-*unaware* implementation of broadcast incurs three inter-cluster messages over a lower performance network (Figure 3). Existing two-level topology-aware approaches [42, 52] cluster

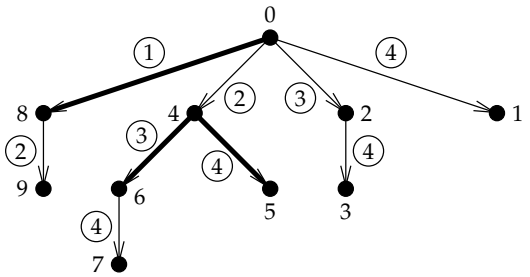

Figure 3. *Topology*-unaware *broadcast using a binomial tree: 3 inter-cluster messages (bold arrows) and 6 intra-cluster messages.*

computers into groups. In Figure 4, the root of the broadcast first sends the message to process 5 in the remote cluster, then processes 0 and 5 broadcast the message within their respective clusters using a binomial-tree algorithm. This solution performs all inter-cluster messaging first while also minimizing inter-cluster messaging.

A computational grid like the one described above typically involves multiple

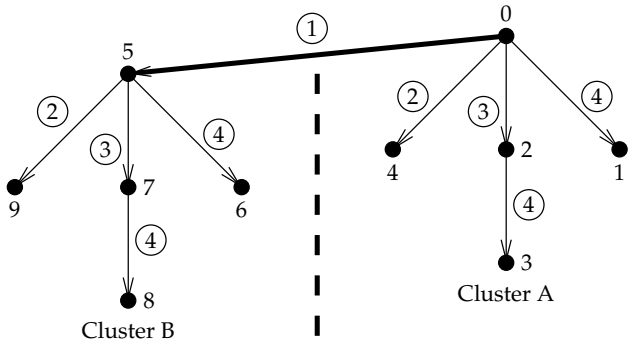

Figure 4. *Topology*-aware *broadcast: only one inter-cluster message (bold arrow).*

sites and may also include multiple clusters at a single site. The NSF TeraGrid, for example, has a 32-bit cluster and a 64-bit cluster both located at Argonne National Laboratory. Such grids induce three or more network levels (i.e., wide-area, local area, and intra-cluster) with different network characteristics at each level. In these grids, if the processes are grouped by clusters, the two-level topology-aware approach will not minimize the number of inter-site messages. If, on the other hand, the processes are instead grouped by sites, the two-level approach will not minimize the number of inter-cluster messages within each site. A *multi-level* strategy is needed, grouping the processes first by sites, and then by clusters in which both inter-site and inter-cluster messages are minimized.

Early performance evaluations [46] of the multi-level approach have shown significant performance gains for the broadcast operation. Encouraged by those results, eleven of the fourteen collective operations of the MPI-1 standard have been implemented in a multi-level topology-aware manner in MPICH-G2[7] [47].

4 Dynamic Load Balancing for Heterogeneous Environments

[8]

An attractive feature of clusters is the ability to expand their computational power incrementally by incorporating additional nodes. This expansion often results in heterogeneous environments, as the newly-added nodes and interconnects often have superior capabilities. This section focuses on the Dynamic Resource Utilization Model (DRUM)[9] [29], which is a software library that provides support for scientific computation in heterogeneous environments. The current focus is on providing information to enable resource-aware partitioning and dynamic load balancing procedures in a manner that incurs minimal effort to set up, requires very

[7] `http://www.globus.org/mpi/`
[8] Primary section authors: Faik, Teresco, and Flaherty, with Luis G. Gervasio
[9] `http://www.cs.williams.edu/drum`

few changes to applications or to dynamic load balancing procedures, and adjusts dynamically to changing loads while introducing only small overheads.

A number of recent papers have addressed these and similar issues. Minyard and Kallinderis [60] monitor process "wait times" to assign element weights that are used in octree partitioning. Walshaw and Cross [91] couple a multilevel graph algorithm with a model of a heterogeneous communication network to minimize a communication cost function. Sinha and Parashar [75] use the Network Weather Service (NWS) [94] to gather information about the state and capabilities of available resources; they compute the load capacity of each node as a weighted sum of processing, memory, and communications capabilities.

DRUM incorporates aggregated information about the capabilities of the network and computing resources composing an execution environment. DRUM can be viewed as an abstract object that (*i*) encapsulates the details of the execution environment, and (*ii*) provides a facility for dynamic, modular and minimally-intrusive monitoring of the execution environment.

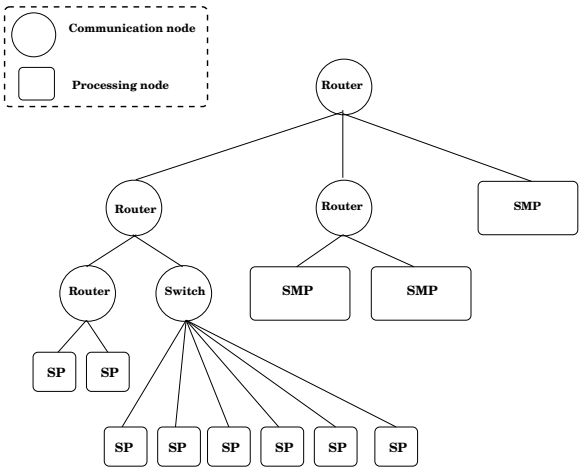

Figure 5. *Tree constructed by DRUM to represent a heterogeneous network.*

DRUM addresses hierarchical clusters (e.g., clusters of clusters, or clusters of multiprocessors) by capturing the underlying interconnection network topology (Figure 5). The tree structure of DRUM leads naturally to a topology-driven, yet transparent, execution of hierarchical partitioning (Section 5). The root of the tree represents the total execution environment. The children of the root node are high level divisions of different networks connected to form the total execution environment. Sub-environments are recursively divided, according to the network hierarchy, with the tree leaves being individual single-processor (SP) nodes or shared-memory multiprocessing (SMP) nodes. *Computation nodes* at the leaves of the tree have data representing their relative computing and communication power. *Network nodes*, representing routers or switches, have an aggregate power calculated as a function of the powers of their children and the network characteristics. The model of the

execution environment is created upon initialization based on an XML file that contains a list of nodes and their capabilities and a description of their interconnection topology. The XML file can be generated with the aid of a graphical configuration tool or may be specified manually.

