Towards Space-Efficient Dynamic Process Scheduling for Recursive Divide and Conquer Programs in MPICH2

Stéfano D. K. Mór, Nicolas B. Maillard
Federal University of Rio Grande do Sul, Porto Alegre/RS, Brazil,
{sdkmor, nicolas}@inf.ufrgs.br

Abstract

MPICH2's dynamic process creation scheduler produces undesired workload imbalance with recursive D&C algorithms due to its default scheduling algorithm. The memory limitations, at the overloaded nodes, turn critical for the programs to scale well. To overcome this constraint, we have modified MPICH2's dynamic process creation (`MPI_Comm_spawn`) in order to implement a ring-based, distributed, load balance scheduling algorithm that implements work stealing. This is a (stricter) distributed version of previously published work [2]. Experimental results confirm the expected performance: benchmarks achieved near-optimal spatial complexity, allowing MPICH2 to run instances four times larger than usual while keeping execution time nearly the same.

1. Introduction

Recursive divide and conquer (D&C) is one of the most important programming techniques in Computer Science [3], allowing a programmer to write elegant solutions for problems like data sorting or combinatorial optimization. It also leads to a natural introduction of parallelism by dynamic process creation: each recursive function call can be mapped to a process spawned in parallel; a parent process blocks until all its spawned children return their outputs, which will later be combined. For such algorithms, usually with irregular load balancing constraints, one useful idea with MPI is to use `MPI_Comm_spawn` to dynamically create new processes.

MPI-2 has introduced `MPI_Comm_spawn` (and a family of correlated primitives) to allow a given executing process to spawn MPI processes dynamically. It has not imposed, however, any canonical way to schedule these processes across the available nodes, leaving it up to the programmer or to the operating system. The default round-robin strategy, used by some major MPI-2 implementations (*e.g.*, LAM-MPI and MPICH2), is very problematic when one considers parallel recursive D&C; some nodes are unnecessarily overloaded, while others remain idle. The scheduler's inefficient load balancing behavior bounds the size of a given problem's instance by unnecessarily increasing spatial complexity due to overloaded nodes reaching memory limits too soon during the execution. Sec. 2 discusses this topic with more details.

Thus, space-optimal load balancing of dynamic processes is critical when considering parallel execution on a multicomputer with a limited number of resources. It would allow to maximize the simultaneous running processes in MPI-2 D&C programs. This work has modified MPICH2 to do so, because this implementation is the basis for many commercial distributions of the norm (*e.g.*, Intel's and Microsoft's) as well as popular in the academic field.

The remainder of this paper is organized as follows. Section 2 shows, in short, how MPICH2 handles a new MPI process spawn. Section 3 specifies parallel D&C algorithms and their bounds. Section 4 shows the modifications required in order to achieve space-efficient D&C scheduling with MPICH2, while Section 5 shows some results on practical experiments that reinforce previous considerations and statements. Finally, Section 6 brings some final remarks and conclusions.

2. On-line Scheduling of Dynamic Processes in MPICH2

MPD, written in Python, is the process manager daemon in MPICH2. Each active node has a MPD running. It is responsible for initializing the system and to handle the creation of MPI processes, both static and dynamic, and to schedule the dynamic processes.

As a Python class, two components are highlighted in the context of this work.

MPDRing is a component class for ring-based communication. It also provides detection of and union with an already established ring.

MPDMan is a component class that works as a "header" for a running C, C++, or Fortran user MPI process. It is implemented as a separate process.

The default dynamic process creation in this ring topology works basically as follows. Consider a process $proc$ working on a node, and that all MPDs own a queue Q of processes to be spawned:

1. *proc* invokes `MPI_Comm_spawn`, sending, through internal function calls, a corresponding message to its MPDMan that, on its turn, re-pass it to the corresponding MPD.
2. MPD, after receiving the spawn message from a MPDMan, pushes it on top of Q.
3. From time to time, MPD tests Q; if it is empty, MPD proceeds with its normal execution. If it is not, MPD sends Q's top message to the MPD of the next node (at "right") on the ring. Whenever a spawn is being processed, no message is popped from Q. When an "ack" is received, the same process goes on again.
4. When a given MPD receives a spawn message from the "left" node on the ring, it spawns locally a given number of processes, specified by the spawn message, decrements this number from the total number process to be spawned and re-pass the message to its "right" node on the ring. This goes on until the total number of already spawned processes is reached. The last MPD to spawn a process sends an "ack" message to the original spawner (identified in the message).

