Task Pool Teams for Implementing Irregular Algorithms on Clusters of SMPs

Judith Hippold[*] and Gudula Rünger
Chemnitz University of Technology
Department of Computer Science
09107 Chemnitz, Germany
{judith.hippold, ruenger}@informatik.tu–chemnitz.de

Abstract

The characteristics of irregular algorithms make a parallel implementation difficult, especially for PC clusters or clusters of SMPs. These characteristics may include an unpredictable access behavior to dynamically changing data structures or strong irregular coupling of computations. Problems are an unknown load distribution and expensive irregular communication patterns for data accesses and redistributions. In this paper we propose task pool teams for implementing irregular algorithms on clusters of PCs or SMPs. A task pool team combines multithreaded programming using task pools on single nodes with explicit message passing between different nodes. As application example we use the hierarchical radiosity algorithm.

1 Introduction

Irregularity of algorithms may be caused by different characteristics including unpredictable accesses to data structures due to sparsity or dynamic changes, varying computational effort because of refinement or adaptivity, and irregular dependencies between computations. Examples are sparse linear algebra methods like sparse Cholesky factorization, grid-based codes with dynamic refinements like adaptive FEM, or hierarchical algorithms like the fast multipole or hierarchical radiosity algorithm. Although most hierarchical and adaptive algorithms have been invented to save computation time while still getting a good solution, there is still need for a parallel implementation since the computation time for those methods can be quite large when realistic problems are considered. The computational characteristics of irregular algorithms can be different, but all irregular algorithms have in common that the actual program behavior of a specific program run strongly depends on the specific input data of the program. Thus, not much information is statically available and static planning of parallelism is difficult. Especially when an irregular algorithm has unpredictably evolving computational work, dynamic load balance is required to employ all processors evenly.

For shared memory platforms the concept of task pools can be used to realize dynamic load balance. The basic idea is to manage tasks in a special global data structure called task pool, see e.g. [4]. Each processor can take tasks from the pool and add new tasks to the pool until the entire computation is done. Task pool implementations for dynamic load balance have been presented in e.g. [13, 16]. Detailed investigations of different versions are given in [11].

For distributed memory machines there is a close connection between dynamic load balance and communication since a redistribution of work at runtime can only be achieved with message passing. For an efficient parallel implementation a trade-off between decreased runtime due to balanced load and increased runtime due to communication overhead has to be found. Furthermore the unpredictable access behavior forces communication for the exchange of data structures.

This paper introduces the realization of irregular algorithms on PC clusters or clusters of SMPs (symmetric multiprocessors) with a new approach, called *task pool teams*. A task pool team combines the concept of task pools for load balancing on individual nodes with explicit communication for irregularly occurring, remote data accesses. We have implemented task pool teams on top of the Message Passing Interface (MPI) and POSIX Threads (Pthreads), although the concept is more general. Thus, the resulting implementation of an irregular algorithm with task pool teams is entirely realized on the application programmers level. The advantage is that the mechanism of task pool teams can be used for an efficient parallel implementation while the application programmer can still exploit algorithm specific properties explicitly.

Our hybrid programming model is suitable for PC clusters or clusters of SMPs and for applications with arbi-

[*]Supported by DFG, SFB393 *Numerical Simulation on Massively Parallel Computers*

trary task graphs or arbitrary dynamic data structures. We illustrate and test the model with the hierarchical radiosity method, a global illumination method from computer graphics. This application has several of the properties of irregular algorithms mentioned above. Data are organized in dynamically growing quadtrees and computations are guided by varying interactions between nodes of those quadtrees. The resulting parallel algorithm is tested on two platforms, a Beowulf cluster (CLiC) and a cluster of SMPs.

We have implemented several variants for task pools and experimented with different approaches for the combination with communication to form task pool teams. Special care was taken to design the communication mechanism between remote threads running on different cluster nodes. General problems are the absence of thread-safe communication in the implementation environment or a potential of deadlocks for blocking communication operations in irregular thread-based programs.