Computational, memory and communication resource capabilities are assessed initially using benchmarks, which are run *a priori* either manually or using the graphical configuration tool. Capabilities may be updated dynamically by *agents*: threads that run concurrently with the application to monitor each node's communication traffic, processing load and memory usage. Network monitoring agents use the `net-snmp` library[10] or kernel statistics to collect network traffic information at each node. An experimental version that interfaces with NWS has also been developed. Processor and memory utilization are obtained using kernel statistics. The statistics are combined with the static benchmark data to obtain a dynamic evaluation of the powers of the nodes in the model.

DRUM distills the information in the model to a single quantity called the "power" for each node, which indicates the portion of the total load that should be assigned to that node. For load-balancing purposes, a node's power is interpreted as the percentage of overall load it should be assigned based on its capabilities. The power is computed as a weighted sum of processing and communication power, each of which are computed based on static benchmark and dynamic monitoring information.

DRUM has been used in conjunction with the Zoltan Parallel Data Services Toolkit [27, 28], which provides dynamic load balancing and related capabilities to a wide range of dynamic, unstructured and/or adaptive applications, to demonstrate resource-aware partitioning and dynamic load balancing for a heterogeneous cluster. Given power values for each node, any partitioning procedure capable of producing variable-sized partitions, including all Zoltan procedures, may be used to achieve an appropriate decomposition. Thus, any applications using a load-balancing procedure capable of producing non-uniform partitions can take advantage of DRUM with little modification. Applications that already use Zoltan can make use of DRUM simply by setting a Zoltan parameter, with no further changes needed.

We conducted an experimental study in which we used DRUM to guide resource-aware load balancing in the adaptive solution of a Laplace equation on the unit square, using Mitchell's Parallel Hierarchical Adaptive MultiLevel software (PHAML) [62]. Runs were performed on different combination of processors of a heterogeneous cluster. Figure 6 shows the relative change in execution time obtained when DRUM is used for different weights of the communication power. The ideal relative change is a theoretical value that would be obtained if partitions sizes perfectly match nodes processing capabilities and no inter-process communication takes place during the execution. The complete DRUM experimental study can be found in [29].

[10] `http://www.net-snmp.org`

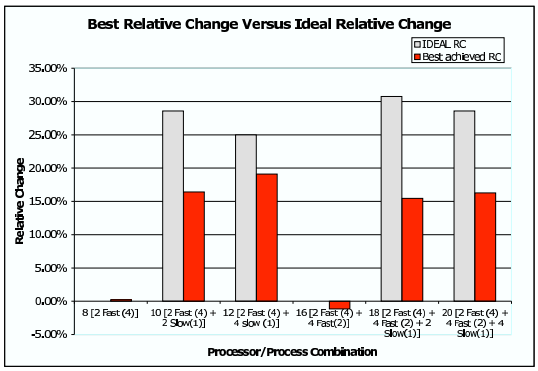

Figure 6. *Ideal and achieved (using DRUM) relative changes in execution times compared to homogeneous partitioning for an adaptive calculation using PHAML on different processor combinations.*

5 Hierarchical Partitioning and Dynamic Load Balancing

11

An effective partitioning or dynamic load balancing procedure maximizes efficiency by minimizing processor idle time and interprocessor communication. While some applications can use a static partitioning throughout a computation, others, such as adaptive finite element methods, have dynamic workloads that necessitate dynamic load balancing during the computation. Partitioning and dynamic load balancing can be performed using recursive bisection methods [11, 74, 81], space-filling curve (SFC) partitioning [15, 33, 61, 64, 66, 67, 68, 92] and graph partitioning (including spectral [69, 74], multilevel [14, 39, 48, 90], and diffusive methods [24, 40, 54]). Each algorithm has characteristics and requirements that make it appropriate for certain applications; see [13, 85] for examples and [83] for an overview of available methods.

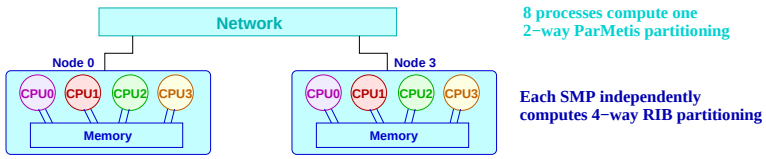

Figure 7. *Hierarchical balancing algorithm selection for two 4-way SMP nodes connected by a network.*

[11] Primary section author: Teresco

Modern clusters, supercomputers, and grid environments often include hierarchical interconnection networks. For hierarchical and heterogeneous systems, different choices of partitioning and dynamic load balancing procedures may be appropriate in different parts of the parallel environment. There are tradeoffs in execution time and partition quality (e.g., amount of communication needed, interprocess connectivity, strictness of load balance) [85] and some may be more important than others in some circumstances. For example, consider a cluster of symmetric multiprocessor (SMP) nodes connected by Ethernet. A more costly graph partitioning can be done to partition among the nodes, to minimize communication across the slow network interface, possibly at the expense of some computational imbalance. Then, a fast geometric algorithm can be used to partition to a strict balance, independently, within each node. This is illustrated in Figure 7. Such hierarchical partitionings of a 1,103,018-element mesh used in a simulation of blood flow in a human aorta are presented in [59].