The above approach leads to imbalance and non-efficient space usage when one considers recursive D&C programming. The next section will show why.

3. Recursive Divide and Conquer Parallel Programming with MPI-2

Recursive D&C parallel function calls (henceforward referred as *spawn*) can be easily coded with the use of `MPI_Comm_spawn`, as described in Sec. 1. When combined with the `MPI_Recv` family of primitives, it provides a robust way to block until all the produced outputs arrive from the children (henceforward referred as *sync*). In other words, one may express parallel recursive D&C in MPI-2 by defining

spawn := `MPI_Comm_spawn` plus data sending
sync := `MPI_Recv` (or similar).

The parallel D&C MPI program, thus, *divides* its input, performs a *spawn*, that recursively runs the program, and a *sync*. After having received all children's results, the *conquer* phase is run to merge them.

A *threshold* is used to impose a minimal granularity to the recursive tasks, as usual in sequential implementations. We assume that the runtime of this minimal, sequential task, is constant (*i.e.* for a given input of size n, even if the number of tasks increases with n, the size of the sequential task will not); and that the time it takes to divide the input among P processors and to merge the output are, both, proportional to P.

The default behavior of MPICH2 (when based on MPD) described in Sec. 2 produces workload imbalance when one considers D&C programming like Fig. 1(a). Fig. 2 shows examples of the process scheduling for typical D&C programs spawning $\delta = 3$ processes recursively. The rounded-corner thin-line squares represent a processor p where each

```
int
fib ( int i )
{
  int a, b;
  MPI::Intercomm comm1, comm2;

  if ( i <= THRESHOLD )
    return seq_fib(i);

  // EVAL BEGIN //
  if ( i == 0  || i == 1 ) return i;
  // EVAL END //

  // DIVIDE BEGIN //
  a = i-1; b = i-2;
  // DIVIDE END //

  // SPAWN BEGIN //
  comm1 = MPI::COMM_WORLD.Spawn
    ("./FibTask", [...]);
  comm1.Send
    (&a, 1, MPI::INT, [...] );
  // SPAWN END //

  /*seq. branch*/
  b = fib(b);

  // SYNC BEGIN //
  comm1.Recv
    (&a, 1, MPI::INT, [...]);
  // SYNC END //

  // CONQUER BEGIN //
  return a+b;
  // CONQUER END //
}
```

(a) Using MPICH2's (MPD) default scheduler.

```
int
fib ( int i )
{
  int a, b;
  MPI::Intercomm comm1, comm2;
  MPI::Request req[2];

  [...]

  // SYNC BEGIN //
  req[0] = comm1.Irecv
    (&a, 1, MPI::INT, [...]);
  MPI_Block_notf();
  while ( ! MPI::Request::Testall(1, req) )
    sched_yield();
  // SYNC END //

  // CONQUER BEGIN //
  return a+b;
  // CONQUER END //
}
```

(b) Using RBWS as the scheduler.

Figure 1. Parallel calculus of the i-th term of Fibonacci's series on a language similar to C++/MPI-2. "[...]" represents non-important parameters in function calls and, on (b), replicated code from (a).

MPI process has already spawned all its dynamic processes. Rounded-corner thick-line squares represent processors where no MPI process has created another MPI process dynamically. Dashed arrows connect parent MPI processes with their sons.

