A SUPER-PROGRAMMING TECHNIQUE FOR LARGE SPARSE MATRIX MULTIPLICATION ON PC CLUSTERS

Dejiang Jin and Sotirios G. Ziavras
Department of Electrical and Computer Engineering
New Jersey Institute of Technology
Newark, NJ 07102

Abstract. The multiplication of large spare matrices is a basic operation for many scientific and engineering applications. There exist some high-performance library routines for this operation. They are often optimized based on the target architecture. The PC cluster computing paradigm has recently emerged as a viable alternative for high-performance, low-cost computing. In this paper, we apply our super-programming approach [24] to study the load balance and runtime management overhead for implementing parallel large matrix multiplication on PC clusters. For a parallel environment, it is essential to partition the entire operation into tasks and assign them to individual processing elements. Most of the existing approaches partition the given sub-matrices based on some kinds of workload estimation. For dense matrices on some architectures estimations may be accurate. For sparse matrices on PC, however, the workloads of block operations may not necessarily depend on the size of data. The workloads may not be well estimated in advance. Any approach other than run-time dynamic partitioning may degrade performance. Moreover, in a heterogeneous environment, statically partitioning is NP-complete. For embedded problems, it also introduces management overhead. In this paper We adopt our super-programming approach that partitions the entire task into medium-grain tasks that are implemented using super-instructions; the workload of super-instructions is easy to estimate. These tasks are dynamically assigned to member computer nodes. A node may execute more than one super-instruction. Our results prove the viability of our approach.

Keywords. 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 building block. Due to its fundamental importance, much effect has been devoted to studying and implementing MM [1-8]. MM has been included in several libraries [1, 6, 9-12,14. Some 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 high 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 matrices into sub-matrix blocks and decomposing the MM are often dependent technology [4,13]. Applying special implementations for sparse matrix operation to sub-matrix blocks may improve performance. It also makes the workload of sub-matrix operation to be diverse. The information about workload of a task for sparse matrix operation to sub-matrix blocks may not be available at compile time or even the time of initialing subroutines. It is only available after these routine programs have been executed. This increases the complexity of load balancing. So it is often ignored.

Most algorithms are optimized based on the characteristics of 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 of a cluster have a lot of resource, which can be used simultaneously. They have relatively weak communicating capability. So far, they lack high performance implementation support for data communication as efficient as supercomputer. It only has some communication channels implemented by software. Data communication still compete CPU resource with computation. So we need a new programming model to fit PC-cluster architecture better.

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

To address this problem, we use in this paper 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 strategies.

2. Super Program Development

2.1 Super Programming Model

In the super-programming model, the parallel system is modeled as a single virtual machine with multiple processing nodes (member Computers). [24] Application programs are modeled as super-programs (SPs) that consist of a collection of basic abstract tasks. These tasks are modeled as super-instructions (SIs). SIs are atomic workload units. SPs are coded with SIs. SIs are dynamically assigned to available processing units at run time, thus rendering the task assignment process a simpler task.

In SPM, the key elements are super-instructions. They are high-level abstract operations to solve application problems. In assembly language programming, each instruction in a program is indivisible and is assigned to respective functional units to execute. Similarly to assembly language programming, in SPM, each SI also is indivisible. Each SI can only be assigned and executed on a single node. At the PC executable level, they are some routines implementing frequently used operations for the corresponding application domain. Each SI can

execute on any member computer in the PC cluster. In heterogeneous environment, for a SI, there are many different executables corresponding to different computers.

The key difference of SPM compared to most of the other high-level programming models is that these super-instructions are designed in a workload-oriented manner rather than function-oriented. (More detailed comparisons follow in Section 4). They have limited workload rather than arbitrary workload. An example of such a SI is "multiply a pair of matrices in which the number of elements is not more than k". k is a design parameter. Programmers or compilers can expect that it can be implemented within a known maximum execution time on a given type of processing unit. In this manner, the system can schedule the task efficiently, thus minimizing the chance of member nodes being idle and reducing overhead of parallel computing. When the degree of parallelism in the SP, measured in number of super-instructions, is much larger than the number of nodes in the cluster, any node has little chance to be idle.

Super-instructions are basic atomic tasks with workloads that have a quit accurate upper bound. Application programs usually consist of logic functions based on the adopted algorithm. Application programs developed under this SPM are composed of SIs. These programs still can group SIs for assignment to logic function units. We call such logic functional units super-functions. A super-function achieves a logic operation with arbitrary workload. SI is workload-oriented. An SI may not achieve a complete logic operation but a part of a logic operation with predefined maximum workload. In SPM, super-functions are implemented with SIs. A super-function with little workload may be completed in one SI. In the general case, based on the designed parameters, a super-function, however, cannot be completed in just one SI, because of its limited workload. A number of SIs may be required. Breaking a super-function into a lot of SIs may increate the capability of the PC cluster to exploit the high-level parallelism in super-programs. As mentioned earlier, duo to communication overheads, it is difficult for PC clusters to exploit low-level parallelism. Thus, to successfully achieve computation parallel, it is critical to sufficiently exploit high-level parallelism. Independently assigning SIs, rather than super-functions, provides more high-level parallelism. 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. Increasing the degree of high-level parallelism makes easier the task of balancing the workload.

To effectively support application portability among different computing platforms and to also balance the workload in parallel systems, an effective instruction set architecture (ISA) should be developed for each application domain under SPM. These operations then become part of the chosen instruction set. 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.

2.2 Super-Instruction Set

An effective ISA should be developed for each application domain. These super-instructions should be frequently used operations for the corresponding application domain. Matrix multiplication is used as a basic

operation in many different applications. The program is often embedded in these host programs. So the super-instruction set for MM should be a common sub set of ISAs for these application domains. Also addition and multiplication of sub-matrix blocks are good candidates of super-instructions. Although matrix multiplication inherently involves the addition and multiplication of elements of operands matrices, it, however, can be achieved through the addition and multiplication of arbitrarily partitioned sub-matrices. In such a block-wise approach, the operands and products are all treated as much smaller matrices with elements also being matrices. Because the computing workload of multiplication and addition are determined by the size of the sub-matrix blocks, their workload can be easily limited by reducing the size of blocks.

The sparsity in matrices can be exploited to reduce the workload of these operations to improve the performance of the entire program. When using specific algorithms, the workload of these operations increases with the sparsity of the sub-matrix block but not more than that of the dense matrix.

