PAPER *Special Section/Issue on Hardware/Software Support for High Performance Scientific and Engineering Computing*

A Super-Programming Technique for Large Sparse Matrix Multiplication on PC Clusters*

ziavras@njit.edu Dejiang JIN and Sotirios G.ZIAVRAS†

Summary

The multiplication of large spare matrices is a basic operation in many scientific and engineering applications. There exist some high-performance library routines for this operation. They are often optimized based on the target architecture. For a parallel environment, it is essential to partition the entire operation into well balanced tasks and assign them to individual processing elements. Most of the existing techniques partition the given matrices based on some kind of workload estimation. For irregular sparse matrices on PC clusters, however, the workloads may not be well estimated in advance. Any approach other than run-time dynamic partitioning may degrade performance. In this paper, we apply our super-programming approach [24] to parallel large matrix multiplication on PC clusters. In our approach, tasks are partitioned into super-instructions that are dynamically assigned to member computer nodes. Thus, the load balancing logic is separated from the computing logic; the former is taken over by the runtime environment. Our super-programming approach facilitates ease of program development and targets high efficiency in dynamic load balancing. Workloads can be balanced effectively and the optimization overhead is small. The results prove the viability of our approach.

Key words:

PC cluster, matrix multiplication, load balancing, programming model, performance evaluation.

1. Introduction

Matrix multiplication (MM) is an important linear algebra operation. A number of scientific and engineering applications include this operation as a building block. Due to its fundamental importance, much effort has been devoted to studying and implementing MM [1-8]. MM has been included in several libraries [1], [6], [9]-[12],[14]. Many MM algorithms have been developed for parallel systems [2], [3], [5], [13], [15], [16]. In many applications, the matrices are sparse. Although all sparse matrices can be treated as dense matrices, the sparsity of matrices can be exploited to gain higher performance. For parallel MM, the entire task should be decomposed. This introduces various overheads. The most important are the communication and load balancing overheads. Partitioning the matrices into sub-matrix blocks and decomposing the MM operation are often technology dependent [4],[13]. Applying special implementations for sparse sub-matrix blocks may improve performance since the workload of sub-matrix operations may vary. The information about the workload of a task for sparse matrix operation may not be available at compile time or even at the time of initiating subroutines. It may be available only after these routines have been executed. Since this increases the complexity of load balancing, it is often ignored.

Most parallel algorithms are optimized based on the characteristics of the targeting platform [2], [3], [5], [15], [16]. The PC cluster computing platform has recently emerged as a viable alternative for high-performance, low-cost computing. Generally, the PCs in a cluster have a lot of resources that can be used simultaneously. They have relatively weak communication capabilities. They lack high performance implementation support for data communications compared to supercomputers. They only support some communication channels implemented by software that capitalizes on Ethernet connections.

MM operations for sparse matrices are embedded in many host programs. Optimal matrix partitioning has to be performed at run time rather than at compile time even if the MM algorithm adopts a static strategy. Parameters must be optimized when a subroutine is launched. This introduces an additional *management overhead*. In heterogeneous environments, finding the optimal partitioning is an NP-complete problem [13]. How we can minimize the management overhead still needs to be studied to find a high performance solution for MM on PC clusters. This becomes a more difficult problem for sparse matrices.

To address this problem, in this paper we use our recently introduced *super-programming model* (SPM) [24]. Using SPM, we analyze the performance of super-programs for MM and discuss the effect of various scheduling strategies. The next section presents our SPM model. Section 3 introduces our SPM-based approach for MM that minimizes the management overhead for better performance. Sections 4 and 5 present simulation and experimental results, respectively. Sections 6 and 7 analyze the results and compare SPM with other programming models, respectively.

Manuscript received Oct 2003.

† The authors are with the ECE Dept., New Jersey Institute of Technology, Newark, NJ 07102

* This work was supported in part by the U.S. Department of Energy under grant ER63384.

2. Super-Programming Model (SPM)

In SMP, the parallel system is modeled as a single virtual machine (VM) with a single super-processor that includes multiple instruction execution units (IEUs) (member computers). For each application domain, a set of coarse data types and a set of basic abstract operations on the "build-in" data, which are called super-data-blocks (SDBs) and super-instructions (SIs), respectively, are defined to customize the VM for the application. Tasks with arbitrary workload representing logic functions are modeled as super-functions (SFs), which are coded with these SIs. Application programs are modeled as super-programs (SPs) that are implemented with SFs.

To effectively support application portability among different computing platforms and to effectively balance the workload in parallel systems, an effective instruction set architecture (ISA) should be developed for operations in each application domain under SPM. Application programs should maximize the utilization of such instructions in the coding process. Then, as long as an efficient implementation exists for each SI on given computing platforms, code portability is guaranteed and good load balancing becomes more feasible by focusing on scheduling at this coarser SI level.

