A Communication API for Implementing Irregular Algorithms on SMP Clusters

Judith Hippold* and Gudula Rünger

Chemnitz University of Technology, Department of Computer Science
09107 Chemnitz, Germany
{juh,ruenger}@informatik.tu-chemnitz.de

Abstract. Clusters of SMPs (symmetric multiprocessors) gain more and more importance in high performance computing and are often used with a hybrid programming model based on message passing and multithreading. For irregular application programs with dynamically changing computation and data access behavior a flexible programming model is needed to achieve load balance and efficiency. We have proposed *task pool teams* as a new hybrid concept to realize irregular algorithms on clusters of SMPs efficiently. Task pool teams offer implicit load balance on single cluster nodes together with multithreaded, MPI-based communication for the exchange of remote data. This paper introduces the programming model and library support with an easy to use application programmer interface for task pool teams.

1 Introduction

Irregular algorithms are characterized by an unknown access behavior to data structures, varying computational effort, or irregular dependencies between computations resulting in an input dependent program behavior. The characteristics of irregular algorithms make an efficient parallel implementation 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. A task pool is a global data structure to store and manage the tasks created for one specific program combined with a fixed number of threads processing these tasks. Each thread can extract tasks for execution and can insert dynamically created tasks. Task pools are classified according to the internal organization or access strategies. Central task pools offer good load balance because each thread can get a new task from the central queue when ready with previous work. Decentralized pools usually have as many queues as threads. The advantage is that there are no access conflicts between several threads, however, dynamic load balance can only be achieved with task stealing. That means, if the private task queue of a thread is empty, it tries to steal tasks from other queues which requires a lock mechanism. Detailed investigations of different task pool versions are given in [1]. Task pool implementations for dynamic load balance have been presented in e.g. [2][3].

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

In [4] we have introduced task pool teams, a generalization of the task pool concept, for realizing irregular algorithms on SMP clusters. Task pool teams combine task pools on single nodes implemented with Pthreads for load balance with explicit message passing using MPI for irregularly occurring, remote data accesses. The resulting implementation of an irregular algorithm with task pool teams is entirely realized on the application programmers level so that the characteristics of the specific algorithm can be exploited.

This paper presents the hybrid programming model and the application programmer interface. The API clarifies the concept and improves the structure of the resulting program while no additional overhead compared to the hand-coded version is introduced. Moreover, we present several communication protocols and corresponding functions which can be used to realize different communication characteristics. This allows an easy adaptation to the communication requirements of different irregular algorithms.

The paper is structured as follows: Section 2 introduces the hybrid programming model and the communication scheme for task pool teams. Sections 3 and 4 describe the API for task pools and communication. Experimental results are presented in Section 5. Section 6 discusses related work and Section 7 concludes.

2 Hybrid programming model

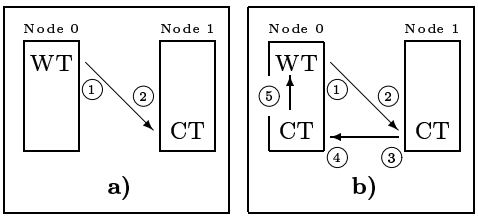
We assume the following basic programming model: Programs are written in an SPMD style, and each cluster node runs the same program containing MPI and Pthread operations as well as calls to the task pool team interface. Within the program there are code sections with task-oriented structure processed in parallel by a task pool. There exists one pool for each cluster node. Mutual communication between nodes is realized with message passing. The task pools of all cluster nodes form a task pool team.

At the beginning and at the end of a program run only the main thread is active. It can use MPI operations arbitrarily, for example to distribute data or to collect data from other processors. The program switches to multithreaded mode when the main thread starts the processing of tasks and switches back to single-threaded when it leaves the task pool. Task pools are realized with a fixed number of *worker threads* (WT) and are re-usable in different multithreaded sections.

Irregular application programs may have irregular communication requirements with communication partners working asynchronously on different parts of the code. Thus the counterpart of the communication operation to be issued by the remote communication partner might be missing when using MPI communication operations. To ensure correct communication, we introduce a communication scheme with a separate *communication thread* (CT) for each task pool and distinguish between two cases of communication (see Figure 1).