The paper is structured as follows: Section 2 presents concepts and realizations of task pools. Section 3 introduces the task pool team approach. Section 4 describes the hierarchical radiosity algorithm (HRA) and discusses special requirements and adaptations with regard to the task pool team realization. Experiments and measurements are presented in Section 5. Section 6 discusses related work and Section 7 concludes.

2 Task pools for shared memory

For the implementation of task pool teams we assume the following basic programming model on a cluster: There is one process running on each cluster node, which can be a one-processor machine or an SMP with several processors. A process consists of a virtual address space and one or more threads of control which are executed by timesharing on the processors of the node. The interaction between processes running on different nodes with different address spaces is realized by explicit message passing. The task pool team approach considers task pools for clusters with one task pool for each cluster node.

This section presents the task pool concept for a single cluster node. The actual combination of task pools with message passing and specific strategies for implementing task pool teams for entire clusters are introduced in the next section.

2.1 General approach

Application programs realized with task pools are structured into a set of interacting tasks. Each task consists of a well-defined sequence of commands often captured in a function or procedure to be executed by a single thread of the process. The commands can include the creation of child tasks which can later be executed by a different thread of the same process. Although the task creation and execution is coded within the implementation of the algorithm, each run of a specific program may actually create a different task graph, especially for irregular algorithms.

A task pool is a shared data structure to store and manage the tasks created for one specific program. All threads of the process have access to the task pool of that process. They extract tasks from the pool for execution and insert tasks into the task pool if the currently executed task creates new child tasks. The cooperation of tasks possibly executed by different threads is realized via the common shared address space of the process where the data for the program are stored. To realize the correct program behavior the execution of *lock* and *unlock* operations is needed to guarantee a conflict-free access to memory locations.

The entire task program can be executed by a fixed number of threads, also if the number of tasks is varying during program execution. This minimizes the overhead for thread creation. Moreover it has the advantage that a varying number of tasks is mapped to a fixed number of threads yielding a dynamic load balance for the execution.

There are several possibilities for the internal organization of task pools and the storage of tasks. Often the tasks are kept in queues and task pools differ in the number of queues and the access strategy to queues. Concerning the number of queues the main cases are:

Central task pool: Only one task queue exists and all threads of the process access this queue to remove or insert tasks. To avoid conflicts each access has to be protected by a lock operation. Threads remove tasks from the queue for execution when they are ready with previous work and so the central queue offers good dynamic load balance. But frequent accesses to the queue, e.g. for many small tasks, may lead to sequentialization due to the lock protection.

Decentralized task pool: (often also called distributed task pool) Each thread has its own queue from which only this thread can remove and insert tasks. The advantage is that accesses to the queues do not have to be protected. On the other hand the static initialization of task queues may lead to imbalances if the initial tasks in the queues create an unequal number of new tasks, which is usually unknown in advance in the case of irregular applications.

Decentralized task pool with task stealing: This task pool variant of a decentralized task pool allows threads to access queues of other threads if its own queue is empty. This is called *task stealing* [16]. Task stealing avoids load imbalances but requires a locking mechanism.

Another implementation decision concerns the order of inserting tasks and extracting tasks. We distinguish the FIFO (first-in first-out) and LIFO (last-in first-out) access strategies for queues.

2.2 Specific task pool implementations

For shared memory nodes we have implemented the following set of task pools:

- **tp_fifocen** and **tp_lifocen** are central task pools with FIFO/LIFO access strategy. The central queue is protected by a lock mechanism.
- **tp_fifo** and **tp_lifo** are decentralized task pools with FIFO/LIFO access strategy. No lock mechanism is necessary.
- **tp_fifost** and **tp_lifost** are decentralized task pools with FIFO/LIFO access strategy and task stealing. The stealing mechanism is very simple: When the private queue is empty, a thread investigates all other queues and steals tasks (if possible).
- **tp_fifost2** and **tp_lifost2** are decentralized task pools with FIFO/LIFO access strategy and task stealing. A task can only be stolen if more than a predefined number of tasks is available in the queue. In contrast to the stealing mechanism above this avoids the stealing from almost empty queues which would force the owner thread of that queue to steal a task too.