In SPM, SIs are the key elements. They are high-level abstract operations to solve application problems. They have some features similar to the instructions of microprocessors. 1) SIs are atomic. Each SI can only be assigned to and be executed on an IEU. 2) SIs can run independently of each other as long as there are no dependencies among them. Interdependencies are handled only at the beginning and the end of their execution. 3) The workload of SIs has a quite accurate upper bound rather than being arbitrary. It can be executed within a known maximum execution time on a given type of computer. At the execution level of IEUs (i.e. PC nodes), SIs can be implemented with any mature ordinary programming technique. In a heterogeneous PC cluster, for an SI there may exist multiple implementations corresponding to different computers. At run time, SIs are dynamically assigned to IEUs. The workload predictability makes the system allocate the task efficiently. When the degree of parallelism in SFs, measured in number of SIs, is much larger than the number of nodes in the cluster, IEUs have little chance of being idle.

In SPM, SFs play a critical role in parallelizing SPs. As mentioned earlier, due to communication overheads, it is difficult for PC clusters to exploit low-level parallelism. Thus, to successfully achieve parallel computation, it is critical to sufficiently exploit high-level parallelism. The VM provides the mechanism to execute multiple SIs simultaneously. In the general case, SFs will issue a number of SIs to complete a large logic function. Many SIs may be executed in parallel. This provides the high-level parallelism to utilize multiple IEUs. When an SP runs on a PC cluster, SFs feed the SIs to the system. However, the dispatch unit of SIs should check for data (operand) dependencies so that data are consistent. Data dependencies limit parallelism. By co-operating with the SI dispatch unit, through their scheduling policies, SFs try to maximize the parallelism among dispatched SIs, thus efficiently exploiting the parallel capabilities of the system. Extending the functional unit to handle multiple SIs makes the high-level parallelism not only to be determined by the algorithm but also by the SI designer. Also, scheduling policies for SFs will affect multiple facets of parallel execution. Increasing the degree of high-level parallelism makes easier the task of balancing the workload. SFs make the parallelization transparent to the application developers.

3. Our SPM Approach for MM

3.1 Basic Data Blocks

In this paper, we define a set of build-in data types for MM on PC clusters. It includes only one abstract data type called *sub-matrix block*. The sub-matrix block is an n x m matrix, where both m and n are not larger than a predefined parameter k. The sub-matrix block not only includes the original values of the contained elements, but also some features, such as the sparsity of the block, in the form of metadata. Large matrices can be partitioned into sub-matrix blocks. The sparsity and type of sub-matrix blocks may vary in sparse matrices.

3.2 Super-Instruction (SI) Set for MM

An effective ISA should be developed for each application domain. The contained SIs should correspond to frequently used operations for the corresponding application domain.. To support MM, here we define the SI **Multiply(**A, B, C**)**, where A, B and C are sub-matrix blocks. The functionality of this SI is C = C+ A*B. It is obvious that the task completed by the SI is isolated. Once all operands are available, the SI can be executed at any node without interrupts. Because the workload is determined by the size of the sub-matrix blocks, and the latter have limited data size, there is a pre-known upper bound on its execution time. It can be easily reduced by reducing the designed parameter k. Other application domains may require many more SIs.

SIs include all required references to operands. These references include the data ID and meta-data of the SDBs

that store sub-matrix blocks. Thus, the assigned nodes always know the type of the sub-matrix block and its sparsity.

3.3 A Super-Function (SF) for Matrix Multiplication

We focus on the SF that implements arbitrarily large MM C= A*B, where A, B and C are N x N square matrices. For the sake of simplicity, we focus on square matrices and sub-matrix blocks. Let $X=\{X_{i,j}\}$, for X = A, B or C. The sub-block elements $X_{i,j}$ of these matrices are square and have the same size. Initially, $C_{i,j} = \{0\}$ for any i and j. The SF invokes the multiply SI N^3 times. Each SI instance $I_{i,j,k}$ is for updating $C_{i,j} = A_{i,k} * B_{k,j} + C_{i,j}$, for all of i, j, k =0, 1, … , N-1. If i? i' or j? j', then $I_{i,j,k}$ and $I_{i',j',k'}$ do not have any data dependence and can be executed in parallel. If only k? k', then $I_{i,j,k}$ and $I_{i,j,k'}$ will update the same block. Since we lack a simple addition SI, they have to be executed in sequence.
In SPM, a high-level global logic space is created for each SP. IEUs can query SDBs based on their IDs to find out if they reside in external storage or have been cached by other IEUs. The operands (SDBs) of an SI can be found by the runtime environment based on their IDs before the body of the SI is executed. The overhead of loading a data block may be significant. However, well established cache techniques may help us reduce this penalty. In our study, all scheduling policies assign SIs that update the same block of the result to the same IEU. This increases substantially the cache hit ratio. Further, 2 of the 3 operands of SIs are read only. Caching them in multiple nodes does not involve any coherence penalty. Additionally, we apply multithreading so that if there is a cache miss for a thread implementing an SI, the IEU switches execution to another SI with operands stored locally, if possible.
Since SIs are implemented in isolation, no communication exists between threads implementing different SIs. SIs can only share operands, that is sub-matrix blocks. These operands are loaded from files or remote nodes. In the latter case, our environment that was built in Java uses RMI (Remote Method Invocation) to load data. RMI provides a simple and direct model for distributed computation with Java objects.