Hierarchical partitioning and dynamic load balancing has been implemented in the Zoltan Parallel Data Services Toolkit [27, 28]. Using Zoltan, application developers can switch partitioners simply by changing a run-time parameter, facilitating comparisons of the partitioners' effect on the applications. Zoltan's hierarchical balancing implementation allows different procedures to be used in different parts of the computing environment [84]. The implementation utilizes a lightweight "intermediate hierarchical balancing structure" (IHBS) and a set of callback functions that permit an automated and efficient hierarchical balancing which can use any of the procedures available within Zoltan (including recursive bisection methods, space-filling curve methods and graph partitioners) without modification and in any combination. Hierachical balancing is invoked by an application the same way as other Zoltan procedures. Since Zoltan is data-structure neutral, it operates on generic "objects" and interfaces with applications through callback functions. A hierarchical balancing step begins by building an IHBS, which is an augmented version of the graph structure that Zoltan builds to make use of the ParMetis [49] and Jostle [90] libraries, using the application callbacks. The hierarchical balancing procedure then provides its own callback functions to allow existing Zoltan procedures to be used to query and update the IHBS at each level of a hierarchical balancing. After all levels of the hierarchical balancing have been completed, Zoltan's usual migration arrays are constructed and returned to the application. Thus, only lightweight objects are migrated internally between levels, not the (larger and more costly) application data. Zoltan's hierarchical balancing can be used directly by an application or be guided by the tree representation of the computational environment created and maintained by DRUM (Section 4).

Preliminary results applying hierarchical balancing to a parallel, adaptive simulation are promising [84]. A comparison of running times for a perforated shock tube simulation [32] on a cluster of SMPs shows that while ParMetis multilevel graph partitioning alone often achieves the fastest computation times, there is some benefit to using hierarchical load balancing where ParMetis is used for inter-node partitioning and inertial recursive bisection is used within each node. Hierarchical balancing shows the most benefit in cases where ParMetis introduces a larger imbalance to reduce communication.

Studies are underway that utilize hierarchical balancing on larger clusters, on other architectures, and with a wider variety of applications. Hierarchical balancing may be most beneficial when the extreme hierarchies found in grid environments are considered.

6 Autonomic Management of Parallel Adaptive Applications

12

Parallel structured adaptive mesh refinement (SAMR) [65, 66] techniques yield advantageous ratios for cost/accuracy compared to methods based on static uniform approximations, and offer the potential for accurate solutions of realistic models of complex physical phenomena. However, the inherent space-time heterogeneity and dynamism of SAMR applications coupled with a similarly heterogeneous and dynamic execution environment (such as the computational grid) present significant challenges in application composition, runtime management, optimization, and adaptation. These challenges have led researchers to consider alternate self-managing *autonomic* solutions, which are based on strategies used by biological systems to address similar challenges.

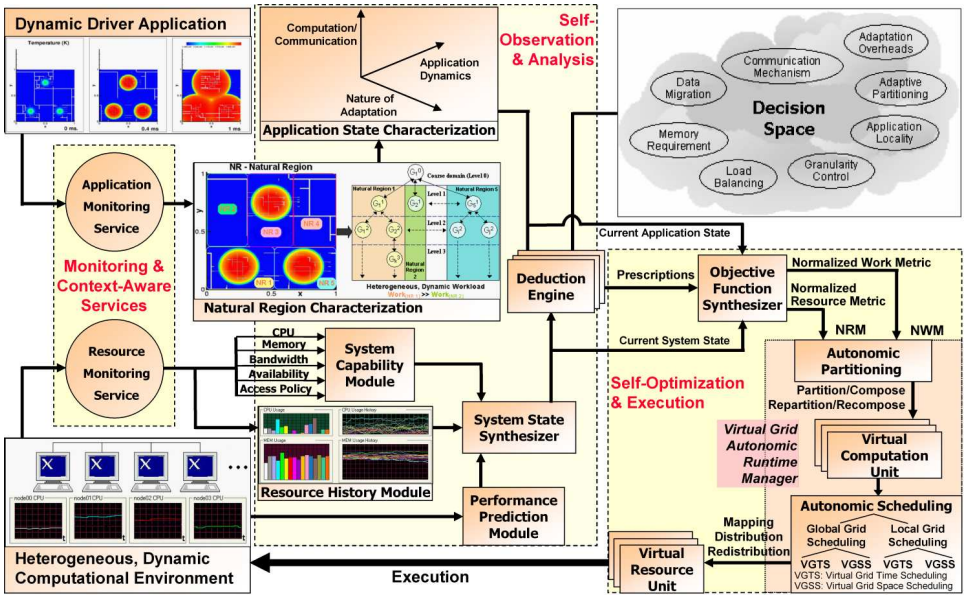

Figure 8. *Conceptual Overview of GridARM.*

GridARM [19] (Figure 8) is an autonomic runtime management framework that monitors application behaviors and system architecture and runtime state,

[12] Primary section author: Parashar

and provides adaptation strategies to optimize the performance of SAMR applications. The framework has three components: (*i*) services for monitoring resource architecture and application dynamics, and characterizing their current state, capabilities, and requirements; (*ii*) a deduction engine and objective function that define the appropriate optimization strategy based on runtime state and policies; and (*iii*) an autonomic runtime manager that is responsible for hierarchically partitioning, scheduling, and mapping application working-sets onto virtual resources, and tuning applications within the grid environment.

The monitoring/characterization component of the GridARM framework consists of embedded application-level and system-level sensors/actuators. Application sensors monitor the structure and state of the SAMR grid hierarchy and the nature of its refined regions. The current application state is characterized in terms of application-level metrics such as computation/communication requirements, storage requirements, activity dynamics, and the nature of adaptations [77]. Similarly, system sensors build on existing grid services such as NWS, and sense the current state of underlying computational resources in terms of CPU, memory, bandwidth, availability, and access capabilities. These values are fed into the system state synthesizer along with system history information and predicted performance estimates (obtained using predictive performance functions) to determine the overall system runtime state.

The current application and system state along with the overall "decision space" are provided as inputs to the GridARM deduction engine and are used to define the autonomic runtime objective function. The decision space is comprised of adaptation policies, rules, and constraints defined in terms of application metrics, and enables autonomic configuration, adaptation, and optimization. Application metrics include application locality, communication mechanism, data migration, load balancing, memory requirements, adaptive partitioning, adaptation overheads, and granularity control. Based on current runtime state and policies within the decision space, the deduction engine formulates prescriptions for algorithms, configurations, and parameters that are used to define the SAMR objective function.