2.3 User interface for task pools

For the programming with a task pool the user API provides the following set of access functions:

(1) **void** tp_init (**unsigned** number_of_threads) ;
allocates and initializes the task pool. The parameter number_of_threads denotes the total number of threads.

(2) **void** tp_reinit ( ) ;
prepares the task pool for re-use.

(3) **void** tp_destroy ( ) ;
deallocates the task pool structure and destroys the task pool threads.

(4) **void** tp_initial_put (**void** (* taskroutine)(), arg_type *arguments) ;
inserts an initial task. The parameter taskroutine is a pointer to the function representing the task. The arguments for that function are given by the pointer arguments.

(5) **void** tp_put (**void** (* taskroutine)(), arg_type *arguments, **unsigned** thread_id) ;
inserts dynamically created tasks into the pool. The parameter thread_id is the identifier of the thread inserting the task. The other parameters are identical to the parameters of tp_initial_put.

(6) **void** tp_get (**unsigned** thread_id) ;
extracts tasks for processing. The parameter thread_id denotes the identifier of the thread removing a task.

The application programmer formulates functions or procedures as tasks and guides their creation explicitly by using the interface above. The strategy of the chosen task pool variant is known, but the actual mechanism is hidden to the user so that the programmer can concentrate on the task structure of the program. The task structure, e.g. the granularity of tasks or the dependencies between tasks, might influence the efficiency for some task pool versions.

3 Task pool teams

Several task pools are combined by mutual explicit message passing to build task pool teams for clusters of PCs or SMPs. Mutual communication between cluster nodes is realized with MPI. In the first subsections the compatibility of both models and the communication behavior of irregular applications are considered. Then we present our communication scheme for task pool teams.

3.1 Compatibility with message passing

Although the Message Passing Interface was designed thread-safe which guarantees that the threads of a process can call MPI operations simultaneously without mutual interference or influence, most implementations of the standard are not thread-safe. MPI-2, an extension of the MPI standard, provides functions to support multithreaded programs. However, current implementations often support only the single-threaded mode. For that reason we use MPI and protect each operation by a lock. This ensures that MPI calls are only made by one thread at a time, but the combination of blocking MPI operations and access protection can lead to deadlock. Therefore nonblocking communication operations are used when necessary.

3.2 Irregular communication behavior

Depending on the specific application there are the following communication situations:

- administrational communication emerging from the need to balance load and synchronize cluster nodes and
- application specific communication to exchange input data, (intermediate) results and other data structures for calculation.

Due to the behavior of irregular algorithms the actual communication is hardly predictable in most cases. Moreover, in a task-pool-based implementation the threads of different cluster nodes work asynchronously on different parts of the code so that the counterpart of communication operations, which is needed for the communication in the SPMD style of the MPI programming, might be missing when calling a communication operation. In order to solve those problems we introduce a separate *communication thread* for each task pool.

3.3 Communication scheme

To integrate a communication thread the basic programming model of task pools for shared memory nodes is modified. The modified task pool of each node consists of a fixed number of $n + 1$ threads: n worker threads, responsible for processing tasks, and a separate communication thread. Figure 1 illustrates a task pool team of two decentralized task pools with n worker threads $WT_1, \dots, WT_n$, their corresponding task queues, and one separate communication thread. We have implemented and tested several communication schemes. Figure 1 shows the most efficient scheme which performs the following steps:

(1) If a worker thread of Node 0 needs data situated in the address space of Node 1, it sends a request directly to that node and waits passively. The communication thread of Node 1 receives the message.

(2) According to the message type the communication thread of Node 1 reacts and sends data back.

(3) The communication thread of Node 0 receives the message and signals the waiting worker thread that non-local data are available.