3.4 Scheduling Policies

Under the SPM model, the chosen scheduling policies for SFs affect performance. For this reason, we describe a few scheduling policies. We compare their performance below. In these policies, all SIs that update the same sub-matrix block will be assigned to a same computer. Once a block is assigned to it, an IEU executes all SIs that update this block exclusively. Except for the synchronous policy, all other policies are asynchronous. Once an IEU is free to accept an SI, a qualified SI is assigned to it immediately. Thus, IEUs are not necessarily assigned tasks simultaneously.

1) **Synchronous Policy (syn)**
This policy makes the SF of MM imitate an implementation for distributed-memory parallel computers. Under this policy, SIs are issued in synchronous steps. In each step, each IEU is assigned an SI. After it finishes its work, an IEU stays idle until all IEUs are done. Then, the next step begins. In this scheme, sub-matrix blocks of the product matrix C are statically assigned to IEUs evenly. In every N steps, an IEU calculates a sub-matrix block.

2) **Static Scheduling Policy (S)**
In this scheme, sub-matrix blocks of the product C are statically assigned to IEUs. Once an IEU is available to execute another SI, the scheduler first assigns an SI that updates the same block with the previous SI. If the block is done or there is no previous SI, any SI that updates an uncalculated sub-matrix block is assigned to the IEU.

3) **Random (Dynamic) Scheduling Policy (R)**
Under this policy, if an IEU is available to execute another SI, the scheduler first assigns an SI that updates the same block with the previous SI assigned to this IEU. If the block is done or there is no previous SI, the scheduler will find an uncalculated sub-matrix block randomly and assign the first SI that updates the block to the IEU.

4) **Smart Scheduling Policy with Random Seed (SR)**
This scheduling policy is basically similar to the Random Scheduling Policy. The only difference is the way to assign a sub-matrix block to an IEU. If it is the first block of the IEU, the scheduler still picks up one randomly. If it is not the first one, based on the assignment history the scheduler will choose an uncalculated block in this order: a) Block $C_{i,j}$ has high priority if there exist sub-matrix blocks $C_{i',j}$ and $C_{i,j'}$ that were assigned earlier to the IEU. b) Block $C_{i,j}$ has medium priority if there exists either a sub-matrix block $C_{i',j}$ or $C_{i,j'}$ in the blocks that were assigned to the IEU. c) Other blocks have low priority. If only low priority blocks exist, a random choice is made for assignment.

5) **Smart Scheduling Policy with Static Seed (SS)**
This policy is very similar to the above SR policy except that the first blocks assigned to IEUs are statically selected with a 2D grid arrangement, rather than randomly. The scheme is similar to the S policy.

3.5 Exploiting the Sparsity in Matrices

Matrix sparsity can be exploited to improve the overall performance in MM. In our approach, the sparsity of matrices can be exploited at the SI or SF level. In the former

case, when executing an SI the IEU can select the implementation procedure that not only matches the computer platform but also is the best to exploit the specific sparsity in the operands. Any sequential algorithm for sparse MM can be utilized here.

At the SF level, an SF can easily skip SIs with zero operands. The execution of SIs no longer has to take fixed time. An SI with sparser operands takes less time. Further, since we approximate the SI execution time by checking the meta-data of sub-matrix blocks, it is not so important to select the "correct" algorithm based on the distribution of non-zero sub-block elements.

4. Simulation

Our simulation programs run on a PC and fully implement these scheduling policies for MM. In these programs, an SI is represented as an abstract task with a statically determined workload. A member computer is represented as an abstract "IEU" object with a peak performance parameter. IEU objects keep track of all SIs that have been assigned to them and simulate the execution of SIs by recording the time of loading operands and "executing" these SIs. They also record their accumulated idle time when they are free but cannot get an SI to execute.

In our simulations, the number of PCs in the cluster is chosen as 1, 2, 4, 8, 16, 32 and 64. The performance of computer nodes is normalized here. In a homogeneous environment, the (relative) peak performance of any node is 1. In a heterogeneous environment, the relative peak performance of any node with an odd index is 3 and the relative peak performance of any node with an even index is 1. The input/output matrices contain 32x32 blocks, each of size 256x256.. All elements in them are sparse square matrices having the same dimensions. Based on a pilot experiment, we assume that the workload of SIs to multiply a pair of blocks and accumulate the results is distributed uniformly between 0 and 20000 units (1000 units represent 1ms of CPU time on the reference node). To load a sub-matrix block remotely, the sending node requires 250 units and the receiver takes 250 units of time so that the total cost to load the two operands of an SI remotely is 10% of its execution time. All simulations repeat 16 times with different data sets. The results presented here are the average values of multiple simulation runs.

The static schedule and the workload of SIs were created in advance. For the S policy, matrices are partitioned into contiguous sub-blocks. Each node is assigned one sub-block. In the homogeneous case, the sub-blocks are simply square. In the heterogeneous case, they are rectangular. Their size for odd-numbered nodes is three times larger than for even-numbered nodes. The SS policy assigns the upper-left SDB initially. Both unicast and multicast communications are simulated. For multicasts, the membership of nodes in the multicast group is determined statically based on the mapping table of the static schedule. All nodes located on the same row form a group for multicasting the sub-matrix blocks of matrix A; all nodes located on a column form a group for multicasting the sub-matrix blocks of matrix B.