The prescriptions provided by the deduction engine along with the objective function yield two metrics - normalized work metric (NWM) and normalized resource metric (NRM), which characterize the current application state and current system state respectively, and are the inputs to the autonomic runtime manager (ARM). Using these inputs, the ARM defines a hierarchical distribution mechanism, configures and deploys appropriate partitioners at each level of the hierarchy, and maps the application domain onto virtual computational units (VCUs). A VCU is the basic application work unit and may consist of computation patches on a single refinement level of the SAMR grid hierarchy or composite patches that span multiple refinement levels. VCUs are dynamically defined at runtime to match the natural regions (i.e., regions with relatively uniform structure and requirements) in the application, which significantly reduces coupling and synchronization costs.

Subsequent to partitioning, spatio-temporal scheduling operations are performed across and within virtual resource units (VRUs) using Global-Grid Scheduling (GGS) and Local-Grid Scheduling (LGS) respectively. During GGS, VCUs are hierarchically assigned to sets of VRUs, whereas LGS is used to schedule one or

more VCU within a single VRU. A VRU may be an individual resource (compute, storage, instrument, etc.) or a collection (cluster, supercomputer, etc.) of physical grid resources. A VRU is characterized by its computational, memory, and communication capacities and by its availability and access policy. Finally, VRUs are dynamically mapped onto physical system resources at runtime and the SAMR application is tuned for execution within the dynamic grid environment.

Note that the work associated with a VCU depends on the state of the computation, the configuration of the components (algorithms, parameters), and the current ARM objectives (optimize performance, minimize resource requirements, etc.). Similarly, the capability of a VRU depends on its current state as well as the ARM objectives (minimizing communication overheads implies that a VRU with high bandwidth and low latency has higher capability). The normalized metric NWM and NRM are used to characterize VRUs and VCUs based on current ARM objectives.

The core components of GridARM have been prototyped and the adaptation schemes within GridARM have been evaluated in the context of real applications such as the 3-D Richtmyer-Meshkov instability solver (RM3D) encountered in compressible fluid dynamics. Application aware partitioning [17] uses current runtime state to characterize the SAMR application in terms of its computation/-communication requirements, its dynamics, and the nature of adaptations. This adaptive strategy selects and configures the appropriate partitioner that matches current application requirements, thus improving overall execution time by 5 to 30% as compared to non-adaptive partitioning schemes. Adaptive hierarchical partitioning [56] dynamically creates a group topology based on SAMR natural regions and helps to reduce the application synchronization costs, resulting in improved communication time by up to 70% as compared to non-hierarchical schemes. Reactive system sensitive partitioning [75] uses system architecture and current state to select and configure distribution strategies and parameters at runtime based on the relative capacities of each node. This system sensitive approach improves overall execution time by 10 to 40%.

Proactive runtime partitioning [97] strategies are based on performance prediction functions and estimate the expected application performance based on current loads, available memory, current latencies, and available communication bandwidth. This approach helps to determine when the costs of dynamic load redistribution exceed the costs of repartitioning and data movement, and can result in a 25% improvement in the application recompose time. Architecture sensitive communication mechanisms [73] select appropriate messaging schemes for MPI non-blocking communication optimization suited for the underlying hardware architecture and help to reduce the application communication time by up to 50%. Workload sensitive load balancing strategy [18] uses bin packing-based partitioning to distribute the SAMR workload among available processors while satisfying application constraints such as minimum patch size and aspect ratio. This approach reduces the application load imbalance between 2 and 15% as compared to default schemes that employ greedy algorithms. Overall, the GridARM framework has been shown to significantly improve the performance of SAMR applications [19].

7 Worldwide Computing: Programming Models and Middleware

[13]

The Internet is constantly growing as a ubiquitous platform for high-performance distributed computing. This section describes a new software framework for distributed computing over large scale dynamic and heterogeneous systems. This framework wraps data and computation into *autonomous actors*, self organizing computing entities, which freely roam over the network to find their optimal target execution environment.

The actor model of concurrent computation represents a programming paradigm enforcing distributed memory and asynchronous communication [1]. Each actor has a unique name, which can be used as a reference by other actors. Actors only process information in reaction to messages. While processing a message, an actor can carry out any of three basic operations: alter its encapsulated state, create new actors, or send messages to peer actors. Actors are therefore inherently independent, concurrent and autonomous which enables efficiency in parallel execution [53] and facilitates mobility [3]. The actor model and languages provide a very useful framework for understanding and developing open distributed systems. For example, among other applications, actor systems have been used for enterprise integration [86], real-time programming [71], fault-tolerance [2], and distributed artificial intelligence [30].

The presented worldwide computing framework[14] consists of an actor-oriented programming language (SALSA) [89], a distributed run-time environment (WWC) [88], and a middleware infrastructure for autonomous reconfiguration and load balancing (IOS) [26] (see Figure 9). SALSA provides high-level constructs for coordinating concurrent computations, which get compiled into primitive actor operations, thereby raising the level of abstraction for programmers while enabling middleware optimizations without requiring the development of application-specific checkpointing, migration, or load balancing behavior.

Load balancing is completely transparent to application programmers. The IOS middleware triggers actor migration based on profiling application behavior and network resources in a decentralized manner. To balance computational load, three variations of random work stealing have been implemented: load-sensitive (LS), actor topology-sensitive (ATS), and network topology-sensitive (NTS) random stealing. LS and ATS were evaluated with several actor interconnection topologies in a local area network. While LS performed worse than static round-robin (RR) actor placement, ATS outperformed both LS and RR in the sparse connectivity and hypercube connectivity tests, by a full order of magnitude [26]. We are currently evaluating NTS in diverse heterogeneous grid environments.