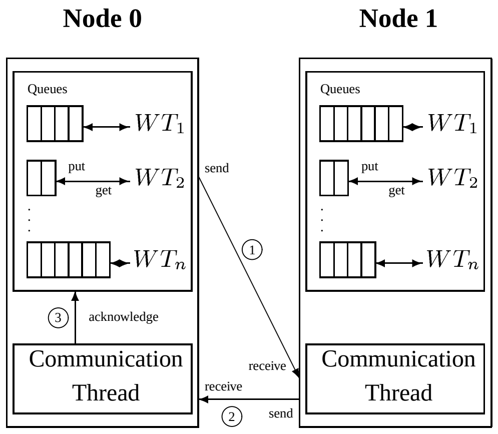

Figure 1. Illustration of the communication scheme for a task pool team with one pool per node. Each node employs n worker threads (WT). The communication protocol with the steps (1)-(3) is described in the text.

The communication thread of Node 0 needs a unique identification of the requesting worker thread in order to awake this thread and to assign the requested data. For that reason we label the entire communication process with an identifier corresponding to the requesting worker thread.

3.4 User interface for task pool teams

Programs for task pool teams are written in an SPMD style and each cluster node runs the same SPMD program. The program for a node has a task-oriented structure and uses a task pool for processing the tasks as described in Section 2. Additionally, the program code contains MPI communication operations and function calls of the Pthread library according to the described communication protocol. Worker threads can initiate the request of remote data by calling an MPI send operation and function calls of the Pthread library for passive waiting. All other send and receive operations of the communication protocol are performed by the communication thread and can be used within a user-programmed function which has a predetermined structure to guarantee the correct functionality.

4 The hierarchical radiosity algorithm

The hierarchical radiosity algorithm (HRA) represents a complex irregular algorithm requiring dynamic load balancing for achieving an efficient parallelization and thus is suitable for testing task pool teams. The next subsections give a short introduction to the HRA and provide information about our application-specific adaptations to reduce communication.

4.1 The hierarchical radiosity algorithm

The basic radiosity algorithm is an observer-independent global illumination algorithm from computer graphics to simulate diffuse light in three-dimensional scenes. The algorithm uses a geometric description (input polygons) of the scene with values of light emission and reflection coefficients. The input polygons are divided into smaller elements for which radiosity values (radiosity = radiation per time and surface unit) are computed. The radiation power of the overall system is modeled by the equation:

$$\underbrace{B_i A_i}_{1} = \underbrace{E_i A_i}_{2} + \underbrace{\rho_i \underbrace{\sum_{j=1}^{n} \underbrace{F_{ji}}_{3a} \underbrace{B_j A_j}_{3b}}_{3}}_{4}, \qquad i = 1, ..., n, \quad (1)$$

1	radiation power of surface element i
2	emission of surface element i
3	incident radiation power from other elements
3a	form factor
3b	radiation power of surface element j
4	diffuse reflection of energy incident from other elements

where B_i is the radiosity value, A_i is the surface area, and E_i is the emission per time and surface unit of element

i. The parameter n is the number of surface elements and determines the refinement level of the division of surface polygons. A large n causes a high quality of the scene and increases the computational effort considerably. The form factor F_{ji} describes the portion of light energy incident on an element from another surface element. The computation of form factors is the most expensive part of the algorithm since it involves visibility tests and the computation of double integrals.

In order to decrease the costs the number of form factors is reduced with a hierarchical approach. The computation of form factors is based on the basic law for the transmission of radiation which means the energy exchanged between elements is decreasing quadratically with the distance of these elements. This fact makes it possible to perform less exact computations for remote surface elements while getting good realistic results. The hierarchical approach results in an uneven division of input polygons into smaller elements. After finishing the algorithm the entire scene is represented by a set of quadtrees with one tree for each input polygon. The leaf elements of all trees represent the surfaces to display, but all levels of the quadtrees are required for computation.

The subdivision of a surface element depends on its size and on the portion of energy incident from other elements. The refinement process stops if a minimal size is reached or the form factors for two interacting elements are small enough to create a realistic scene. The interaction of distant elements takes place on higher levels of the tree. Neighboring elements interact at leaf level. To store the diverse interaction partners each element represented by a quadtree node owns an interaction list which contains pointers to those elements providing portions of light energy. The interaction lists are changed dynamically due to the dynamic refinement of elements depending on the specific geometry and energy situation. The HRA computes the radiosity values by an iterative method for solving Equation (1) with top-down and bottom-up passes over the quadtrees accessing data according to the interaction lists.