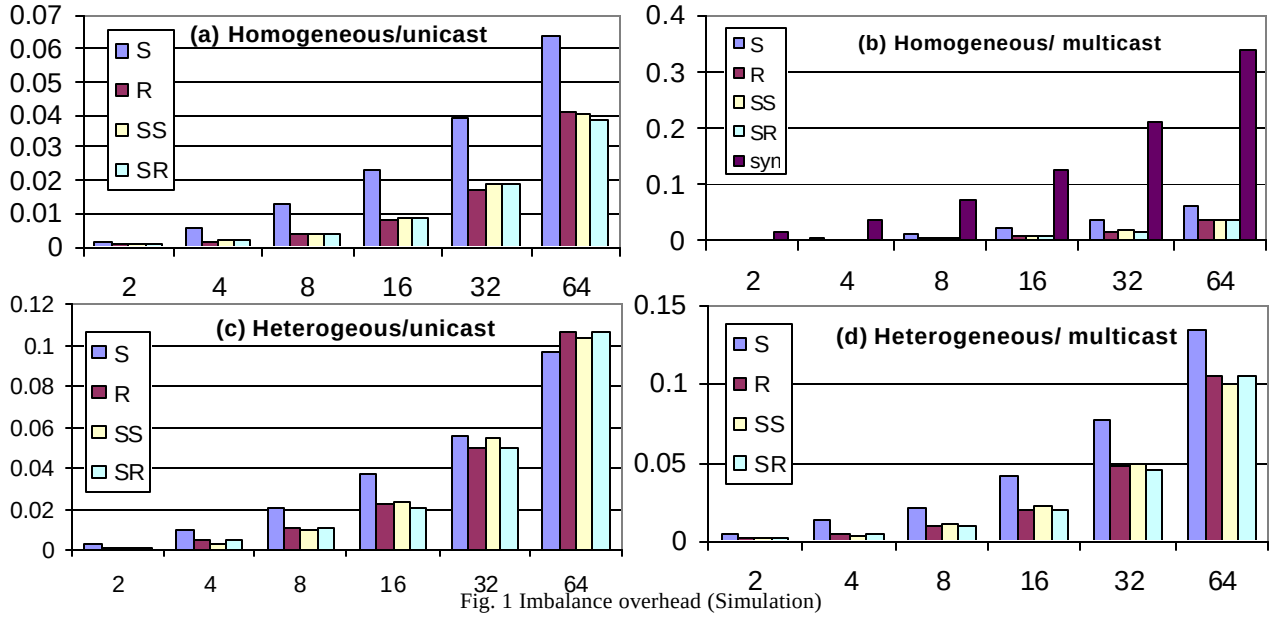


Fig. 1 Imbalance overhead (Simulation)

All combinations of scheduling policies and environments were used. But the syn policy was used only with multicast operations in a homogeneous environment. Simulation

results of the imbalance and communication overheads are shown in Fig. 1 and Fig. 2, respectively.

The syn policy has very high imbalance overhead (thus, it is only used in a homogeneous environment with multicasts). Except this, the S policy has higher imbalance overhead than the others. The highest communication overhead corresponds to the R policy independently of the environment's homogeneity because of the underlying randomness in scheduling. The next section presents actually results on a PC cluster.

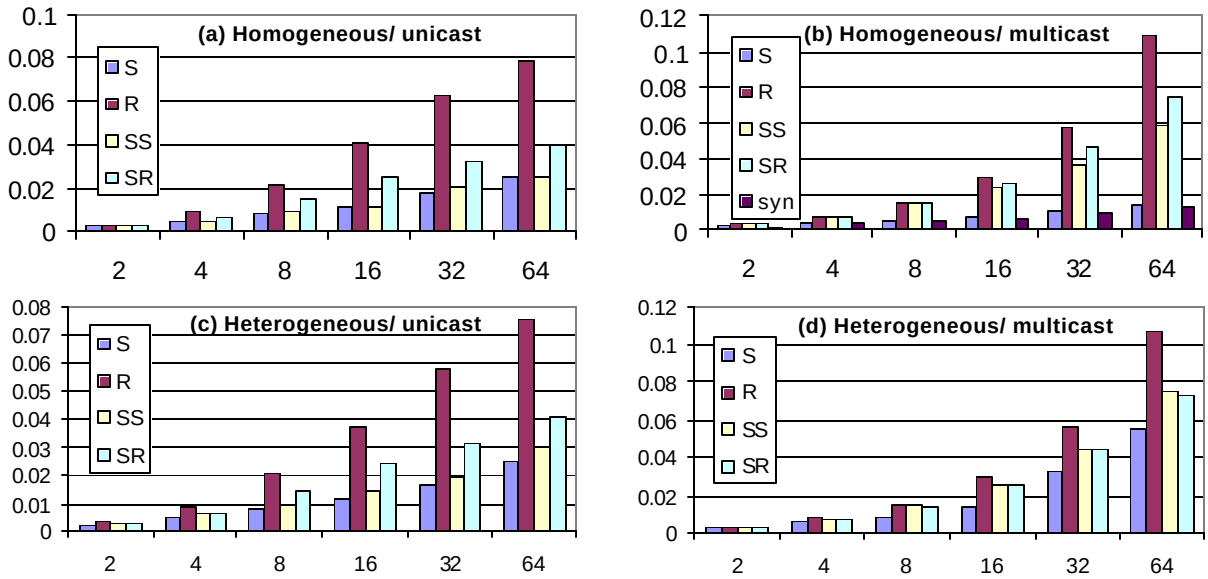


Fig 2. Communication overhead (Simulation)