Simple communication can be used e. g. in parallel backtracking algorithms using treelike solution spaces where locally calculated, intermediate results are exchanged to restrict the search space. Figure 1 a) illustrates this communication process consisting of the following steps: *(1) A worker thread of Node 0 sends calculated data to Node 1. (2) The remote communication thread receives and processes the data.*

a)
1. send_data
2. receive_data



b)
1. initial_send
2. initial_receive
3. final_send
4. final_receive
5. awake WT

Fig. 1. Simplified illustration of a) simple and b) complex communication between two task pools only showing one worker thread and the communication threads.

Complex communication occurs if intermediate results or data structures of one process are needed by another process to continue the calculations, as in the hierarchical radiosity algorithm. The following steps have to be performed as illustrated in Figure 1 b): *(1) A worker thread of Node 0 needs data situated in the address space of Node 1. Thus it sends a request and waits passively. (2) The communication thread of Node 1 receives the request, reacts according to the message tag, and (3) sends the data back. (4) The CT of Node 0 receives the message and (5) signals the waiting worker thread that non-local data are available.*

Within every communication situation there are actually two distinct kinds of thread interactions to manage: the local cooperation of worker thread and communication thread and the handling of remote communication needs. We have decoupled the local cooperation and the remote communication in order to avoid conflicting requirements for the CT which might occur when initializing data requests interferes with the receipt of asynchronously arriving messages. The decoupling realized in the complex communication scheme makes this scheme suitable for the unknown data requests within irregular algorithms.

3 User interface for task pools

The interface for task pools was modified compared to the former version (see [4]) concerning an implicit re-initialization of the task pools and an explicit selection of the task pool run mode to provide a more general and optimized implementation. If there is no exchange of messages between different MPI processes, the pools should run in the WITHOUT_CT mode (see function (4)) because this results in a smaller execution time.

(1) **void** tp_init (**unsigned** number_of_threads, **void** (* ct_routine)()) ;
allocates and initializes the task pool. number_of_threads denotes the total number of threads including the main thread and the communication thread. ct_routine points to the function performed by the communication thread (see Section 4.3).

(2) **void** tp_destroy ( ) ;
deallocates the task pool structure and destroys all threads except the main thread.

(3) **void** tp_initial_put (**void** (* task_routine)(), arg_type *arguments) ;
inserts an initial task into the task pool. task_routine is a pointer to the function representing the task. The arguments for that function are given by arguments.

(4) **void** tp_run (**unsigned** type) ;
starts the processing of tasks. Depending on the type (WITH_CT or WITHOUT_CT) the task pool runs with or without a communication thread.

(5) **void** tp_put (**void** (* task_routine)(), 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.

Functions (1)-(4) are performed by the main thread. Function (5) can be used by each thread working on the pool. The main thread becomes a worker thread after calling tp_run. When the task pool is empty, the main thread prepares the pool for re-use and returns. There is no need to destroy the pool and to create a new one when it might be used again which saves thread creation time. Each worker thread is identified by its thread_id. This identifier is assigned after thread creation and different from the identifier provided for Pthreads.

4 User interface for communication in task pool teams

Task pool teams are created by combining task pools (see Section 3) with the communication schemes from Section 2. We present an interface for simple and complex communication and further distinguish between the cases of a) the size of incoming messages is not known and b) the size of incoming messages is known in advance. The provided functions in Subsections 4.1 and 4.2 encapsulate MPI and Pthread operations. The use of those functions is illustrated in 4.3.

4.1 Complex communication

(1) **void** *initial_send(**void** *buffer, **int** count, **MPI_Datatype** type, **int** dest, **int** tag, **unsigned** thread_id);
sends a data request to processor dest. The pointer buffer denotes the data of length count and data type type necessary to identify the requested data. tag denotes the message in order to initiate a specific action on the destination process. The function returns a pointer to the requested data.