The SPLASH2 benchmark suite contains an implementation of the hierarchical radiosity algorithm with task pools for shared memory. We started with the pure thread-based program version modified by [13] and further modified within the scope of [11] for Pthreads. Further adaptations were necessary for the use of our specific task pools and for the integration of the entire communication structure of task pool teams.

4.2 Application-specific optimizations

In order to make the algorithm more efficient several optimizations and techniques have been examined:

Dummy data structures for remote data: In the distributed algorithm the elements and the corresponding quadtrees are distributed over the different address spaces of the cluster nodes. Accordingly, the information in the interaction lists might point to remote data. Thus, the computation of radiosity values might cause irregular communication since radiosity values of other elements stored in non-local address spaces have to be made available by message passing. Because non-local data access with message passing takes usually much more time than local memory access and in most cases the same element is needed again in subsequent calculations, dummies are created to store non-local information. For that reason a suitable data structure was developed that allows easy and fast access by any thread. The correctness of data is guaranteed by refreshing data in each step of the iterative solution method. Experiments have shown that this *software cache* approach leads to efficiency gains.

Initial distribution of data: If two input polygons of the HRA are mutually invisible, there is no exchange of energy between them and therefore an assignment to different nodes does not cause communication. We have investigated the runtime of the HRA with several simple distribution strategies which actually lead to lower communication. Unfortunately, this reduction of communication was associated with a heavy imbalance of computational work. The best runtime results could be achieved with a regular cyclic assignment of initial polygons to the physical processors, although the number of messages was not minimized.

Detecting redundant communication: Each worker thread computes radiosity values for elements using data from interacting local and non-local elements. If two threads of the same process need values of the same non-local element at the same time and this element is not yet present in local memory as dummy, redundant communication occurs. We have implemented a mechanism which initiates a communication of only one thread and blocks any other thread requiring the data until the specific data are available. The underlying structure is a small array for each task pool containing all currently requested elements. Before initiating a communication each thread checks that array for the needed element. If an entry for that element already exists, the thread blocks otherwise it creates an entry and starts the communication process. The thread actually sending the request awakes all waiting threads after the requested data are available and deletes the element from the array. There are at most as many entries as worker threads per task pool and so the search for elements in those arrays is cheap, especially when compared with communication.

Combining requests: In order to reduce communication, requests can be collected and send as a single message. The HRA offers two obvious situations for the combination of requests: a.) The preliminary radiosity values for elements computed at one node within one iteration step are immediately accessible for local threads. New values of non-local elements are exchanged between nodes at the end of an iteration step rather than immediately. This algorithmic change results in slight changes of the convergence rate. b.) During the computation of the radiosity values of an element each interacting element is investigated for locality. If there are non-local interacting elements, they are accessed separately. The access of all interacting non-local elements can be combined because they are already known at the beginning of an iteration step.

Although these improvements are application-specific, the basic ideas can be applied to other irregular algorithms.

5 Experimental results

This section presents some of our experimental results with the hierarchical radiosity algorithm. As example scenes we use ”largeroom” (532 initial polygons), ”hall” (1157 initial polygons), and ”xlroom” (2979 initial polygons). The models ”hall” and ”xlroom” were generated by [13]. ”Largeroom” belongs to the SPLASH2 benchmark suite [19].

We have investigated task pool teams on two architectures: the Chemnitzer Linux Cluster (*CLiC*), a Beowulf cluster consisting of 528 Intel Pentium III processors with 800 MHz. CLiC has a Fast Ethernet network. *SB1000* is a small SMP cluster of four Sun Blades 1000 with two 750 MHz UltraSPARC3 processors. SB1000 uses SCI.