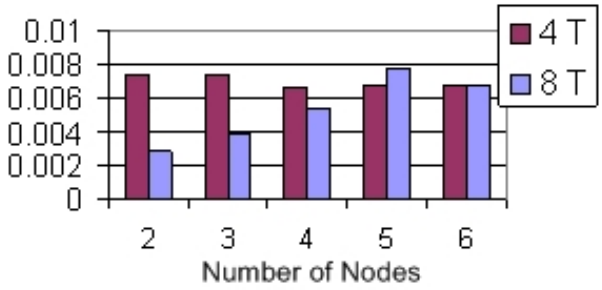


Fig 3. IEU relative idle time (experiment)

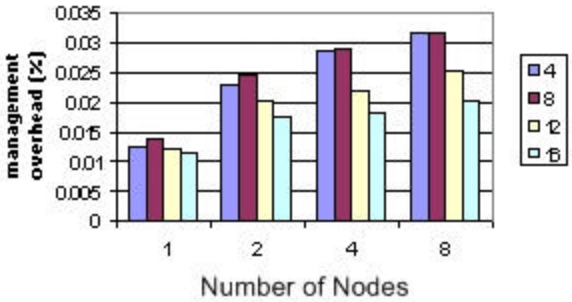


Fig 4. Relative management overhead

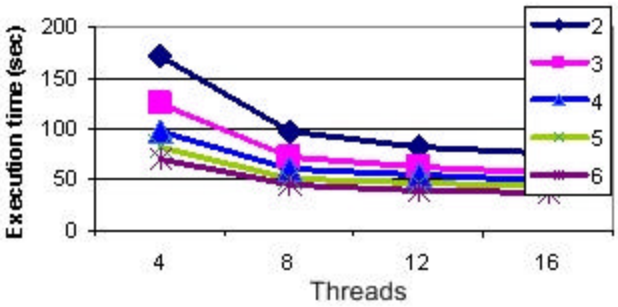


Fig 5. Effect of multithreading

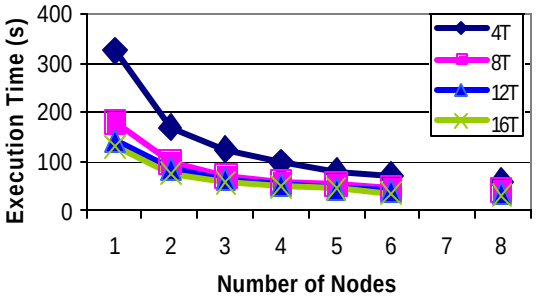


Fig 6. Overall performance

5. Experimental Results on a PC Cluster

All the experiments were performed on a PC cluster with eight dual-processor nodes. The random/dynamic scheduling approach was followed. Each node is equipped with two Athlon processors running at 1.2 GHz, 1GB of main memory, a 64K Level-1 cache and a 256K Level-2 cache. All the nodes are connected through a switch that forms an Ethernet LAN. Each link has 100Mbps bandwidth. All the PCs run Red Hat 9.0 and share the same view of the file system for files via an NFS file server.

We used a set of synthetic sparse matrices of size 8192x8192; they are completely irregular with 5% non-zero elements. These matrices were partitioned into 32x32 sub-matrices of size 256x256. The super-program was developed manually using the SF described in Section 3. We implemented a runtime environment to support our SPM. In our runtime environment, there is a virtual IEU on each PC node. Recently used SDBs are cached in nodes. The runtime environment and the SIs are implemented in the Java language. To hide the long latency of loading data remotely, an IEU receives multiple SIs. They are executed in separate threads and are scheduled by the local operating system when their operands are locally available.

Experimental results of the imbalance overhead are shown in Fig. 3. Since the simulation programs were implemented with dynamic management functions, the management overhead should not be heavier than the simulation time. Fig. 4 shows the results of management overhead (upper bound). The number in the legend in Fig. 3 and Fig. 4 represents the number of threads. The effect of multithreading is shown in Fig. 5. The number in the legend is the number of nodes used in the experiment.

Fig. 6 shows that the execution time of the experimental program decreases when the number of nodes increases. The SPM program preserves the intrinsic parallelism of MM, so when the parallelism of the hardware is increased, the SPM can efficiently exploit the hardware. This property implicitly indicates that SPM supports scalability.