For sparse matrices, generally, the non-zero elements may not be distributed in uniform manner. This means the sub-matrix blocks may have different sparisity. Some blocks may be sparser than entire matrix but some others may be denser than the entire matrix or may not even have any zero element. Since the workload of multiplication implemented with an optimal algorithm for sparse matrices depends on the sparsity of the matrix, this diversity of sparsity for sub-matrix blocks makes the workload of sub-matrix multiplication to vary substantially. We have chosen following super-instructions:

1) **denseToDenseMultiply**(A, B, C): A is an n_1 x n_2 dense matrix; B is an n_2 x n_3 dense matrix; C is an n_1 x n_3 dense matrix. The functionality of this SI is C = C+ A*B. n_1, n_2, and n_3 are all not greater than a pre-determined value k_1.
2) **sparseTodenseMultiply**(A, B, C): A is an n_1 x n_2 sparse matrix; B is an n_2 x n_3 dense matrix; C is an n_1 x n_3 dense matrix. The functionality of this SI is C = C+ A*B. n_1, n_2, and n_3 are all not greater than a pre-determined value k_2.
3) **sparseToSpareMultiply**(A, B, C): A is an n_1 x n_2 sparse matrix; B is an n_2 x n_3 sparse matrix; C is an n_1 x n_3 potentially dense matrix. The functionality of this SI is C = C+ A*B. n_1, n_2, and n_3 are all not greater than a pre-determined value k_3.

2.3 A Super-Program for Matrix Multiplication

We first discuss MM for distributed-memory parallel computers. We focus on an implementation for the ScaLAPACK library using 2D homogeneous grids [11]. We assume the problem C= A*B, where A, B and C are N x N square matrices; the elements of these matrices have the same size square sub-matrix blocks. We also assume that the computer system has $n = p^2$ nodes configured in a 2D square grids; Also N can be divided evenly by p so that N/p = q. The three matrices share the same layout over the 2D grid as follows: Each processor node $P_{r,s}$ (0 = r, s< p) holds all of the $A_{i,j}$, $B_{i,j}$ and $C_{i,j}$ sub-matrix blocks where i = rq+i', j = sq+j'and 0 = i', j'< q . Thus, any sub-matrix block $X_{i,j}$ (for X = A, B, C) is deterministically held by node $P_{r,s}$, where r = i /q and s = j /q .

Then, the MM algorithm is divides MM in N steps, where each step involves three phases. In the k-th step, where k = 1, 2, …, N, the three phases are:

1. Every node $P_{r,s}$ that holds $A_{i,k}$ (for all i =0, 1, … , q-1) multicasts the matrix block horizontally to all nodes $P_{r,*}$, where "*" stands for any qualified number.

2. Every node $P_{r,s}$ that holds $B_{k,j}$ (for all j =0, 1, … , q-1) multicasts the matrix block vertically to all nodes $P_{*,s}$

3. Every node $P_{r,s}$ performs the update $C_{i,j} = A_{i,k} * B_{k,j} + C_{i,j}$, for all $C_{i,j}$ held by the node.

Each node must have some temporary space to receive 2q matrix blocks.

Let us now focus on the SPM implementation of MM. For a super-program of MM in SPM, there are N^3 SIs. They are $I_{i,j,k}$ for updating $C_{i,j} = A_{i,k} * B_{k,j} + C_{i,j}$, for all of i, j, k =0, 1, … , q-1. If i? i' or j? j', then $I_{i,j,k}$ and $I_{i',j',k'}$ can be executed in parallel. If only k? k', then $I_{i,j,k}$ and $I_{i,j,k'}$, however, can only be executed in sequence. Since we lack a simple addition SI, they have to be executed on the same node. Under these constrains, any scheduling policy is acceptable.

3. Scheduling Strategies

Under the SPM model, super-programs can be implemented under various scheduling policies. Because of various factors that affect performance for MM, we discuss the following few scheduling policies.

3.1 Synchronous Policy

This policy makes super-program of MM imitate a implementation for distributed memory parallel computer (DMPC). A DMPC consists of a number of nodes. Each node has its own local memory, and there is no global shared memory. Nodes communicate with each other via message passing. Computations and communications in a DMPC are globally synchronized into steps. A step is either a computation step or a communication step. In a computation step, each node performs a local arithmetic/logic operation or is idle. A computation step takes a constant amount of time for all nodes. If a node finishes its work for the time slot, it also is idle for reminded time.

Under this policy, initially we determine the mapping of sub-matrix blocks of the product matrix $\{ C_{i,j} \}$ to processing nodes. It is similar to the layout described above. Each node is responsible of a part of the product matrix. But the area needs to neither a rectangle nor continuous. Scheduler schedules super-instructions based on the round and the mapping. In the k-th round, only SI $I_{i,j,k}$ (for all of i, j =0, 1, … , q-1) will be scheduled. Only corresponding updates are performed in this round computation step. When a node is available to execute another SI, the node is assigned an SI to execute if and only if there is an unassigned SI, which $I_{i,j,k}$ update a sub-matrix block that is mapped to the node. Otherwise, the node waits for the next round to begin.

Thus, the super-instructions belong to different rounds are never executed in parallel. All SIs belonging to the same round are executed pseudo-synchronously. So we call this policy synchronous policy. It is clear that this scheduling policy makes the super-program implementation similar to that for the algorithm described in the previous section.

3.2 Static Scheduling Policy

Similarly to the above synchronous scheduling policy, a static mapping of sub-matrix blocks of C onto processor nodes is determined at static time. But these SIs are not assigned to the nodes in a synchronous manner. Once a node is available to execute another SI, the scheduler checks to see if there exists such a qualified candidate SI waiting for execution; this candidate should correspond to the update of a matrix block that belongs to this node. If so, no matter what round this SI belongs to, the scheduler will assign the SI to this node for execution. Otherwise, it leaves the node idle. Thus, a node only executes the SIs that computes sub-matrix blocks statically assigned to it. Among these nodes, these SIs are executed asynchronously.

In a more restricted (k-inner) manner, the scheduler may first check to see if there exists a qualified candidate that involves the update of the same matrix block as the SI previously running on the node (with the same i and j). If yes, it schedules this SI. Otherwise, it schedules any qualified SI. Thus a node computes the sub-matrix blocks of the product one-by-one.

3.3 Random (Dynamic) Scheduling Policy

In this policy, once a node is free the scheduler first tries to find an unassigned SI that updates the same sub-matrix block with the previous SI running on this node (these SIs have the same i and j but different k). If it is successful, it assigns the new SI to the node. Otherwise, it checks to see if there exist some sub-matrix blocks that have never been updated. If so, it randomly chooses a sub-matrix block from this set and assigns an SI to update the block in this node.

Thus, any sub-matrix block can be assigned to a node. Once a block is assigned to a node, the node exclusively executes all the SIs that update the block. When a node needs more workload, the scheduler will randomly choose a sub-matrix block.