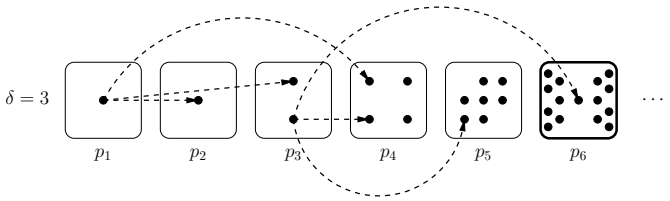

Figure 2. Example of execution with MPICH2's default scheduling algorithm for D&C programs.

If we consider that, on each node, the available memory bounds the number of processes by node to 15 (like on p_6), the computation ends with p_1 having running only one process, bounding the total number of processes to 30, just two times the individual upper limit. An ideal load balancing mechanism would lead to run 15 processes by processor (90 in total, six times the upper limit).

The next section proposes a ring-based work stealing algorithm that provides near-optimal load balancing for D&C MPI-2 programs.

4. Space-Efficient D&C Scheduling in MPICH2

In order to enable spatial-complexity efficiency, we propose a ring-based algorithm to replace the default implementation of MPICH2. It works as follows:

1. *proc* invokes `MPI_Comm_spawn`, sending, through internal function calls, a corresponding message to its MPDMan that, in turn, re-passes it to the corresponding MPD.
2. MPD, after receiving the spawn message from a MPDMan, pushes it on top of Q.
3. Whenever one MPD is not executing any MPI process (or all of the executing ones are waiting for sub-optimal results), it verifies Q and
   (a) if $Q \neq \emptyset$, then it pops the bottom-most message to spawn a process; or
   (b) if $Q = \emptyset$, then it sends a work-stealing message across the communication ring and waits for its answer.
4. When one MPD receives a work-stealing message, if it has any message on Q and the stealing request has no work attached, then it attaches the top-most spawn message of Q to it. Then it forwards the message to the "right" node in the ring.
5. The sender s of a Work Stealing request receives the original message after one complete loop on the ring. If it has a spawn message attached, then s spawns this process locally. If not, s re-sends the message. This cycle continues until the computation reaches its end.

This algorithm is named ring-based work stealing (RBWS), because a given node uses the ring of MPICH2 control messages in order to "steal" waiting-to-be-spawned processes from another node's queue. Work stealing, as stated by [1], is a more efficient way to balance workload because, in opposition to *work-pushing*, it is an idle node which spends time trying to steal work, not an already working one.

RBWS has near-optimal space consumption. To understand why, let us consider an execution of this algorithm. Consider that the program is run on P nodes (1 processor per node). At a given time, if all Qs have spawn messages, then the system is at optimal balancing, even if these Qs have different sizes. Whether a node is not executing any MPI process (or all its processes are blocked at *sync*), then it has an empty Q. This node, then, sends a work-stealing request around the ring. Given that, three situations may happen, exclusively:

1. At least one Q has a spawn message.
2. No Q has any spawn message; the remaining $P-1$ processors are processing under-threshold inputs sequentially.
3. No Q has any spawn message; some of the remaining $P - 1$ processors are executing the phases of *divide* or *conquer* of the input/output. The others (if any) are processing under-threshold inputs sequentially.

If the program is in situation (1), then it will be trivially balanced again after one loop over the ring. If it is in situation (2), then it is already balanced and a distributed ending is in progress, because the sequential final tasks do not produce new processes. Finally, in situation (3), the program is again balanced after $\mathcal{O}(P)$ time, since both *divide* and *conquer* steps take this time, as well as the *spawn* step, which takes a constant number of ring loops.

Thus, in all the cases, in time $\mathcal{O}(P)$, the system will be balanced.

One important remark is that this implementation is not fully transparent. Considering item 3 from RBWS, when all running processes on a given node are executing the *sync* step, this node is able to steal from the queue of another node. Because MPD is unable to directly know when all its processes are blocked on *sync*, we have defined a new primitive, `MPI_Block_notf()`, that explicitly does it, thus redefining the *sync* step as follows:

sync := `MPI_Block_notf()` and non-blocking waiting.