6. Discussion of Results

From Fig. 1(b), we can see that the synchronous policy introduces significantly higher imbalance overhead. This is because SIs, unlike instructions in microprocessors, have very irregular execution times. Thus, for high performance it is more favorable for PC clusters to issue SIs asynchronously. Also, dynamic scheduling algorithms with multicasting have less imbalance overhead than static scheduling algorithms independent of unicasting or multicasting for data communications in homogeneous [(a) and (b)] or heterogeneous environments [(c) and (d)]. The difference in the imbalance overhead among different (dynamic) scheduling strategies, however, is very small. The difference in the imbalance overhead between the static and dynamic schedules depends on the average number of tasks executed on a node. The more tasks a node executes (i.e. fewer nodes in the cluster), the larger the difference. The experiments give similar results (Fig. 3). Thus, for sparse MM, in the case where there exists diverse workload between updating different sub-matrix blocks, especially for a large sparse MM, the asynchronous programming models are more appropriate than synchronous programming models. For large problems, dynamic strategies have an advantage in reducing the imbalance overhead.

Comparing the results [(a) and (b)] in a homogeneous environment with their counterparts [(c) and (d)] in a heterogeneous environment in Fig 1, we can see that the imbalance overhead increases for the latter. This is because in a heterogeneous environment the nodes have different peak performance. The slower nodes have better chance to complete earlier their last SI. Other nodes with higher peak performance may be idle at the same time, thus wasting more computing capability. The increase in dynamic scheduling policies is larger. This makes the advantage of dynamic scheduling policies to diminish.

Most simulation programs run for about 60-200ms. Since the scheduling algorithms were implemented rather than simulated, the management overhead for both static and dynamic scheduling is less than 200 ms. Considering that there are 32K SIs to be executed for the multiplication of two 32x32 matrices of sub-blocks, each SI may take a few ms. The relative management overhead is less than 1% of the total execution time. Fig. 4 shows the upper bound on the relative management overhead which is expressed as the ratio of the execution times between simulations and experiments. Compared to the communication and imbalance overheads, this overhead can be ignored.

It is very important that the management overhead be kept low even for dynamic scheduling in heterogeneous environments. As mentioned above, in our simulations static scheduling is based on offline (optimization) arrangements. When embedding the MM function in other application programs, the optimization of the execution plan itself will be an execution overhead. In heterogeneous PC cluster environments, resolving the NP-complete optimization problem may introduce significant high execution overhead. Adopting a dynamic scheduling policy for SFs, SPM transfers this kind of overhead into the

Manuscript received Oct 2003.

† The authors are with the ECE Dept., New Jersey Institute of Technology, Newark, NJ 07102

* This work was supported in part by the U.S. Department of Energy under grant ER63384.

management overhead of the VM for SI scheduling, issue and commitment.

From Fig. 2, we can find out that in most cases the overhead of SS is almost as low as the overhead of S, although the communication overhead of S is the lowest. Combining all these overheads, we may think that SS is a better alternative to S.

Fig. 5 shows that when up to 8 threads are allowed per node, the overall execution time of the program is reduced considerably compared to 4 threads per node. But the decrease is insignificant with further increases in the number of allowed threads. Considering the fact that the average execution time of an SI is about 4ms and the average latency of loading a block is 20ms, we can understand that the multi-threading scheme is helpful to hide the latency of remotely loading data. But once the latency is hidden, further increasing the number of threads cannot improve performance. It only makes more difficult the task of predicting the finish time of an SI if multiple SIs are scheduled by the local operating system.

7. Comparison with Other Parallel Programming Models

There exist two distinct popular programming models for parallel programming. They are the Message-Passing and Shared-Memory models [18], [22]. Generally, both programming models are independent of the specifics of the underlying architecture. For example, ZPL is an array programming language that follows the shared-memory model [23]. Nevertheless, for distributed memory implementations it employs either MPI or PVM for communications. The optimization of parallel application programs, however, is done for specific architectures. For this reason, the Distributed-Shared Memory model has been proposed. UPC does follow this model [17]. It gives up the high-level abstraction and exposes the lower-level system architecture to programmers. It provides programmers the capability to exploit the distributed features of the architecture, thus increasing their responsibility in better program implementations.

SPM targets primarily PC clusters. It employs high-level abstractions with a VM. Programmers can operate with an SI on an SDB by considering it an atom. They can never access the SDB's internal data structure. Thus, application programmers can only access these objects rather than lower-level block structures. This gives programmers a coarse-grain data space. Optimization is achieved by customizing the VM. Programmers can exploit the system with many mature techniques developed for sequential computers.

An array programming language, like ZPL, exploits only low-level data parallelism. The higher-level structure in programs is executed sequentially. POOMA is similar but follows an object-oriented style [25]. A library provides a few constructs to build logic objects and access them in a global-shared style using data-parallel computing. However, high-level parallelization is more appropriate for PC clusters. To exploit higher-level parallelism, task-oriented approaches are used where extracting the parallelism between program structures is a major task for programmers. The runtime environment exploits this parallelism through task scheduling. A few multithreaded approaches have also been developed for task mapping to threads. Cilk is such an approach that employs a C-based multithreaded language [20]. The programmer uses directives to declare the structure of the task tree. All the sibling tasks are executed in parallel within multi-threads that share a common context in the shared memory. EARTH-C also is such an approach [19], [21].