3.4 Smart (Dynamic) Scheduling with Random Seed policy

This scheduling policy is similar as above Random (Dynamic) Scheduling Policy. The only difference is the way of choosing a sub-matrix block. When a node needs more workload, the scheduler will choose a sub-matrix block from all unassigned sub-matrix blocks for the node based on its priority that is determined as follows:

1) For a node, an unassigned sub-matrix block $C_{i,j}$ has high priority if there exist both a sub-matrix block $C_{i',j}$ and a sub-matrix block $C_{i,j'}$ in the blocks that have been assigned to the node.

2) For a node, an unassigned sub-matrix block $C_{i,j}$ has medium priority if it has not high priority and there exist either a sub-matrix block $C_{i',j}$ or a sub-matrix block $C_{i,j'}$ in those blocks that have been assigned to the node.

3) For a node, an unbound sub-matrix block $C_{i,j}$ has low priority if there exist neither a sub-matrix block $C_{i',j}$ nor a sub-matrix block $C_{i,j'}$ in those blocks that have been assigned to the node.

The scheduler always assigns a block with highest priority to a node. If only low priority blocks exist, a random choice is made for assignment.

3.5 Smart (Dynamic) Scheduling with Static Seed policy

This policy is very similar to the above Smart Scheduling with Random Seed policy except that we now provide a scheme to choose the first block assigned to each node. When scheduler assigns a sub-matrix block to a node initially, the list of blocks assigned to it is empty and all unassigned candidate blocks have low priority. In this special case, the scheduler chooses a block based on a 2D grid arrangement, like static scheduling policy rather than a random choice.

4. 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 machines. There exist tools to run programs developed under any of these models for execution on any type of parallel architecture. For example, a language that follows the shared-memory model can be correctly implemented on both shared-memory architectures and share-nothing multi-computers; this is also true for a language that follows the message-passing model. 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. Cilk also follows the shared-memory model [20].

The optimization of parallel application programs 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 of the shared-memory model 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.

An array programming language, like ZPL, exploits only data-level 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 for high performance. 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 for 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. Load balancing is done implicitly by the programmer though data partitioning. For task-oriented approaches, load balancing is normally performed at run time through a combination of assigning tasks to processors, and often preempting and/or migrating tasks at run time, as deemed appropriate by the run-time environment. Cilk uses thread migration to balance the workload. Charm++ behaves similarly; it encapsulates

working threads and relevant data into objects. The system balances the workload by migrating these objects at run time.

Our SPM model employs load control technology for load balancing. Only a limited number of SIs are originally assigned to processors, where each SI has limited workload. All other SIs are left in an SI queue. When a processor completes the execution of an SI, the system automatically assigns it another SI. This mechanism guarantees that no processor is either overloaded or underloaded. Thus, the workloads are balanced automatically among the processors. Our approach, similar with Cilk, employs a greedy utilization scheduler at run time. However, our scheduling approach is not based on processes and does not migrate threads, thus eliminating the overhead of task migration. On a shared-memory system, thread migration with Cilk has very low overhead (the implementers claim 2-6 times the overhead of a function call). However, this cost does really exist. [20] suggests a high utilization schedule on a shared-memory system may be better than a maximum utilization schedule. It means that if the difference in processor speeds is not large enough, migrating a thread from a slow processor to a faster processor is not better than keeping the thread on the original (slow) processor. On 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.

Generally, SPM is a non-preemptive, multi-threaded execution model. It was first tested with a data-mining problem [24].

In the above mentioned projects, the life time of threads is determined by the programmer or compiler. In SPM, it is controlled by the run-time environment. The programmer only requests thread creation but threads are created by the runtime system based on a load control policy. Load control technology has been used for many years to improve the performance of multitasking in centralized (single processor) computer systems. Both Cilk and Charm++ do not use any load control. When a function is invoked, the runtime environment of Cilk creates a new thread and adds the load to a processor of the system no matter how busy the system is. The threads are similar to SPM SIs. But an SI under SPM is assigned to a processor only when enough resources are available. Thus, SIs never need to be migrated.

In all of the parallel programming models, the addressable data are treated similarly to data for the sequential programming model. In the message passing model, an application uses a set of cooperating concurrent sequential processes. Each process has its own private space and runs on a separate node. Except for message exchanges among these processes, the processes behave like sequential. For the shared memory model, the programmer can access any non-hidden data. For example, in Cilk or UPC programmers can access all data as ANSI C. In SPM, similarly to the shared-memory model, there is global data that has higher-level abstraction. Primary data are super-data blocks which are pure logic objects in the objects domain and are associated with a given application domain; the primary operations are carried out by super-instructions. These super-data may map to a large block of underlying data by the runtime system. But unlike all the aforementioned models, programmers can operate on an object only as an atom with an SI. They can never access their internal components. Thus, application

programmers can only access these objects rather than lower-level block structures. This gives programmers a coarse-grain data space.

5. Performance Analysis

Let us begin with some definitions. The ideal (inherent) workload W_{ideal} of a piece of program is defined as the total number of its ordinary instructions for its optimal sequential implementation. The ideal (inherent) performance P_{ideal} of a computer is defined as the rate to execute ordinary instructions per second. The shortest execution time is $T = W_{ideal} / P_{ideal}$. The ideal (peak) performance of a cluster system with multiple computers is defined as the sum of the performance of all member computers. The (real) workload W of a parallel program run on a PC cluster is defined as the product of the overall execution time and the system (peak) performance. ($W = (? P_i)* T$). The workload W is definitely larger than the ideal workload W_{ideal}. The difference between W and W_{ideal} is the so called overhead.

When a super-problem is implemented in parallel and the program runs on multiple nodes, the system needs to use not only its computing resources to solve the problem but its communications resources as well in order to exchange data with other nodes. For example, when a super-program runs on a PC cluster, the system not only executes SI on its member nodes but it may also need to remotely load operands for the SIs. The latter will increase the workload. It is the so called *communication overhead* W_{comm}. When the tasks distributed among the nodes are not balanced, some nodes will finish their tasks before others. This means that these nodes will be idle for some time. This increases the overall time of solving the problem. We called this the *unbalance overhead or idle overhead* W_{idle}. The scheduler also needs to carry out some computation to correctly schedule SIs. It will introduce some *management overhead* W_{man}.

Thus, $W = W_{ideal} + W_{idle} + W_{comm} + W_{man}$ (4-0)

The relative overheads are $w_{idle} = W_{idle} / W_{ideal}$, $w_{comm} = W_{comm} / W_{ideal}$ and $w_{man} = W_{man} / W_{ideal}$ respectively

Below we will build overhead models to analyze unbalance overhead of SPM program. Only simply discuss management overhead at end of Section 5.