(2) **int** get_size(**unsigned** thread_id) ;
determines the size of the data requested by initial_send. The parameter thread_id identifies the requesting thread. (only necessary for case a))

(3) a) **void** *initial_receive(**int** *count, **MPI_Datatype** type) ;
receives the data request and returns a pointer to the received data. After finishing initial_receive, count contains the length of the received data of type type.
b) **void** initial_receive_buf(**void** *buffer, **int** count, **MPI_Datatype** type) ;
receives the data request. The user provides the buffer buffer of length count.

(4) **void** final_send(**void** *buffer, **int** count, **MPI_Datatype** type) ;
sends the requested data in buffer of length count and data type type back to the requesting process.

(5) a) **void** final_receive(**MPI_Datatype** type) ;
receives the requested data of data type type and awakes the waiting worker thread.
b) **void** final_receive_buf(**void** *buffer, **int** count, **MPI_Datatype** type) ;
receives the requested data of data type type and length count in the user provided buffer buffer and awakes the waiting worker thread.

Function (1) initiates a data request by a worker thread. The calling WT is blocked until the data are available. (2) provides the size of the received data to this worker thread. Functions (3)-(5) are performed by the communication threads of the communication partners. The finally received data are available in the shared memory of the node.

4.2 Simple communication

(1) **void** send_data(**void** *buffer, **int** count, **MPI_Datatype** type, **int** dest, **int** tag) ;
sends the data in buffer of length count and data type type to process dest. tag identifies the message.

(2) a) **void** *receive_data(**int** *source, **int** *count, **MPI_Datatype** type) ;
receives the data of data type type. After finishing this function, the sender and the size are stored in source and count, and a pointer to the received data is returned.
b) **void** receive_data_buf(**void** *buffer, **int** *source, **int** count, **MPI_Datatype** type) ;
receives the data of data type type and length count in the user provided buffer buffer. After finishing this function, the sender is stored in source.

The worker thread performs function (1) and the communication thread performs function (2). There is no need to block the worker thread because the communication process is finished. The CT performs algorithm specific computations on the received data.

4.3 Communication thread

The application programmer has to code the function performed by the communication thread according to algorithmic needs and gives a pointer to it as a parameter of tp_init. The following pseudo-code illustrates the core structure of this function with examples for complex and simple communication. /*... ...*/ labels the location for algorithm specific code.

```
while(1) {
   if(message_check(&tag)) {
      /* COMPLEX COMMUNICATION */
      if(tag == 1) {
         initial_receive();
         /*... get data ...*/
         final_send();
      }
      else if(tag == 2) {
         final_receive();
      }
                                    /* SIMPLE COMMUNICATION */
                                    else if(tag == 3) {
                                       receive_data();
                                       /*... process data ...*/
                                    }
                                    else if ...
                                 } /* end if(message_check()) */
                              } /* end while(1) */
```

We provide the function **int** message_check(**int** *tag) to check for incoming messages. If there is a message available, it returns TRUE and tag contains the tag of the message. According to this message tag the communication thread selects the specific action to perform. The application programmer chooses an arbitrary, odd, and unique integer as tag for a specific type of request. If a message with an odd tag arrives, the communication thread automatically increments the tag

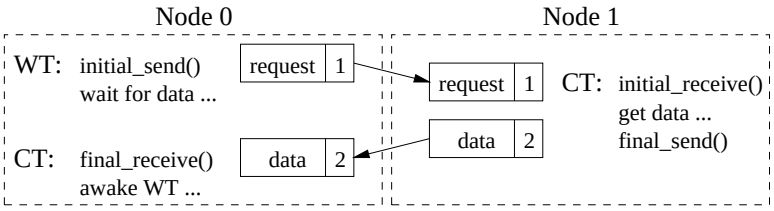


Fig. 2. Communication protocol for complex communication using the tags 1 and 2

and uses it for the response message. Thus the pair $(2i+1, 2i+2), i \in I\!N$, labels a complex communication process of a specific type. The simple communication process needs only one tag because there are no final_send and final_receive operations necessary. The pseudo-code shows a complex communication process with the tags 1 and 2. Figure 2 illustrates the corresponding communication protocol.