Approaches that exploit data parallelism, like ZPL and POOMA, usually do not mention load balancing explicitly. The load is balanced implicitly by the programmers through data partitioning. For task-oriented approaches, load balancing is normally performed at run time with two techniques. One preempts and/or migrates threads at run time. Cilk uses thread migration to balance the workload. Charm++ behaves similarly; it encapsulates working threads and relevant data into objects. When a function is invoked, the runtime environment of Cilk creates a new thread and adds the load to a processor no matter how busy the system is. The system balances the workload by migrating these threads at run time. On a shared-memory system, thread migration with Cilk has very low overhead [20]. However, for PC clusters with a shared-nothing system, the cost of thread migration is much higher than that on a shared-memory system. In this case, a performance model that considers thread migration but ignores its cost cannot be expected to provide a very accurate prediction.

The second technique is load control through task scheduling. In the approach that uses a pool of tasks, tasks are put into a queue, and the runtime environment assigns tasks to nodes when they become free. When assigned, the task keeps running on the node without migration. The system balances the workload by flexibly assigning new tasks to nodes that can quickly finish their work. In existing programming models, tasks are function-oriented and delimited by programmers [26]. The workload of a task may be arbitrarily large and also irregular. If one task is significantly heavy, the system may still be unbalanced.

In contrast, SPM partitions tasks into basic units (SIs) which have limited workload and do not involve data communication during their implementation. The execution

of SIs becomes predictable (an upper bound is known) though it may be irregular. Only a limited number of SIs are assigned to processors, when computing resources are available. This mechanism guarantees that no processor is either overloaded or underloaded. Thus, the workloads are balanced automatically among the processors. Generally, SPM is a non-preemptive, multi-threaded execution model. It was first tested with a data-mining problem [24].

8. Conclusions

(a) In SPM, the parallelization of programs is separated from workload balancing. The predictable characteristics of SIs make load balancing a feasible feature. By asynchronously and dynamically issuing SIs to IEUs, VM can achieve load balancing efficiently. This removes the demand of load balancing from parallel programs, thus making the development of parallel programs much easier, especially in heterogeneous PC cluster environments. (b) Further, SPM also reduces the execution overhead. By adopting dynamic scheduling policies for SFs, SPM transfers the overhead of optimizing the execution into a management overhead of VM for SI scheduling, issue and commitment. The latter can be kept low. In our experiments with sparse MM the latter only introduces insignificant overhead. (c) All achievements in the sequential implementation of sparse MM can be directly utilized to exploit the sparsity of sub-matrix blocks for basic tasks (SIs) and also bind their implementations dynamically. At the sub-blocks level, exploiting any sparsity is achieved by not issuing some SIs; thus, a standard implementation should be sufficient for all kinds of matrices. (d) SPM reduces the communication overhead by exploiting the locality of operands for SIs. This is affected by the scheduling policies of SFs. In our sparse MM SF, the SS policy exploits the local cache efficiently to reduce the communication overhead significantly.

References

[1] I. S. Duff , M. A. Heroux and R. Pozo "An Overview of the Sparse Basic Linear Algebra Subprograms: The New Standard from the BLAS Technical Forum", ACM Trans. on Math. Software, Vol28(2), p239-267, Jun. 2002

[2] K. Li, "Scalable Parallel Matrix Multiplication on Distributed Memory Parallel Computers", Proc. IPDPS, p307 –314, May 2000.

[3] T.I. Tokic, E.I. Milovanovic, et al, "Matrix Multiplication on Non-planar Systolic Arrays", Proc. Telecommunications in Modern Satellite, Cable and Broadcasting Services, p514 -517, Oct. 1999

[4] K. Dackland and Bo Kågström, "Blocked Algorithms and Software for Reduction of a Regular Matrix Pair to Generalized Schur Form", ACM Trans. Math. Software Vol25 (4), p425-454 Dec. 1999

[5] J. Choi, "A New Parallel Matrix Multiplication Algorithm on Distributed-Memory Concurrent Computers", HPC Asia '97, p224 –229, May 1997

[6] I. S. Duff, et al, "Level 3 Basic Linear Algebra Subprograms for Sparse Matrices: a User-Level Interface", ACM Trans. Math. Software Vol23 (3), p379-40, Sept. 1997

[7] A.E. Yagle, "Fast Algorithms for Matrix Multiplication Using Pseudo-Number-Theoretic Transforms", IEEE Trans. Signal Processing, Vol. 43(1), p71 -76, Jan 1995

[8] Hyuk-Jae Lee, J.A.B. Fortes, "Toward Data Distribution Independent Parallel Matrix Multiplication", Proc. Int. Parallel Processing Sym., p436 –440, Apr 1995

[9] T. Hopkins, "Renovating the Collected Algorithms from ACM", ACM Trans. Math. Software Vol28 (1) p59-74, Mar. 2002

[10] B. Kågström , P. Ling and C. van Loan, "GEMM-based Level 3 BLAS: High-performance Model Implementations and Performance Evaluation Benchmark", ACM Trans. Math. Software, Vol24 (3), p268-302, Sept. 1998