Under this model, a parallel program is modeled as a sequence of synchronous steps. Each step ends with synchronization of the nodes, but inside a step there is not any synchronization needed. Within a step all nodes execute as quickly as possible all tasks assigned to them in this step. Then, they wait for all the nodes to finish their assigned tasks. In this case, the unbalance overhead is the sum of the products of the idle times $T_{i\text{-}idle}$ of each node i multiplied by its peak performance P_i (i.e $W_{idle} = ? P_i * T_{i\text{-}idle}$). In a homogeneous environment ($P_i = P_0$), the unbalance overhead is $W_{idle} = P_0 * ? (T_{i\text{-}idle})$. For matrix multiplication, each sub-matrix can be computed independently. There is no inherent synchronization requirement. Super-programs for matrix multiplication can have only one synchronous step. A synchronous algorithm with a synchronous policy, however, may split the program into many steps requiring synchronization.

For the sake of simplicity, without loss of generality, we assume that both operands of multiplication, A and B, are partitioned into sub-matrix blocks of size N x N. The product matrix C is also partitioned into sub-matrix

blocks of size N x N. All of these sub-matrix blocks are n1 x n2 matrices. We assume that the workload of an SI, which multiplies a pair of sub-matrices A_{ik} * B_{kj} and updates the block C_{ij}, conforms to statistical rule R_0 corresponding to average workload L_0 and deviation D_0. Also, the program is executed on a PC cluster with $n = p^2$ nodes.

For the synchronous algorithm, in each step a node carries out (N/p)*(N/p) multiplications. Each is applied for a sub-matrix of C. Thus, in each step the expected workload of a node is

$L_{syn} = (N/p)*(N/p)* L_0$ (4-1)

and the deviation is $D_{syn} = (N/p)* D_0$. We still estimate the practical maximum workloads $L_{max\text{-}syn} = L_{syn} + D_{syn}$. Then, the unbalance overhead of the system in a given step is $(n-1)*(L_{max\text{-}syn} – L_2) = (n-1)* D_2$ and the total unbalance overhead of the system for the N steps is

$W_{idle\text{-}syn} = N* (n-1)* D_{syn} = (n-1)/n^{1/2} * (N^2 * D_0) \sim n^{1/2} * (N^2 * D_0)$ (4-2)

For static scheduling, each node is responsible of computing (N/p)*(N/p) sub-matrices and the computation of a sub-matrix of C involves N steps of multiplying a pair of sub-matrices A_{ik} * B_{kj}. If N is large enough, according to the large number theorem the expected workload of a node is $L_s = (N/p)*(N/p)*N* L_0$ and the deviation is $D_s = \{(N/p)*(N/p)*N\}^{1/2}* D_0 = N^{3/2} / n^{1/2} * D_0$. We estimate the maximum workload in the p^2 nodes to be $L_{s\text{-}max} = L_s + D_s$. Then, the unbalance overhead of system is

$W_{idle\text{-}s} = (n-1)*(L_{s\text{-}max} – L_s) = (n-1)* D_s = (n-1)/n^{1/2} * (N^{3/2} * D_0) \sim n^{1/2} * (N^{3/2} * D_0)$ (4-3)

For any dynamic scheduling, we assume that the unit of assignment corresponds to computing a sub-matrix of C rather than performances A_{ik} * B_{kj} . In this case, the expected workload of an assignment is $L_1 = N* L_0$ and the deviation is $D_1 = N^{1/2} * D_0$. When the last assignment is made to a node, all other nodes are doing some computation and do not have any chance to be idle. If we estimate the average workload reminded for other nodes to be half of L_1 , then the expected value of unbalance overhead is

$W_{idle\text{-}dyn} = (n-1)*1/2 * L_1 = (n-1)*N/2 * L_0 \sim n/2 *N * L_0.$ (4-4)

Comparing $W_{idle\text{-}s}$ with $W_{idle\text{-}syn}$, we can find $W_{idle\text{-}syn} = N^{1/2} * T_{idle\text{-}s}$. Thus, synchronous algorithm always has $N^{1/2}$ times larger unbalance overhead than an asynchronous algorithm with a static scheduling policy. We also have

$W_{idle\text{-}s} / W_{idle\text{-}d} = 2 * (N/n)^{1/2} *(D_0/ L_0)$ (4-5)

Thus, when the deviation of the basic operation (it corresponds here to the multiplication of a pair of sub-matrices) is insignificant, then static scheduling has less unbalance overhead. However, if the deviation of the basic operation is significant, when $N > n *(L_0/ D_0)$, $T_{uo\text{-}s} / T_{uo\text{-}d} > 1$. Thus, static scheduling has heavier unbalance overhead than any dynamic scheduling.

6. Simulation

6.1 Simulation Program

Any SI is simulated as an abstract task with a workload that is determined at static time. A member computer is simulated as an abstract “Node” object with a peak performance parameter. Each node records its

accumulated idle time, the time for loading data and the time for executing SIs. It also keeps track of all tasks that have assigned to the node, the next task it will finish and the time it finishes the task. All nodes are put in a queue based on the time they will finish execution of their next task. Scheduling algorithms are implemented rather than simulated. After nodes have been assigned their initial tasks, in each step simulator finds the next task to be ended, get the time that the task would be finished and the node that executes the task. The system time is updated to the time that the task would be finished, which is published to all nodes; the scheduler schedules new tasks to the node; and then the node would be put into queue if it is still active. The simulator keeps running until all SIs have been executed and all nodes have become idle. In our simulation, the static schedule was created manually in advance. Except for some parameters that were determined at run time by the scheduler, all others simulation parameters were generated in advance based on the statistical distribution features of the parameters. These include the workloads of the SIs. Except for the simulation of synchronous multicast, in simulations tasks scheduled as a group of SIs update the same sub-matrix block in C.

6.2. Parameters Chosen

The average time to execute an SI is 10000 units of time. The parameter is a basic parameter. Some networking parameters are chosen based on some pilot program and others are chosen arbitrarily. We choose the size of all matrices to be 32 x 32. All elements of them are sparse square matrices having same dimension. We assume that the workload of SIs conform with the uniform distribution and the deviation of maximum/minimum values is the same as the average value. This means that the workload is distributed uniformly between 0 and 20000.