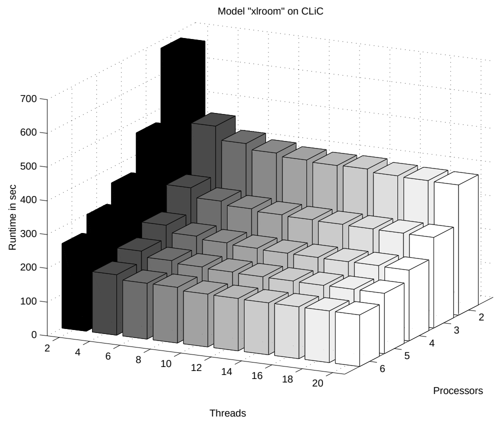

Figure 2. Runtimes for model "xlroom" on CLiC

Figure 2 gives runtimes depending on the number of threads and the number of processors of the HRA using the scene ”xlroom”. The figure shows that there are dependences of the runtime on the number of processors as well as on the number of threads. The overall runtime behavior is similar for other task pool teams, scenes and platforms. Therefore we present all runtime and speedup measurements with regard to threads and processors. The speedup values have been computed using the implementation without communication thread and with only one worker thread.

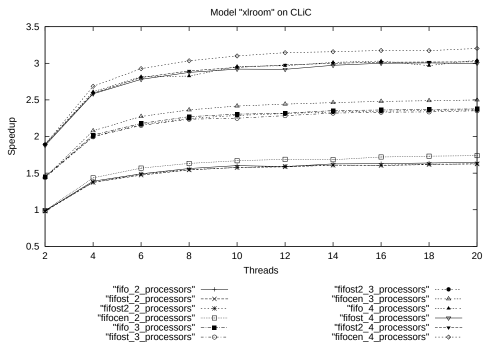

Figure 3. Speedups for model "xlroom" and task pools with FIFO access on CLiC

Although CLiC has only one CPU per cluster node the usage of only two threads per node (one worker thread and one communication thread) is not reasonable since an increasing number of threads leads to better speedups as Figure 3 shows. This fact is caused by the competition between threads for processing time. For a large number of threads per process the allocated time for the communication thread (which mostly performs unsuccessful checks for incoming messages) is reduced. Also, the wasted time due to blocking worker threads is minimized. Our experiments have shown that the effect of an increasing efficiency for an increasing number of threads reaches a saturation point. The speedup results for the smallest model ”largeroom” are less regular than for the models ”hall” and ”xlroom”. This is caused by the small number of initial polygons for each cluster node compared with the irregular communication requirements.

Because all threads of a cluster node have to compete for two processors the dependence between speedup and number of threads on SB1000 is similar to the results of CLiC as Figure 4 shows.

The task pool teams computing the model ”largeroom” achieve very different results when using different task pool variants while the runtimes for computing the models ”hall” and ”xlroom” differ only slightly as Figure 5 shows. In most cases computing the model ”largeroom”, the central task pool and the distributed task pool with the *st2* stealing mechanism have better speedups than the other distributed task pools. This fact results from a better load balance between the threads of one cluster node. The distributed task pools without task stealing do not balance load very well and the stealing mechanism *st* wastes too much time when trying to steal from almost empty queues. For the larger

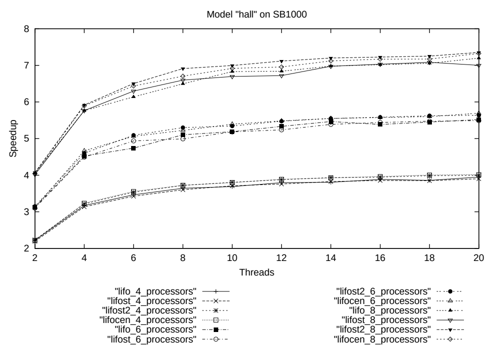

Figure 4. Speedups for model "hall" and task pools with LIFO access on SB1000

models ”hall” and ”xlroom” the runtimes for different task pools differ only slightly. It seems that load imbalances within one cluster node hardly affect the total runtime because of the long over-all runtime compared with the idle time caused by load imbalances.

In the following the speedups with a fixed number of 20 threads per task pool on CLiC and SB1000 are presented. On both platforms the models ”largeroom” and ”hall” achieve the best speedups with pools with the task stealing mechanism *st2*. The central task pool with FIFO access principle is best-suited for the largest model ”xlroom”. The numbers in brackets are the number of physical processors.