Whenever some process is blocked, waiting for all of its sub-optimal results, MPD is notified by `MPI_Block_notf` and spawns the next process on Q or send work stealing

message. Doing it allows the system to avoid busy-waiting. Effective receive of sub-optimal results is done through pooling, using a combination of MPI's non-blocking receive primitives and UNIX system call `sched_yield`. This approach is showed on Fig. 1(b).

The next section shows some practical results about this discussion.

5. Results

All the experiments have been performed on the cluster ICE of the Parallel and Distributed Processing Group of the Federal University of Rio Grande do Sul. Its configuration is:

- 14 Dell PowerEdge 1950 nodes.
- Each node with two Intel Xeon E5310 Quad Core de 1.60 GHz.
- Each processor with 2×4MB *cache* memory (1066 *Front Side Bus*).
- Node interconnection made by a 3Com 2816 *switch* with Gigabit Ethernet.

All executions have been made on 12 nodes (96 processors) with 30 random-generated input sets, where each input has been executed 5 times (150 executions in total).

Fig. 3 shows the execution time of an algorithm with variable spatial complexity, Mergesort. It shows that the reduction of the memory consumption is considerable on variable memory consumption (2×), while the execution time was nearly the same with all the schedulers.

6. Conclusions and Final Remarks

Experimental results meet our theoretical assumptions: the proposed distributed scheduler for D&C MPI programs, based on spawning dynamic processes, shows near-optimal spatial complexity, with just the same execution time as the default MPICH2 scheduler.

The present work can be seen as a stricter and distributed version of the previously published work [2]. It extends the latter because its distributed nature solves the problem of a centralized bottleneck.

Supporting D&C with dynamic tasks in MPI meets the current trend in parallel programming in shared memory systems: Intel's Thread Building Blocks [5] and OpenMP3 are shared memory examples that address the same problem. Projects like KAAPI [4] are also proposal to use recursivity and task-based parallelism, this time in C++ and for distributed memory. Having this kind of expression of the parallelism natively supported by MPI would be an improvement for a whole set of algorithms.

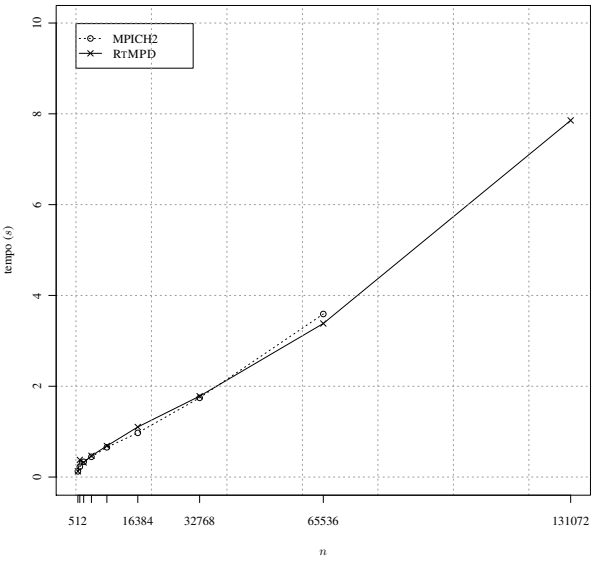

Figure 3. Mergesort execution. Execution time t *vs.* vector size. Black points represent average execution time, with the standard deviation being always ≤ 0.01. The threshold for Mergesort was $n = 256$.

References

[1] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *In Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 356–368, 1994.

[2] M. C. Cera, G. P. Pezzi, E. N. Mathias, N. Maillard, and P. O. A. Navaux. Improving the Dynamic Creation of Processes in MPI-2. *Lecture Notes in Computer Science - 13h European PVM/MPI Users Group Meeting*, 4192/2006:247–255, Sept. 2006.

[3] H. T. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[4] T. Gautier, X. Besseron, and L. Pigeon. KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation*. ACM, 2007.

[5] C. Pheatt. Intel®threading building blocks. *J. Comput. Small Coll.*, 23(4):298–298, 2008.