The system runs under the CPU bound condition. For loading a sub-matrix block remotely, a sending node requires 250 units of time and a receiver take 250 units so that the total cost to load the two operands of an SI remotely is 10% of its execution time. In our pilot program, we found for our PC cluster system that takes about 20% CPU time to handle high-level object communication when the networking resource is fully used. Under the CPU bound condition, the networking resource may not be fully used. So we assume that the total percentage of time to load the operands of an SI remotely is 10%.

The number n of PCs in the cluster is chosen as 1, 2, 4, 8, 16, 32, 64, 128 and 256. In homogeneous environment, the peak performance of any node is 1 unit. In heterogeneous environment, the relative peak performance of any node with an odd index is 3 units and the relative peak performance of any node with an even index is 1 unit.

For the static strategy, the layout of sub-matrix blocks statically is scheduled as shown in Table 1 for 4 nodes. The left column and the top row are the indices of sub-matrix blocks. The numbers in the tables are the index of the nodes that the sub-matrix block is assigned to. For other cases with a different number n of nodes in the cluster (n = 1 to 256), similar schemes are given. For the Smart scheduling with Static seed (SS strategy), the initial task assigned to each node is chosen to be the same as that for the static strategy.

Table 1 Static schedule for a cluster with 4 nodes

(a) a homogeneous environment

	0-15	15-31
0-15	0	2
15-31	1	3

(b) a heterogeneous environment

	0-11	12-19	30-31
0-15	0	1	2
15-31	0	3	2

For multicasts, the membership of nodes in the multicast group is determined statically as follows for all strategies. For multicasting the sub-matrix blocks of matrix A, all nodes located in a same row in Table 1 form a group; for multicasting the sub-matrix blocks of matrix B, all nodes located in a same column in Table 1 form a group. When a node requests a data block, all nodes in the same group will receive this data block.

In our simulation, the sub-matrix blocks of A and B are cached separately. The unit of cached data is a row of sub-matrix blocks of A and a column of sub-matrix blocks of B. The size of the local cache is the number of rows of A and columns of B cached. The size of the cache is chosen to be 1-32. The size of 32 means that every node can cache 32 rows of A and 32 columns of B. Therefore, every node can fully cache all data it needs.

All simulations repeat 16 times with different data sets. The results are the average values of multiple simulation runs.

6.3. Results of Simulations

Simulations results of for different scheduling algorithms in a homogeneous environment are shown in Fig 1. Syn represents the synchronous algorithm; S represents the static scheduling algorithm; R represents the random scheduling algorithm; SS represents the smart scheduling algorithm with a static initial scheme; SR represents the smart scheduling algorithm with a random initial scheme. Excepts for Syn, all others try to use both unicast and multicast operations for data communications.

The results for different scheduling algorithms in a heterogeneous environment are shown in Fig. 2.

The execution times of the simulation program in various cases are shown in Fig.3. They are the upper bound of management overheads. u and m represents the program that uses unicast and multicast for communication, respectively. H and R represents the programs run in a homogeneous and heterogeneous environment, respectively.

From Fig.1, we can find that synchronous algorithms has much higher unbalance overhead although the synchronous, algorithm are accepted widely in parallel implementations for MM. For high performance, it is favored to asynchronously update those independent sub-matrix blocks to perform MM on the PC cluster. Thus,

when there exists diverse workload between updating different sub-matrix blocks, the super programs in SPM

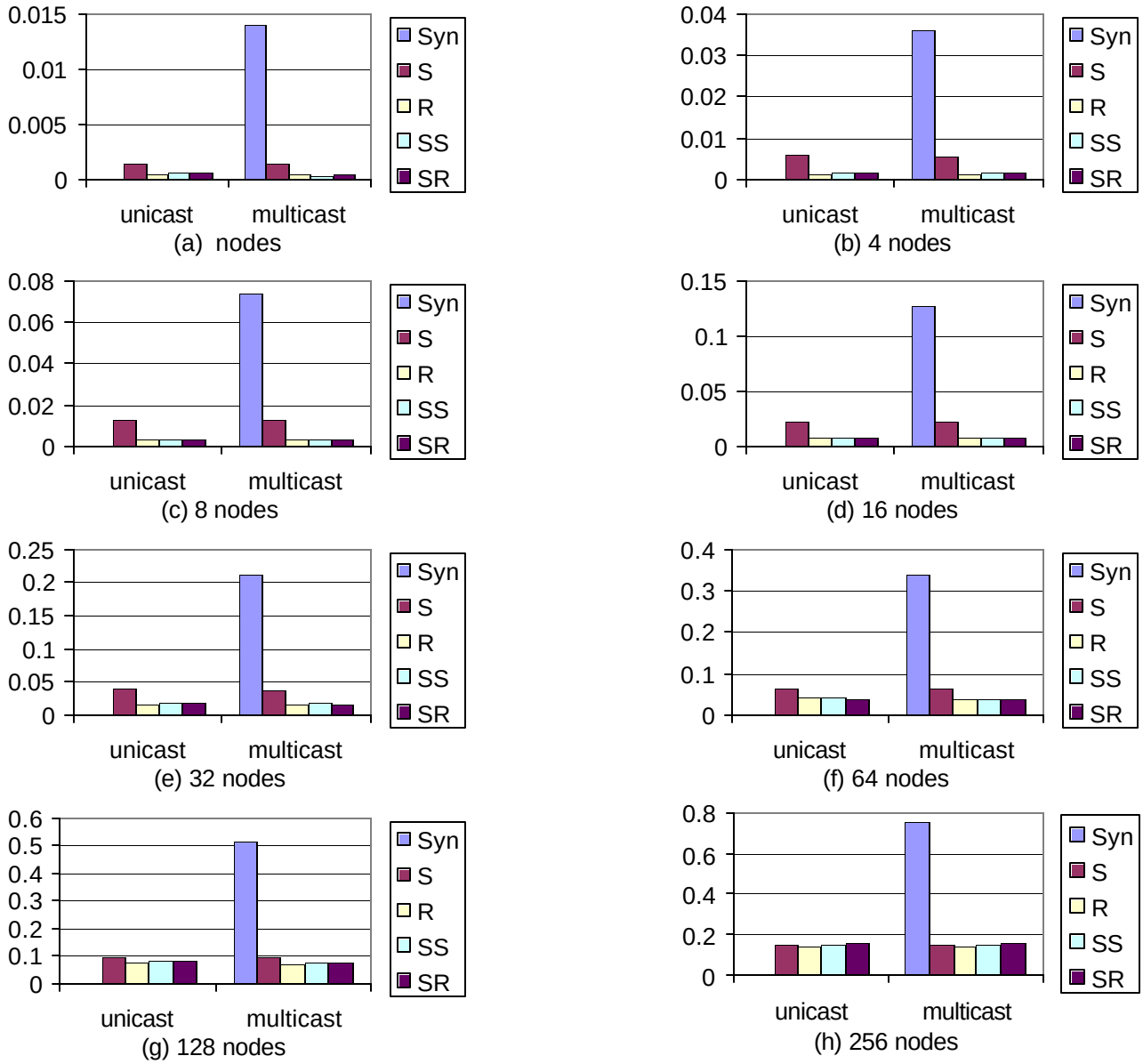


Fig 1 comparison of unbalance overhead of super-programs with different strategies in homogeneous environment for PC clusters with different number of nodes.

are more appropriate to perform MM in the PC cluster environment than the programs in synchronous fashion for synchronous fashion for DMPC.