CLiC:	larger.	1.7 (2), 2.5 (3), 2.7 (4), 3.4 (5), 3.7 (6)
	hall	1.7 (2), 2.5 (3), 3.2 (4), 3.7 (5), 4.4 (6)
	xlroom	1.7 (2), 2.5 (3), 3.2 (4), 3.7 (5), 4.4 (6)
SB1000:	largeroom	3.7 (4), 5.4 (6), 5.9 (8)
	hall	4.0 (4), 5.8 (6), 7.4 (8)
	xlroom	3.8 (4), 5.4 (6), 7.1 (8)

Due to the large number of tasks compared with the communication the speedup values of ”hall” and ”xlroom” are better than the values for the small model ”largeroom”.

6 Related work

There are several methods for implementing irregular algorithms on distributed memory. Some systems (Titanium [20], TreadMarks [1]) provide implicit support for irregular applications by a distributed shared memory layer.

The *inspector/executor* model (PARTI [17], CHAOS [15], PETSc [3]) performs optimizations to reduce the runtime of nested loops over distributed arrays. The necessary array elements are provided in a preprocessing step by predefined communication operations. This approach requires compiler support and does not realize dynamic load balance.

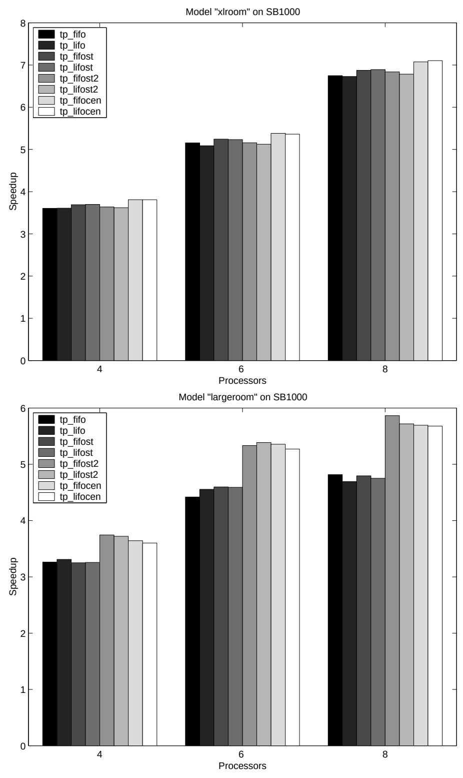

Figure 5. Speedups with 20 threads for models "xlroom" and "largeroom" on SB1000

Especially for problems with irregular grid-based data structures load balance can be achieved by *partitioning* into blocks. The aim is to obtain blocks of approximately identical size with minimal interdependence. This problem known as the graph partitioning problem is NP-complete. MeTis [10] and CHACO [8] realize several partitioning algorithms. PLUM [12] is a load balancing framework on the base of such partitioners. The usage of partitioning algorithms is not useful if the time for the calculation of partitions and for repartitioning exceeds the time savings gained. A dynamic load balancing algorithm for unstructured grids is presented in [5]. [9] gives an overview of further work done in this field.

There are several approaches for *hybrid programming* of SMP clusters. SIMPLE [2] provides user level primitives on the base of shared memory programming and message passing and can only be used for applications which allow a strict separation of computation and communication phases. NICAM [18] is a user level communication layer for SMP clusters which supports overlapping of communication and computation in iterative data parallel applications. An application specific example which uses threads and remote memory operations is presented in [14]. This approach introduces the parallelization of sparse Cholesky factorization using threads for the parallel computation of blocks of the factor matrix and remote memory operations for synchronization of remote data accesses.

There are some packages providing threads on distributed memory. Nexus [6] is a runtime environment for irregular, heterogeneous, and task-parallel applications. The focus is more on the realization of the combination on lower levels. Chant [7] presents threads capable of direct communication on distributed memory. This library uses lightweight thread libraries and communication libraries available at the system used. In contrast, our approach is entirely situated within the application programmers level in order to provide a systematic programming approach to the programmer without hiding important details and implicit load balance.