The IOS software infrastructure naturally allows for the dynamic addition and removal of nodes from the computation, while continuously balancing the load given the changing resources. The ability to adapt applications to a dynamic network is

[13] Primary section author: Varela, with Travis Desell and Kaoutar El Maghraoui

[14] `http://www.cs.rpi.edu/wwc/`

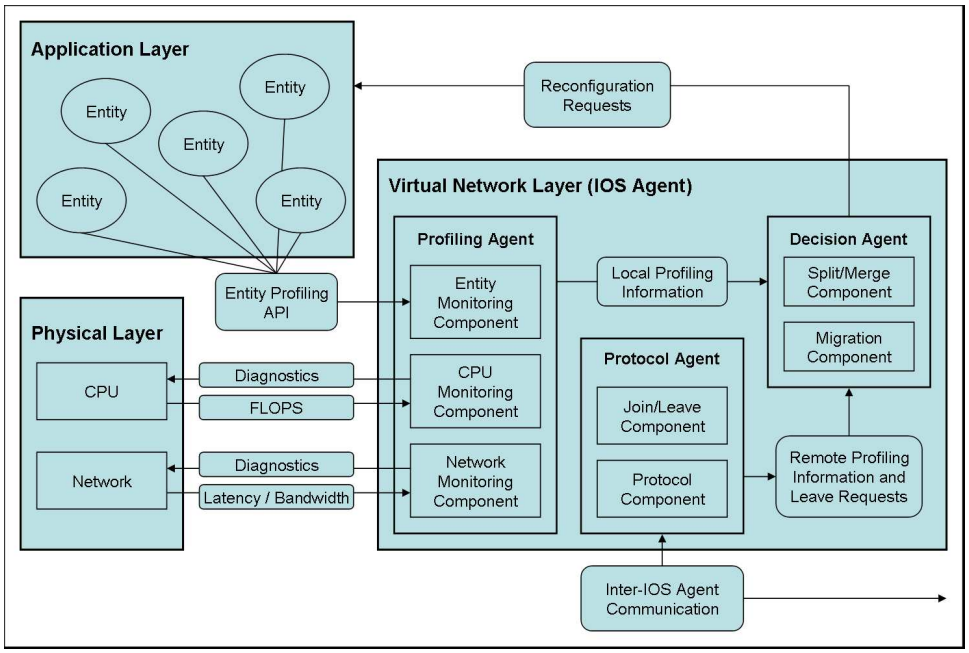

Figure 9. *A modular middleware architecture as a research testbed for scalable high-performance decentralized distributed computations*

critical upon node and network failures, common in Internet computing environments. This adaptability is also critical in shared environments with unpredictable variations in resource usage, e.g., by applications running concurrently on a grid.

Our current research focuses on resource management models and their middleware implementations for non-functional distributed systems behavior. Examples include data and process replication for fault tolerance, and split and merge behavior to dynamically optimize the granularity of computation and migration. We are particularly interested in the interaction of high-level programming abstractions and middleware optimizations [59]. While application-level load balancing may in general afford better performance, the simplicity of the autonomous actor programming model and the availability of computing power in large-scale dynamic networks may ultimately make optimization in middleware more beneficial for scientific computing [4, 78].

Acknowledgments

Teresco, Flaherty and Faik were supported by contract 15162 with Sandia National Laboratories, a multi-program laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000. Baden was supported by the Institute for Scientific Computing Research (DoE) and NSF contract ACI-0326013. Taylor was supported in

part by the National Science Foundation under NSF NGS grant EIA-9974960, a grant from NASA Ames, and two NSF ITR grants – GriPhyN and Building Human Capital. Parashar's work presented in this paper was supported in part by the National Science Foundation via grants numbers ACI 9984357 (CAREERS), EIA 0103674 (NGS), EIA-0120934 (ITR), ANI-0335244 (NRT), CNS-0305495 (NGS) and by DOE ASCI/ASAP via grant numbers PC295251 and 82-1052856. Varela's work was supported by a Rensselaer Seed Funding grant, two NSF grants (CISE MRI, CISE-RR), and two IBM Shared University Research (SUR) grants.

Bibliography

[1] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.

[2] G. Agha, S. Frølund, R. Panwar, and D. Sturman, *A linguistic framework for dynamic composition of dependability protocols*, in Dependable Computing for Critical Applications III, International Federation of Information Processing Societies (IFIP), Elsevier Science Publisher, 1993, pp. 345–363.

[3] G. Agha, N. Jamali, and C. Varela, *Agent Naming and Coordination: Actor Based Models and Infrastructures*, in Coordination of Internet Agents, A. Ominici, F. Zambonelli, M. Klusch, and R. Tolksdorf, eds., Springer-Verlag, 2001, ch. 9, pp. 225–248.

[4] G. Agha and C. Varela, *Worldwide computing middleware*, in Practical Handbook on Internet Computing, M. Singh, ed., CRC Press, 2004.

[5] Arvind and R. S. Nikhil, *Executing a program on the MIT tagged-token dataflow architecture*, IEEE Transactions on Computers, 39 (1990), pp. 300–318.

[6] I. Babb, R.G., *Parallel processing with large-grain data flow technique*, Computer, 17 (1984), pp. 55–61.

[7] S. B. Baden and S. J. Fink, *Communication overlap in multi-tier parallel algorithms*, in Proc. of SC '98, Orlando, Florida, November 1998.

[8] S. B. Baden and S. J. Fink, *A programming methodology for dual-tier multicomputers*, IEEE Transactions on Software Engineering, 26 (2000), pp. 212–216.

[9] S. B. Baden and D. Shalit, *Performance tradeoffs in multi-tier formulation of a finite difference method*, in Proc. 2001 International Conference on Computational Science, San Francisco, 2001.

[10] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow, *The NAS parallel benchmarks 2.0*, Tech. Report NAS-95-020, NASA Ames Research Center, 1995.

[11] M. J. Berger and S. H. Bokhari, *A partitioning strategy for nonuniform problems on multiprocessors*, IEEE Trans. Computers, 36 (1987), pp. 570–580.

[12] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, *Optimizing matrix multiply using phipac: A portable, high-performance, ANSI c coding methodology*, in International Conference on Supercomputing, 1997, pp. 340–347.

[13] E. Boman, K. Devine, R. Heaphy, B. Hendrickson, M. Heroux, and R. Preis, *LDRD report: Parallel repartitioning for optimal solver performance*, Tech. Report SAND2004–0365, Sandia National Laboratories, Albuquerque, NM, February 2004.

[14] T. Bui and C. Jones, *A heuristic for reducing fill in sparse matrix factorization"*, in Proc. 6th SIAM Conf. Parallel Processing for Scientific Computing, SIAM, 1993, pp. 445–452.