This set of results also shows that dynamic scheduling algorithms always have less unbalance overhead than static scheduling algorithms no matter if unicasting or multicasting is used for data communication. The

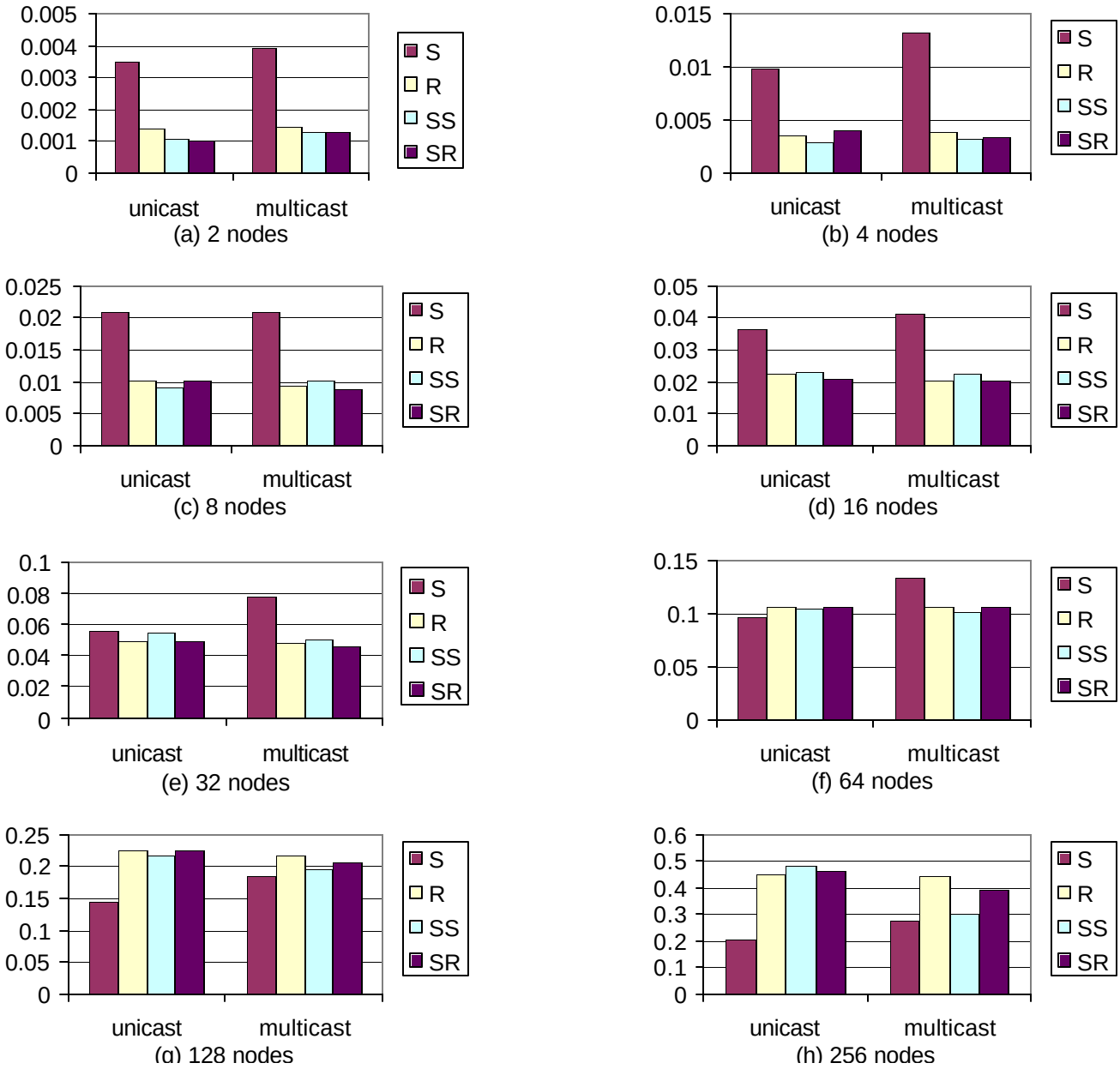


Fig 2 comparison of unbalance overhead of super-programs with different strategies in heterogeneous environment for PC clusters with different number of nodes.

difference in the unbalance overhead between static and dynamic scheduling depends on the average number of tasks executed on a node . The more tasks a node executes, (less nodes in the cluster), the larger the difference. When there are only 48 tasks per node (case g and h), the different is small. In other cases, the different is significant. Thus, for large problems, dynamic strategies have an advantage to reduce the unbalance overhead. The difference in the unbalance overhead among different (dynamic) scheduling strategies,