7 Conclusion

The parallel implementation of irregular algorithms on clusters of PCs or SMPs requires special dynamic load balancing and organization of irregular accesses. For this purpose we have presented task pool teams, a generalized task pool approach. Task pool teams combine multithreaded programming with explicit communication on the application programmers level. The efficiency results are good and depend on the platform and the specific input data. We have chosen the hierarchical radiosity algorithm as test program since this application program is very complex and combines many characteristics of irregular algorithms. The experiments on several platforms have shown that the approach is suitable for efficient parallelization.

References

[1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, 1996.

[2] D. A. Bader and J. JáJá. SIMPLE: A Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors (SMPs). *Journal of Parallel and Distributed Computing*, 58(1):92–108, 1999.

[3] S. Balay, W. Gropp, L. McInnes, and B. Smith. Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries. In E. Arge, A. M. Bruaset, and e. H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[4] D. R. Butenhof. *Programming with POSIX Threads.* Addison-Wesley, 1997.

[5] S. K. Das, D. J. Harvey, and R. Biswas. Parallel Processing of Adaptive Meshes with Load Balancing. *IEEE Trans. on Parallel and Distributed Computing*, 12(12):1269–1280, 2001.

[6] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, 1996.

[7] M. Haines, P. Mehrotra, and D. Cronk. *Chant: Lightweight Threads in a Distributed Memory Environment.* Technical report, ICASE, 1995.

[8] B. Hendrickson and R. Leland. *The Chaco User's Guide, Version 2.0.* Sandia National Laboratories, Technical Report SAND94-2692, 1994.

[9] P. K. Jimack. An overview of parallel dynamic load-balancing for parallel adaptive computational mechanics codes. In B. H. V. Topping, editor, *Parallel and Distributed Processing for Computational Mechanics: Systems and Tools*, pages 350–369. Saxe-Coburg Publications, 1999.

[10] G. Karypis and V. Kumar. *MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0.* Departement of Computer Science, University of Minnesota, Technical Report, 1995.

[11] M. Korch and T. Rauber. Evaluation of Task Pools for the Implementation of Parallel Irregular Algorithms. *Proc. of ICPP'02 Workshops, CRTPC 2002, Vancouver, Canada*, pages 597–604, 2002.

[12] L. Oliker and R. Biswas. PLUM: Parallel load balancing for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 52(2):150–177, 1998.

[13] A. Podehl, T. Rauber, and G. Rünger. A Shared-Memory Implementation of the Hierarchical Radiosity Method. *Theoretical Computer Science*, 196(1-2):215–240, 1998.

[14] S. Satoh, K. Kusano, Y. Tanaka, M. Matsuda, and M. Sato. Parallelization of Sparse Cholesky Factorization on an SMP Cluster. In *Proc. HPCN Europe 1999, LNCS 1593*, pages 211–220, 1999.

[15] S. D. Sharma, R. Ponnusamy, B. Moon, Y. Hwang, R. Das, and J. H. Saltz. Run-time and Compile-time Support for Adaptive Irregular Problems. In *Proc. of Supercomputing '94*, pages 97–106, 1994.

[16] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy. Load Balancing and Data Locality in Adaptive Hierarchical N-body Methods: Barnes-Hut, Fast Multipole, and Radiosity. *Journal of Parallel and Distributed Computing*, 27(2):118–141, 1995.

[17] A. Sussman, J. Saltz, R. Das, S. Gupta, D. Mavriplis, R. Ponnusamy, and K. Crowley. PARTI Primitives for Unstructured and Block Structured Problems. *Computing Systems in Engineering*, 3(1-4):73–86, 1992.

[18] Y. Tanaka. Performance Improvement by Overlapping Computation and Communication on SMP Clusters. *In Proc. of the 1998 Int. Conf. on Parallel and Distributed Processing Techniques and Applications*, 1:275–282, 1998.

[19] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd Annual Int. Symposium on Computer Architecture*, pages 24–36, 1995.

[20] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A High-Performance Java Dialect. In *Proc. ACM 1998 Workshop on Java for High-Performance Network Computing*, 1998.