4.4 Synchronization hierarchy

Although programs with task pool teams have SPMD style, the task-oriented structure and the irregularity of the algorithm lead to different, asynchronous program runs on different cluster nodes. Implicit synchronization is only necessary for switching from multi- to single-threaded mode to guarantee that the remote communication partners are active and can handle data requests. Task pool teams ensure this by a synchronization hierarchy (see Figure 3) completely hidden to the user.

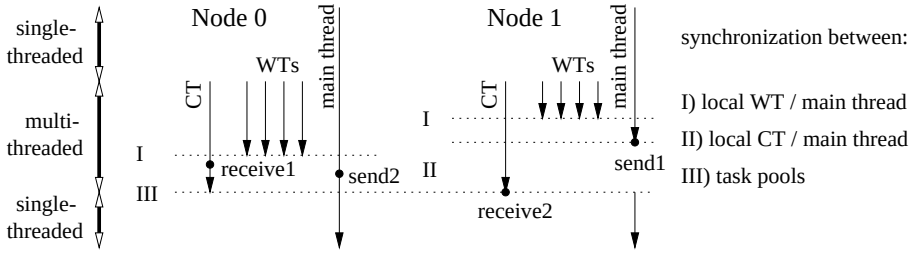


Fig. 3. Synchronization hierarchy illustrated with two cluster nodes. The main thread of Node 1 has to wait till the local CT receives a synchronization message of Node 0. The main thread of Node 0 continues with computation after sending the synchronization message because the message of Node 1 was already received.

The synchronization of task pools (Figure 3, III) and local main thread and communication thread (Figure 3, II) is closely connected. Task pools are synchronized by sending synchronization messages after all local tasks are processed. This is done by the main thread. Each communication thread gathers these messages and awakes the waiting main thread when an acknowledge of every task pool has arrived. The following pseudo-code illustrates that mechanism:

```
/* MAIN THREAD LOOP */
CT_run = TRUE;
signal_communication_thread();
/* ... process tasks ... */
send_synchronization_message();
while(CT_run) wait();
```

```
/* COMMUNICATION THREAD LOOP */
while(!CT_run) wait();
/* handle communication till synchro-
   nization message of all pools */
CT_run = FALSE;
signal_main_thread();
```

To ensure correct program behavior, the main thread is not allowed to leave the task pool till every other local worker thread is ready and waits passively for re-activation (Figure 3, I). Otherwise, it would be possible that the communication thread is stopped by the main thread while there are worker threads still waiting for non-local data. The pseudo-code realizes such a barrier:

```
waiting_threads += 1;
if(waiting_threads < number_of_WT) wait();
else { if(main_thread) return;
       else {   signal_main_thread();
                wait();
}      }
```

Threads entering the barrier wait passively except the last thread which awakes the main thread and blocks, unless it is the main thread itself. Then it returns.

5 Experimental results

We have implemented the hierarchical radiosity algorithm using our application programmer interface for task pool teams and compare this realization with a former hand-coded version of [4], where more detailed investigations and evaluations about the basic task pools can be found. We use three example scenes: "largeroom" (532 initial polygons) of the SPLASH2 benchmark suite [5], "hall" (1157 initial polygons) and "xlroom" (2979 initial polygons) from [2]. We have investigated our API on a small SMP cluster of 4 SunBlade 1000 with two 750 MHz UltraSPARC3 processors and SCI network.