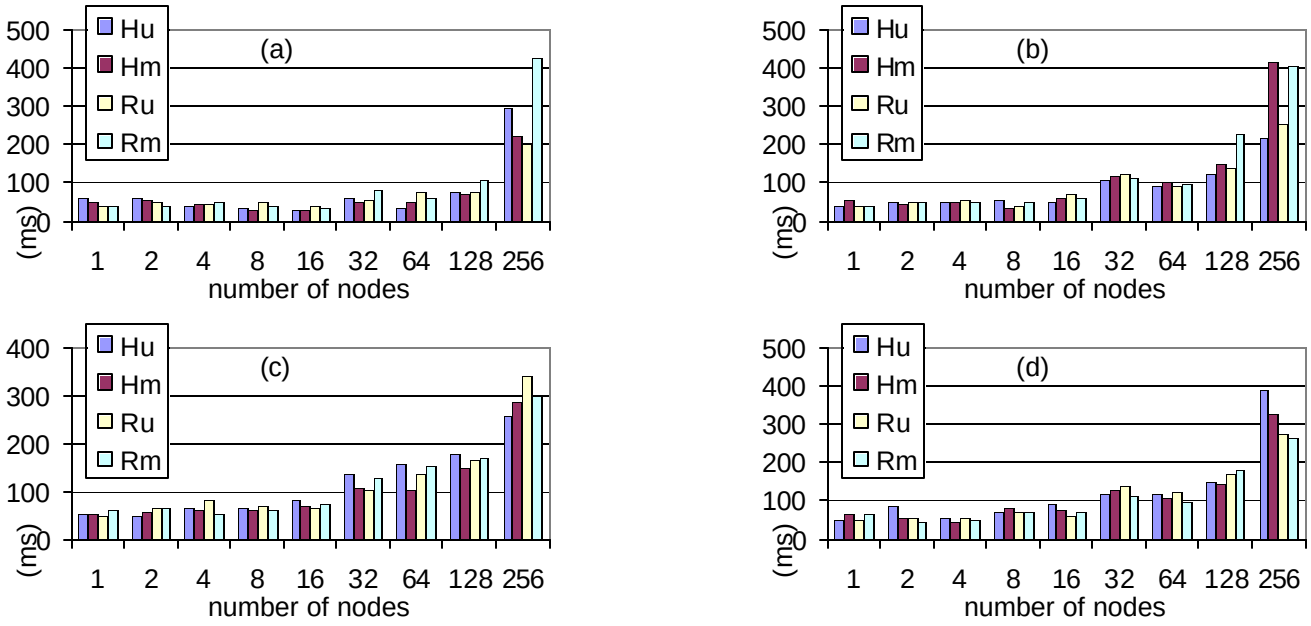


Fig 3 comparison of management overhead of super-programs with different strategies

(a) static strategy (c) smart strategy with static initials

(b) random strategy (d) smart strategy with random initials.

however, is very small in a homogeneous environment

Comparing the results for a heterogeneous environment shown in Fig2 with corresponding results for a homogeneous environment shown in Fig 1, we can see that the basic feature is similar. But the unbalance overhead increases. This is because in a heterogeneous environment the nodes have different peak performance. The slower nodes have more chance to execute the last SI. Other nodes with higher peak performance may be idle at the same time, thus wasting more computation capability. The increase in the dynamic strategies is larger than the increase in the static strategy. This makes the advantage of dynamic strategies to decrease. The critical value is increased to 32 tasks per node. Fig 2 (e). When there are only 4-8 tasks per node, (case g and h), the static strategy becomes the favor. This indicates that in a heterogeneous environment the dynamic strategies also are favorable to reduce the unbalance overhead. But they only are helpful for large problems

In Fig.3, we can know that the execution times of simulation programs are all less than 400ms. Many cases, the execution times are less than 100 ms. Since the scheduling algorithms were implemented rather than simulated, the management overhead for both static and dynamic scheduling is less than 400 ms for the multiplication of two 32x32 matrices. Considering there that are 32K SIs to be executed in multiplying two 32x32 matrices, and each SI may take a few ms, the relative management overhead of SP is very low. The average time to execute an SI depend on the size of sub-matrix blocks that is a designed parameter. If the average time is 4ms, the maximum management overhead is only 0.3% of total execution time. In many cases, they are less than 0.1%. Compared to the unbalance overheads, it can be ignored. Comparing the figures for

different scheduling strategies, we can find that smart scheduling strategies have a little higher management overhead than the random strategy and the static strategy. But the difference is small. The execution times to simulate super-programs in a heterogeneous environment have not big different from those in a homogeneous environment in all cases. Thus, it is appropriate to applying these strategies in both a homogenous and heterogeneous environment

It should be indicated that, for embedded problems using the static algorithm, finding the static scheme makes some additional management overheads. The research in [13] proves that in a heterogeneous environment finding the optimal static scheme itself is not a trivial task. It is an NP-complete problem. A dynamic approach does not have this additional overhead. All management overheads is included in the implementation of scheduling.

7. Experimental Results on a PC Cluster

All the experiments were performed on a PC cluster with eight nodes. The random/dynamic scheduling approach was followed. Each dual-processor node is equipped with AMD Athlon processors running at 1.2 GHz. Each node has 1GB of main memory, a 64K Level-1 cache and a 256K Level-2 cache. All the nodes are connected through a switch to form an Ethernet LAN. Each link has 100Mbps bandwidth. All the PCs run Red hat 9.0 and have the same view of the file system by sharing files via an NFS file server. All data are obtained from this file server. We used a set of synthetic sparse matrices of size 8192x8192; they contain 5% non-zero elements. These matrices were partitioned into 32x32 sub-matrices of size 256x256. Non-zero elements were generated randomly and contained double-precision floating-point numbers between 1.0 and 2.0. The location of non-zero elements was chosen randomly. The super-program was developed manually using the super-instructions described in Section 2. We implemented a simple runtime environment (i.e., a virtual machine), which supports our super-programming model. The runtime environment consists of a number of virtual nodes (each residing in a PC node). Super-instructions are scheduled to run on these virtual nodes. The runtime environment also caches recently used super-data blocks. The runtime environment and the super-instructions were implemented in the Java language. A special function was added in the code to collect information at runtime about the execution of Sis and the utilization of individual PCs. The execution times for matrix multiplication were extracted from this collected time information.

To check how many threads were needed for the system to run under the CPU bound condition, we first run the program on just two PCs. Each PC was allowed to launch n threads to execute n SIs simultaneously, for n = 4, 8, 12 and 16. The results are shown in Fig. 4. The execution with 4 threads has much higher execution time. Using 8 or more threads for each node, the total execution times are very close. Thus, 8 threads can hide the long latency of loading remotely super-data blocks (operands). To check the speedup of the program, we run it with n nodes, for n = 1, 2, 3, 4, 5, 6 and 8. The results are shown in Fig. 5. When using 4 threads on each node,

the system may run under the delay bound condition. Even the speedup in this case (4T) is better than the case (8T) where 8 threads are allowed in each node, the total performance in the former case is worse than the latter.