[15] P. M. Campbell, K. D. Devine, J. E. Flaherty, L. G. Gervasio, and J. D. Teresco, *Dynamic octree load balancing using space-filling curves*, Tech. Report CS-03-01, Williams College Department of Computer Science, 2003.

[16] R. Chandra, R. Menon, L. Dagum, D. Koh, D. Maydan, and J. McDonald, *Parallel Programming in OpenMP*, Morgan Kaufmann, 2000.

[17] S. Chandra and M. Parashar, *ARMaDA: An adaptive application-sensitive partitioning framework for SAMR applications*, in Proc. 14th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002), Cambridge, 2002, ACTA Press, pp. 446–451.

[18] S. Chandra and M. Parashar, *Enabling scalable parallel implementations of structured adaptive mesh refinement applications*. Submitted for publication to the Journal of Supercomputing, 2004.

[19] S. Chandra, M. Parashar, and S. Hariri, *GridARM: An autonomic runtime management framework for SAMR applications in Grid environments*, in Proc. Autonomic Applications Workshop, 10th International Conference on High Performance Computing (HiPC 2003), Hyderabad, 2003, Elite Publishing, pp. 286–295.

[20] J. Chen and V. E. Taylor, *Mesh partitioning for distributed systems: Exploring optimal number of partitions with local and remote communication*, in Proc. 9th SIAM Conf. on Parallel Processing for Scientific Computation, 1999.

[21] T. M. Chilimbi, M. D. Hill, and J. R. Larus, *Cache-conscious structure layout*, in Proc. ACM SIGPLAN '99 Conf. on Prog. Lang. Design and Implementaion, May 1999.

[22] N. Chrisochoides, K. Barker, J. Dobbelaere, D. Nave, and K. Pingali, *Data movement and control substrate for parallel adaptive applications*, Concurrency Practice and Experience, (2002), pp. 77–101.

[23] N. Chrisochoides, K. Barker, D. Nave, and C. Hawblitzel, *Mobile Object Layer: A runtime substrate for parallel adaptive and irregular computations*, Advances in Engineering Software, 31 (2000), pp. 621–637.

[24] G. Cybenko, *Dynamic load balancing for distributed memory multiprocessors*, J. Parallel Distrib. Comput., 7 (1989), pp. 279–301.

[25] J. Dennis, *Data flow supercomputers*, IEEE Computer, 13 (1980), pp. 48–56.

[26] T. Desell, K. E. Maghraoui, and C. Varela, *Load balancing of autonomous actors over dynamic networks*, in Proc. Hawaii International Conference on System Sciences, 2004. HICSS-37 Software Technology Track, To appear.

[27] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, *Zoltan data management services for parallel dynamic applications*, Computing in Science and Engineering, 4 (2002), pp. 90–97.

[28] K. D. Devine, B. A. Hendrickson, E. Boman, M. St. John, and C. Vaughan, *Zoltan: A Dynamic Load Balancing Library for Parallel Applications; User's Guide*, Sandia National Laboratories, Albuquerque, NM, 1999. Tech. Report SAND99-1377. Open-source software distributed at `http://www.cs.sandia.gov/Zoltan`.

[29] J. Faik, L. G. Gervasio, J. E. Flaherty, J. Chang, J. D. Teresco, E. G. Boman, and K. D. Devine, *A model for resource-aware load balancing on heterogeneous clusters*, Tech. Report CS-04-03, Williams College Department of Computer Science, 2004. Presented at Cluster '04.

[30] J. Ferber and J. Briot, *Design of a concurrent language for distributed artificial intelligence*, in Proc. International Conference on Fifth Generation Computer Systems, vol. 2, Institute for New Generation Computer Technology, 1988, pp. 755–762.

[31] S. J. Fink, *Hierarchical Programming for Block–Structured Scientific Calculations*, PhD thesis, Department of Computer Science and Engineering, University of California, San Diego, 1998.

[32] J. E. Flaherty, R. M. Loy, M. S. Shephard, M. L. Simone, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz, *Distributed octree data structures and local refinement method for the parallel solution of three-dimensional conservation laws*, in Grid Generation and Adaptive Algorithms, M. Bern, J. Flaherty, and M. Luskin, eds., vol. 113 of The IMA Volumes in Mathematics and its Applications, Minneapolis, 1999, Institute for Mathematics and its Applications, Springer, pp. 113–134.

[33] J. E. Flaherty, R. M. Loy, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz, *Adaptive local refinement with octree load-balancing for the parallel solution of three-dimensional conservation laws*, J. Parallel Distrib. Comput., 47 (1997), pp. 139–152.

[34] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*, Morgan-Kaufman, 1999, ch. Computational Grids.

[35] M. Frigo and S. G. Johnson, *FFTW: An adaptive software architecture for the FFT*, in Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing, vol. 3, Seattle, WA, May 1998, pp. 1381–1384.

[36] K. S. Gatlin and L. Carter, *Architecture-cognizant divide and conquer algorithms*, in Proc. Supercomputing '99, November 1999.

[37] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI*, M. I. T. Press, 1994.

[38] S. Z. Guyer and C. Lin, *An annotation languge for optimizing software libraries*, ACM SIGPLAN Notices, 35 (2000), pp. 39–52.

[39] B. Hendrickson and R. Leland, *A multilevel algorithm for partitioning graphs*, in Proc. Supercomputing '95, 1995.

[40] Y. F. Hu and R. J. Blake, *An optimal dynamic load balancing algorithm*, Preprint DL-P-95-011, Daresbury Laboratory, Warrington, WA4 4AD, UK, 1995.

[41] C. Huang, O. Lawlor, and L. Kale, *Adaptive mpi*, in Proc. 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03), 2003.

[42] P. Husbands and J. C. Hoe, *MPI-StarT: Delivering network performance to numerical applications*, in Proc. of the IEEE/ACM Supercomputing Conference(SC98), Orlando, FL, Nov. 1998, p. 17.

[43] R. Jagannathan, *Coarse-grain dataflow programming of conventional parallel computers*, in Advanced Topics in Dataflow Computing and Multithreading, L. Bic, J.-L. Gaudiot, and G. Gao, eds., IEEE Computer Society Press, 1995, pp. 113–129.