[11] J. J. Dongarra , L. S. Blackford , et al, ScaLAPACK User's Guide, Soc. for Ind. and Appl. Math., Philadelphia, PA, 1997)

[12] N. J. Higham, "Exploiting Fast Matrix Multiplication within the Level 3 BLAS", ACM Trans. Math. Software, Vol16 (4), p352-368, Dec. 1990

[13] O. Beaumont, et al, "Matrix Multiplication on Heterogeneous Platforms", IEEE Trans. Parallel Distrib. Sys., Vol.12 (10), p1033 –1051, Oct 2001

[14] S. Huss-Lederman, E.M. Jacobson, A. Tsao, "Comparison of Scalable Parallel Matrix Multiplication Libraries", Proc. Conf. Scalable Parallel Libraries, p142 –149, Oct 1993.

[15] K. Li, et al, "Fast and Processor Efficient Parallel Matrix Multiplication Algorithms on a Linear Array with a Reconfigurable Pipelined Bus System", IEEE Trans. on Parallel and Distrib. Sys., Vol.9 (8), p705 –720, Aug 1998

[16] M. Noh, Y. Kim, et al, "Matrix Multiplications on the Memory Based Processor Array", HPC Asia '97, p377 – 382, May 1997

[17] T. El-Ghazawi and F. Cantonnet, "UPC Performance and Potential: A NPB experimental Study", Proc. ACM/IEEE Conf. on Supercomputing (SC'02), p1-26, Nov. 2002

[18] R. Jin and G. Agrawal, "Performance Prediction for Random Write Reductions: a Case Study in Modeling Shared Memory Programs", Proc. ACM SIGMETRICS, p117 – 128, June 2002

[19] G. M. Zoppetti, G. Agrawal, L. Pollock, J. N. Amaral, X. Tang and G. Gao, "Automatic Compiler Techniques for Thread Coarsening for Multithreaded Architectures", Proc. Conf. on Supercomputing (SC'00), p306-315, May 2000

[20] M. A. Bender and M. O. Rabin, "Scheduling Cilk multithreaded parallel programs on processors of different speeds", Proc. ACM Sym. on Parallel Algorithms Architectures (SPAA), pages 13-21, July 2000

[21] A. Sodan, G. R. Gao, et al, "Experiences with non-numeric applications on multithreaded architectures". Proc. ACM SIGPLAN Symp. Principles Practice Parallel Programming, p124-135, June 1997

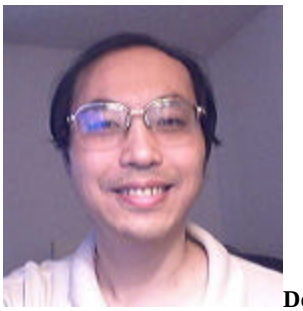
[22] P. Mehra, C. H. Schulbach and J. C. Yan "A Comparison of Two Model-based Performance-Prediction Ttechniques for Message-Passing Parallel Programs", Proc. ACM SIGMETRICS Conf. Meas. Modeling Comp. Systems, p181-190, May 1994

[23] C. Lin and L. Snyder, "ZPL: An Array Sublanguage," Proc. Worksh on Languages and Compilers for Parallel Computing'93, p96-114, Aug. 1993

[24] D. Jin and S.G. Ziavras, "Load-Balancing on PC Clusters With the Super-Programming Model", Worksh. Compile/Runtime Tech. Parallel Computing, (with ICPP03), Oct. 6-9, 2003

[25] S. Atlas, S. Banerjee, et al, "POOMA: A high performance distributed simulation environment for scientific applications", Proc. SuperComputing'95, Nov. 1995

[26] J. Hippold and G. Runger, "Task Pool Teams for Implementing Irregular Algorithms on Cluster of SMPs", Proc. IPDPS03 2003



Dejiang Jin received the B.S. in Chemical Physics from the University of Science and Technology, China in 1986 and the MS in Materials Science from Wuhan University of Technology in 1991. He is now a PhD student at New Jersey institute of Technology.

Dr. **Sotirios G. Ziavras** received the Diploma in EE from the National Technical University of Athens, Greece, in 1984, the M.Sc. in Computer Engineering from Ohio University in 1985, and the Ph.D. degree in Computer Science from George Washington University in 1990. He was a Distinguished Graduate Teaching Assistant at GWU. He was also with the Center for Automation Research at the University of Maryland, College Park, from 1988 to 1989. He was a visiting Assistant Professor at George Mason University in Spring 1990. He joined in Fall 1990 the ECE Department at NJIT as an Assistant Professor. He is currently a Professor and the Associate Chair for Graduate Studies. He also holds a joint appointment in the Computer Science Department. His work has been supported by NSF, NASA, DARPA, Dept. of Energy, AT&T, etc. He is an Associate Editor of the Pattern Recognition journal. He is an author/co-author of about 100 research and technical papers. He is listed, among others, in Who's Who in Science and Engineering, Who's Who in America, Who's Who in the World, Who's Who in Engineering Education and Who's Who in the East.