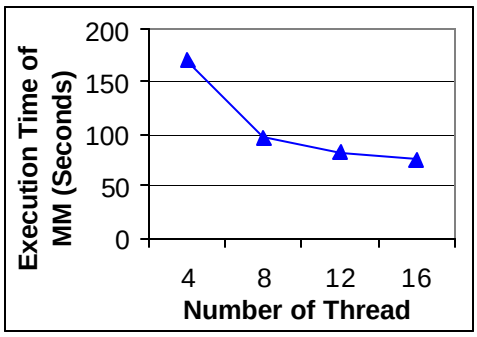
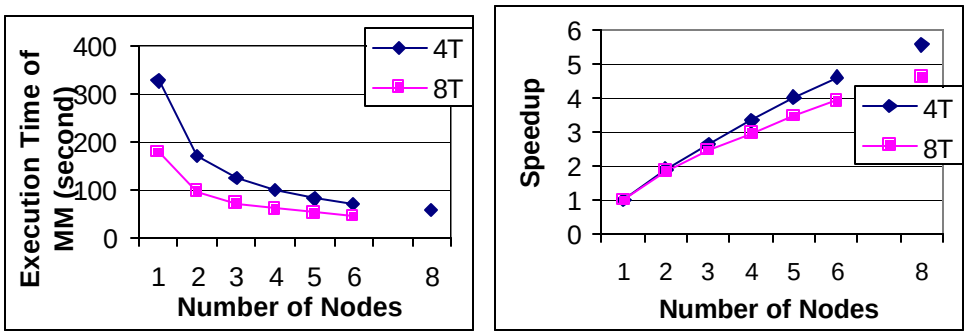


Fig. 4. The effect of the number of threads per node on the performance of the super-program



(a) (b)

Fig.5. The execution time and speedup for different numbers of nodes

8. Conclusions

1. Super-programs in SPM can exploit high-level parallelism of problems by executing coarse/medium-grain tasks parallel. SPM is appropriate to parallel implement various algorithms.
2. The overhead of performing sparse matrix multiplication in parallel involves not only communication overheads but also unbalance overheads. For an embedded problem, it also involves management overhead. If the workload of block operations varies, such as for sparse matrix multiplication, the algorithms with fine–level synchronization have high unbalance overhead. Algorithms using coarse-grain tasks have usually less unbalance overhead. Thus, exploiting high-level parallelism makes super-programs can execute tasks asynchronously. This can avoid high unbalance overhead. Dynamic scheduling of super-programs can reduce the unbalance overhead further effectively. Algorithms using dynamic scheduling policies have less unbalance overhead than those involving a static scheduling policy.
3. The management overhead for dynamic algorithm is similar among different scheduling strategies. There is no additional management overhead for embedded problems no matter in a homogeneous or heterogeneous environment. All management overheads are included in the execution of scheduler. They are insignificant.

In a heterogeneous environment, management overhead of static strategy for embedded problems is expected to be high. Thus, super-programs in SPM with dynamic strategies have a great advantage in a heterogeneous PC cluster.

References

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

[2] K. Li, Scalable Parallel Matrix Multiplication on Distributed Memory Parallel Computers**,** *Proc. Parallel and Distributed Processing Symposium (IPDPS), 2000.* p307 –314, 2000

[3] T.I. Tokic, E.I. Milovanovic, N.M. Novakovic, I.Z. Milovanovic, M.K. Slojcev, Matrix Multiplication on Non-planar Systolic Arrays**,** *4th International Conf. on Telecommunications in Modern Satellite, Cable and Broadcasting Services, 1999*, Vol2, p514 -517, 1999

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

[5] Jaeyoung Choi**,** A New Parallel Matrix Multiplication Algorithm on Distributed-Memory Concurrent Computers**,** High Performance Computing (HPC) Asia '97, p224 –229, May 1997

[6] Iain S. Duff , Michele Marrone , Giuseppe Radicati , Carlo Vittoli, Level 3 Basic Linear Algebra Subprograms for Sparse Matrices: a User-Level Interface, *ACM Trans. on Math. Software (TOMS) Vol23* (3), p379-401,Sept. 1997

[7] A.E. Yagle, Fast Algorithms for Matrix Multiplication Using Pseudo-Number-Theoretic Transforms**,** *IEEE Trans. on 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. 9th Int. Parallel Processing Sym.*, p436 –440, Apr 1995

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

[10] Bo Kågström , Per Ling , Charles van Loan, GEMM-based Level 3 BLAS: High-performance Model Implementations and Performance Evaluation Benchmark, *ACM Trans. on Math. Software (TOMS) Vol24* (3), p268-302, Sept. 1998

[11] Jack J. Dongarra , L. S. Blackford , J. Choi ,A. Cleary , E. D'Azeuedo , J. Demmel , I. Dhillon , S. Hammarling , G. Henry , A. Petitet , K. Stanley , D. Walker , R. C. Whaley, *ScaLAPACK user's guide* (Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997)

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

[13] O. Beaumont, V. Boudet, F. Rastello, Y. Robert, Matrix Multiplication on Heterogeneous Platforms *IEEE Trans. on Parallel and Distributed Systems, 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] Keqin Li, Yi Pan, Si Qing Zheng, Fast and Processor Efficient Parallel Matrix Multiplication Algorithms on a Linear Array with a Reconfigurable Pipelined Bus System**,** *IEEE Trans. on Parallel and Distributed Systems, Vol.9* (8), p705 –720, Aug 1998

[16] Mi-Jung Noh, Youngsik Kim, Tack-Don Han, Shin-Dug Kim, Sung-Bong Yang, Matrix Multiplications on the Memory Based Processor Array**,** *High Performance Computing on the Information Superhighway, (HPC) Asia '97*, p377 –382, May 1997

[17] T. El-Ghazawi and F. Cantonnet, UPC performance and potential: A NPB experimental Study, *Proc. 2002 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, *ACM SIGMETRICS intern.conf. on Measur. and Model. of computer systems*, p117 – 128, 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. 14th intern. Conf. on Supercomputing (SC'00),* p306-315, may2000

[20] M. A. Bender and M. O. Rabin, Scheduling Cilk multithreaded parallel programs on processors of different speeds *Proc. 12th ACM Symposium on Parallel Algorithms and Architectures (SPAA), pages 13-21, 2000.*

[21] A. Sodan, G. R. Gao, O. Maquelin, J.Schultz and X.Tian Experiences with non-numeric applications on multithreaded architectures . *Proc. of 6th ACM SIGPLAN symposium on Principles & practice of parallel programming,* Vol32 (7)

[22] P. Mehra, C. H. Schulbach and J. C. Yan A comparison of two model-based performance-prediction techniques for message-passing parallel programs *Proc. 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, Vol. 22 (1) p181-190, 1994

[23] C. Lin and L. Snyder, "ZPL: An Array Sublanguage," *Languages and Compilers for Parallel Computing*, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, eds, pp. 96-114, 1993.

[24] D. Jin and S.G. Ziavras, Load-Balancing on PC Clusters With the Super-Programming Model, *Worksh. Compile/Runtime Techniques for Parallel Computing, (ICPP03)*, Kaohsiung,Taiwan, Oct. 6-9, 2003 (to appear).

[25] S. Atlas, S. Banerjee, J. Cummings, P.J. Hinker, M. Srikant, J.V.W. Reynders, and M. Tholburn, POOMA: A high performance distributed simulation environment for scientific applications, *SuperComputing '95*