[44] J.R. Gurd, et al., *The Manchester prototype dataflow computer*, Communications of the ACM, 28 (1985), pp. 34–52.

[45] L. V. Kalé, *The virtualization model of parallel programming : Runtime optimizations and the state of art*, in LACSI 2002, Albuquerque, October 2002.

[46] N. Karonis, B. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan, *Exploiting hierarchy in parallel computer networks to optimize collective operation performance*, in Proc. Fourteenth International Parallel and Distributed Processing Symposium (IPDPS '00), Cancun, 2000, pp. 377–384.

[47] N. T. Karonis, B. Toonen, and I. Foster, *MPICH-G2: A grid-enabled implementation of the Message Passing Interface*, J. Parallel Distrib. Comput., 63 (2003), pp. 551–563.

[48] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Scien. Comput., 20 (1999).

[49] ——, *Parallel multivelel k-way partitioning scheme for irregular graphs*, SIAM Review, 41 (1999), pp. 278–300.

[50] P. Kelly, O. Beckmann, A. Field, and S. Baden, *Themis: Component dependence metadata in adaptive parallel applications*, Parallel Processing Letters, 11 (2001), pp. 455–470.

[51] K. Kennedy, *Telescoping languages: A compiler strategy for implementation of high-level domain-specific programming systems*, in Proc. 14th International Parallel and Distributed Processing Symposium, May 2000, pp. 297–306.

[52] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. Bhoedjang, *MagPIe: MPI's collective communication operations for clustered wide area systems*, in Proc. of the 1999 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'99), Atlanta, GA, May 1999, pp. 131–140.

[53] W. Kim and G. Agha, *Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages*, in Proc. Supercomputing'95, 1995.

[54] E. Leiss and H. Reddy, *Distributed load balancing: design and performance analysis*, W. M. Kuck Research Computation Laboratory, 5 (1989), pp. 205–270.

[55] B. Lewis and D. J. Berg, *Multithreaded Programming with pthreads*, Sun Microsystems Press, 1997.

[56] X. Li and M. Parashar, *Dynamic load partitioning strategies for managing data of space and time heterogeneity in parallel SAMR applications*, in Proc. 9th International Euro-Par Conference (Euro-Par 2003), H. Kosch, L. Boszormenyi, and H. Hellwagner, eds., vol. 2790 of Lecture Notes in Computer Science, Klangenfurt, 2003, Springer-Verlag, pp. 181–188.

[57] P. Liniker, O. Beckmann, and P. Kelly, *Delayed evaluation self-optimising software components as a programming model*, in Proc. Euro-Par 2002, Parallel Processing, 8th International Euro-Par Conference, B. Monien and R. Feldmann, eds., vol. 2400 of Lecture Notes in Computer Science, Paderborn, August 2002, pp. 666–674.

[58] R. M. Loy, *AUTOPACK version 1.2*, Technical Memorandum ANL/MCS-TM-241, Mathemetics and Computer Science Division, Argonne National Laboratory, 2000.

[59] K. E. Maghraoui, J. E. Flaherty, B. K. Szymanski, J. D. Teresco, and C. Varela, *Adaptive computation over dynamic and heterogeneous networks*, in Proc. Fifth International Conference on Parallel Processing and Applied Mathematics (PPAM 2003), R. Wyrzykowski, J. Dongarra, M. Paprzycki, and J. Wasniewski, eds., vol. 3019 of Lecture Notes in Computer Science, Czestochowa, 2004, Springer Verlag, pp. 1083–1090.

[60] T. Minyard and Y. Kallinderis, *Parallel load balancing for dynamic execution environments*, Comput. Methods Appl. Mech. Engrg., 189 (2000), pp. 1295–1309.

[61] W. F. Mitchell, *Refinement tree based partitioning for adaptive grids*, in Proc. Seventh SIAM Conf. on Parallel Processing for Scientific Computing, SIAM, 1995, pp. 587–592.

[62] ——, *The design of a parallel adaptive multi-level code in Fortran 90*, in Proc. 2002 International Conference on Computational Science, 2002.

[63] M. Nibhanapudi and B. K. Szymanski, *High Performance Cluster Computing*, vol. I of Architectures and Systems, Prentice Hall, New York, 1999, ch. BSP-based Adaptive Parallel Processing, pp. 702–721.

[64] J. T. Oden, A. Patra, and Y. Feng, *Domain decomposition for adaptive* hp *finite element methods*, in Proc. Seventh Intl. Conf. Domain Decomposition Methods, State College, Pennsylvania, October 1993.

[65] M. Parashar and J. C. Browne, *Distributed dynamic data-structures for parallel adaptive mesh-refinement*, in Proc. IEEE International Conference for High Performance Computing, 1995, pp. 22–27.

[66] M. Parashar and J. C. Browne, *On partitioning dynamic adaptive grid hierarchies*, in Proc. 29th Annual Hawaii International Conference on System Sciences, vol. 1, Jan. 1996, pp. 604–613.

[67] A. Patra and J. T. Oden, *Problem decomposition for adaptive hp finite element methods*, Comp. Sys. Engng., 6 (1995), pp. 97–109.

[68] J. R. Pilkington and S. B. Baden, *Dynamic partitioning of non-uniform structured workloads with spacefilling curves*, IEEE Trans. on Parallel and Distributed Systems, 7 (1996), pp. 288–300.

[69] A. Pothen, H. Simon, and K.-P. Liou, *Partitioning sparse matrices with eigenvectors of graphs*, SIAM J. Mat. Anal. Appl., 11 (1990), pp. 430–452.

[70] D. Quinlan, B. Miller, B. Philip, and M. Schordan, *A C++ infrastructure for automatic introduction and translation of OpenMP directives*, in Proc. 16th International Parallel and Distributed Processing Symposium, April 2002, pp. 105–114.

[71] S. Ren, G. A. Agha, and M. Saito, *A modular approach for programming distributed real-time systems*, J. Parallel Distrib. Comput., 36 (1996), pp. 4–12.