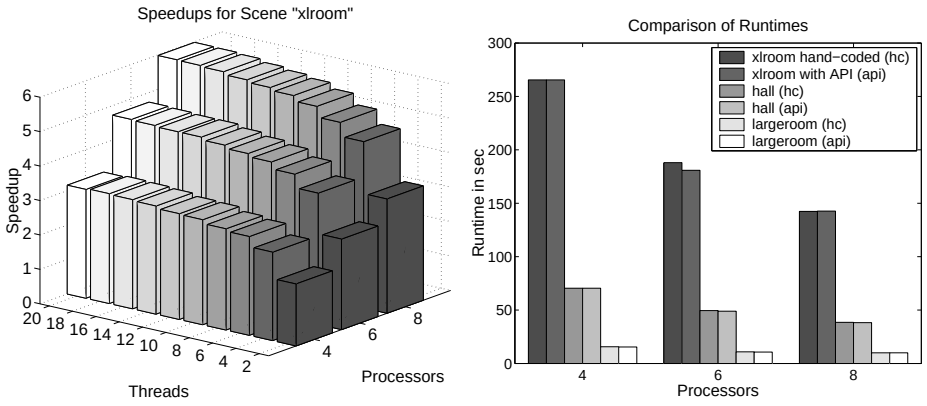


Fig. 4. Left: Speedups for model "xlroom". Right: Speedups of the hand-coded version (*hc*) compared with the radiosity version using the API (*api*).

On the left of Figure 4 the speedups depending on the number of threads and processors are shown for the scene "xlroom" with the best-suited task pool. Each cluster node runs one MPI process. Although there are only two processors per cluster node, an increased number of threads leads to better speedups caused by the reduction of the allocated CPU time for the communication thread and by the overlapping of computation and communication. That means, while a worker thread waits for data, another worker thread might perform calculations. Due to the overhead for threads and the smaller workload per thread with growing numbers of processes the speedups reach a saturation point.

On the right of Figure 4 a comparison between the former hand-coded version and the radiosity version using the API is shown. The runtimes of the best-suited task pools with 20 threads per pool are presented for varying example scenes and numbers of processors. It is shown that there are only slight changes in execution times between both radiosity versions. Thus our interface for communication induces only a marginal overhead.

6 Related work

There are several approaches for hybrid programming of SMP clusters: SIMPLE [6] can be used for applications which allow a strict separation of computation and communication phases. NICAM [7] supports overlapping of communication and computation for data parallel applications. Some packages provide threads on distributed memory. Nexus [8] is a runtime environment for irregular, heterogeneous, and task-parallel applications. Chant [9] presents threads capable of direct communication on distributed memory. In contrast, our approach is entirely situated within the application programmers level in order to provide a systematic programming approach without hiding important details and implicit load balance.

7 Conclusion

We have presented an easy to use interface for task pool teams suitable for the parallelization of irregular algorithms on clusters of SMPs. Our model offers dynamic load balance on individual cluster nodes and multithreaded communication for the task-oriented sections. The efficiency results are good and depend on the specific input data. Task pool teams are also advantageous for PC clusters since communication and computation overlap in task-oriented sections.

References

1. Korch, M., Rauber, T.: Evaluation of Task Pools for the Implementation of Parallel Irregular Algorithms. Proc. of ICPP'02 Workshops, CRTPC 2002, Vancouver, Canada (2002) 597–604
2. Podehl, A., Rauber, T., Rünger, G.: A Shared-Memory Implementation of the Hierarchical Radiosity Method. Theoretical Computer Science **196** (1998) 215–240
3. Singh, J.P., Holt, C., Totsuka, T., Gupta, A., Hennessy, J.: Load Balancing and Data Locality in Adaptive Hierarchical N-body Methods: Barnes-Hut, Fast Multipole, and Radiosity. Journal of Parallel and Distributed Computing **27** (1995) 118–141
4. Hippold, J., Rünger, G.: Task Pool Teams for Implementing Irregular Algorithms on Clusters of SMPs. In Proc. of the 17th IPDPS, Nice, France. (2003)
5. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 Programs: Characterization and Methodological Considerations. In: Proc. of the 22nd Annual Int. Symposium on Computer Architecture. (1995) 24–36
6. Bader, D.A., JáJá, J.: SIMPLE: A Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors (SMPs). Journal of Parallel and Distributed Computing **58** (1999) 92–108
7. Tanaka, Y.: 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** (1998) 275–282
8. Foster, I., Kesselman, C., Tuecke, S.: The Nexus Approach to Integrating Multithreading and Communication. Journal of Parallel and Distributed Computing **37** (1996) 70–82
9. Haines, M., Mehrotra, P., Cronk, D.: Chant: Lightweight Threads in a Distributed Memory Environment. Technical report, ICASE. (1995)