[72] F. D. Sacerdoti, *A cache-friendly liquid load balancer*, master's thesis, University of California, San Diego, 2002.

[73] T. Saif and M. Parashar, *Understanding the behavior and performance of non-blocking communications in MPI*, in Proc. 9th International Euro-Par, Lecture Notes in Computer Science, Pisa, 2004, Springer-Verlag.

[74] H. D. Simon, *Partitioning of unstructured problems for parallel processing*, Comp. Sys. Engng., 2 (1991), pp. 135–148.

[75] S. Sinha and M. Parashar, *Adaptive system partitioning of AMR applications on heterogeneous clusters*, Cluster Computing, 5 (2002), pp. 343–352.

[76] A. Sohn and R. Biswas, *Communication studies of DMP and SMP machines*, Tech. Report NAS-97-004, NAS, 1997.

[77] J. Steensland, S. Chandra, and M. Parashar, *An application-centric characterization of domain-based SFC partitioners for parallel SAMR*, IEEE Trans. Parallel and Distrib. Syst., 13 (2002), pp. 1275–1289.

[78] B. Szymanski, C. Varela, J. Cummings, and J. Napolitano, *Dynamically reconfigurable scientific computing on large-scale heterogeneous grids*, in Proc. Fifth International Conference on Parallel Processing and Applied Mathematics (PPAM'2003), no. 3019 in LNCS, Czestochowa, Poland, September 2003.

[79] V. Taylor, X. Wu, J. Geisler, and R. Stevens, *Using kernel couplings to predict parallel application performance*, in Proc. 11th IEEE International Symposium on High-Performance Distributed Computing (HPDC 2002), Edinburgh, 2002.

[80] V. Taylor, X. Wu, X. Li, J. Geisler, Z. Lan, M. Hereld, I. R. Judson, and R. Stevens, *Prophesy: Automating the modeling process*, in Third Annual International Workshop on Active Middleware Services, San Francisco, 2001.

[81] V. E. Taylor and B. Nour-Omid, *A study of the factorization fill-in for a parallel implementation of the finite element method*, Int. J. Numer. Meth. Engng., 37 (1994), pp. 3809–3823.

[82] J. D. Teresco, M. W. Beall, J. E. Flaherty, and M. S. Shephard, *A hierarchical partition model for adaptive finite element computation*, Comput. Methods Appl. Mech. Engrg., 184 (2000), pp. 269–285.

[83] J. D. Teresco, K. D. Devine, and J. E. Flaherty, *Partitioning and dynamic load balancing for the numerical solution of partial differential equations*, Tech. Report CS-04-11, Williams College Department of Computer Science, 2005. Chapter submitted to *Numerical Solution of Partial Differential*

Equations on Parallel Computers, Are Magnus Bruaset, Petter Bjørstad, Aslak Tveito, editors.

[84] J. D. Teresco, J. Faik, and J. E. Flaherty, *Hierarchical partitioning and dynamic load balancing for scientific computation*, Tech. Report CS-04-04, Williams College Department of Computer Science, 2004. Submitted to Proc. PARA '04.

[85] J. D. Teresco and L. P. Ungar, *A comparison of Zoltan dynamic load balancers for adaptive computation*, Tech. Report CS-03-02, Williams College Department of Computer Science, 2003. Presented at COMPLAS '03.

[86] C. Tomlinson, P. Cannata, G. Meredith, and D. Woelk, *The extensible services switch in Carnot*, IEEE Parallel and Distributed Technology, 1 (1993), pp. 16–20.

[87] S. Vajracharya, S. Karmesin, P. Beckman, J. Crotinger, A. Malony, S. Shende, R. Oldehoeft, and S. Smith, *SMARTS: Exploiting temporal locality and parallelism through vertical execution*, in International Conference on Supercomputing, 1999.

[88] C. Varela, *Worldwide Computing with Universal Actors: Linguistic Abstractions for Naming, Migration, and Coordination*, PhD thesis, University of Illinois at Urbana-Champaign, 2001.

[89] C. Varela and G. Agha, *Programming dynamically reconfigurable systems with SALSA*, in Proc. OOPSLA 2001, Tampa Bay, 2001, ACM.

[90] C. Walshaw and M. Cross, *Parallel Optimisation Algorithms for Multilevel Mesh Partitioning*, Parallel Comput., 26 (2000), pp. 1635–1660.

[91] ——, *Multilevel Mesh Partitioning for Heterogeneous Communication Networks*, Future Generation Comput. Syst., 17 (2001), pp. 601–623. (originally published as Univ. Greenwich Tech. Rep. 00/IM/57).

[92] M. S. Warren and J. K. Salmon, *A parallel hashed oct-tree n-body algorithm*, in Proc. Supercomputing '93, IEEE Computer Society, 1993, pp. 12–21.

[93] R. C. Whaley and J. J. Dongarra, *Automatically tuned linear algebra software*, in Conf. High Performance Networking and Computing, Proc. 1998 ACM/IEEE conference on Supercomputing, Orlando, Florida, November 1998, pp. 1–27.

[94] R. Wolski, N. T. Spring, and J. Hayes, *The Network Weather Service: A distributed resource performance forecasting service for metacomputing*, Future Generation Comput. Syst., 15 (1999), pp. 757–768.

[95] X. Wu, V. Taylor, J. Geisler, Z. Lan, R. Stevens, M. Hereld, and I. Judson, *Design and development of Prophesy Performance Database for distributed scientific applications*, in Proc. 10th SIAM Conference on Parallel Processing, 2001.

[96] X. Wu, V. Taylor, and R. Stevens, *Design and implementation of Prophesy Performance Database for distributed scientific applications*, in Proc. 13th IASTED Parallel and Distributed Computing and Systems Conference (PDCS2001), 2001.

[97] Y. Zhang, S. Chandra, S. Hariri, and M. Parashar, *Autonomic proactive runtime partitioning strategies for SAMR applications*, in Proc. NSF Next Generation Systems Program Workshop, IEEE/ACM 18th International Parallel and Distributed Processing Symposium (CDROM), Santa Fe, 2004, IEEE Computer Society Press.