CPD Report
MAXSAT Solver

Group 17
Helder Duarte N 84982
Samuel Gomes N 76415
Rui Santos N 76384

May 13, 2016

Serial implementation

The serial implementation suffered drastic changes since the last iteration of the project. The most straightforward change is that fact that the binary tree is no longer being generated. Instead, the path itself is coded not by traversing the tree, but by an array of integers representing the variables, for example,

$$[1\ 2\ -3]$$

codes that path going through the positive path of variable 1, then again through the positive path of variable 2, and lastly going through the negative path of variable 3. This implementation is much less memory intensive and thus provides a huge boost in efficiency.

Shared memory implementation with OpenMP

The OpenMP implementation is thus largely simplified, enabling the use of more OpenMP routines. Rather than the static tree division, the dynamic OMP scheduling for a "for" cycle is used.
For this implementation we consider one "task" or "job" as being one assignment in one given position of the tree. It is up to that task to compute the conditions prior to the partial search for that assignment in the function *initConditions()*.
.

As such, a "for" cycle is used to generate one assignment through the binary representation of sequential numbers. For example, while number 4 corresponds to the assignment 100 (var 1 is true, var 2 is false and var 3 is false), the number 7 corresponds to the assignment 111 (all vars are true).

The number of tasks is given by the expression in

$$2^{log_2(number\ of\ processes)+C}$$

(this represents the wanted number of assignments to test) where C is a tunning constant that, according to various tests, gives the optimum number of tasks at $C = 4$.

Distributed memory implementation with MPI

Final implementation

Our implementation is a solution that shares the OpenMP one's simplicity. The available macros for block size allocation are used to statically divide the tasks among the processes. This implementation is the one we used because, although assigning the tasks implicitly, it reveals almost no communication overhead. The downside of this implementation is that if one process finishes before the rest, it will not be assigned more work.

This disadvantage is softened by the usage of openMP in each process (described later in chapter), which we implemented dynamically. As said below, our dynamic approach was not getting as good results as this one (because of not only the implicitly of the task granularity but also the reduction of the communication overhead).

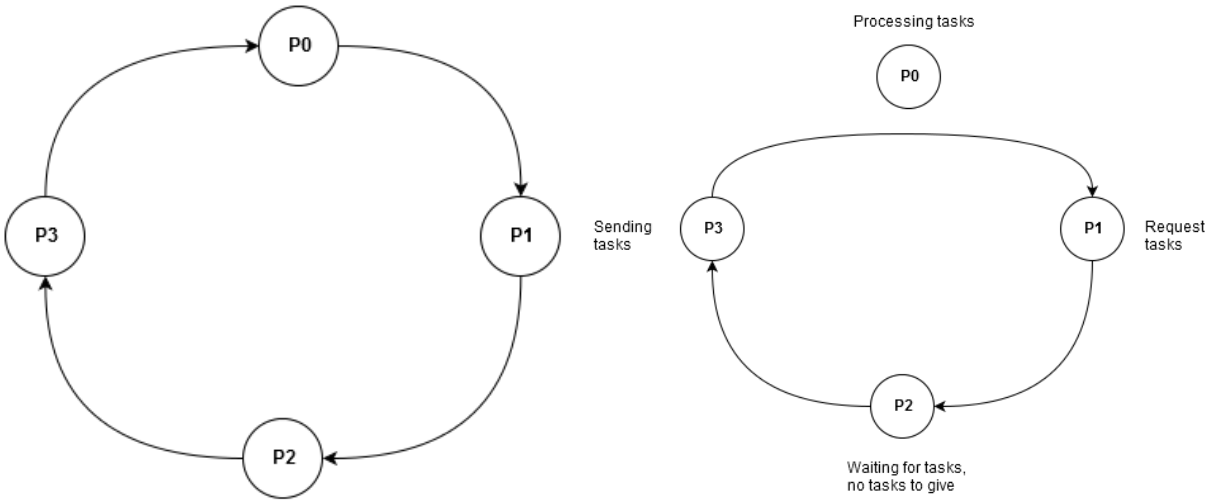
Listing 1: Block Macros

```
#define BLOCK_LOW(id,p,n)   ((id)*(n)/(p))
#define BLOCK_HIGH(id,p,n)  (BLOCK_LOW((id)+1,p,n)-1)
#define BLOCK_SIZE(id,p,n)  (BLOCK_HIGH(id,p,n)-BLOCK_LOW(id,p,n)+1)
#define BLOCK_OWNER(index,p,n)  (((p)*((index)+1)-1)/(n))
```

"

Termination and reduction

After all the processes have done their work, a reduction is done by passing the partial results through the ring and updating the general result the same way as in openMP.



(a) Ring topology, example with 4 processes.

(b) Requesting tasks.

Testing other implementations

A dynamic load balancing version was developed, among others, in addition to the above static load balancing version. It assumes a decentralized approach. In this version, the communication assumes a ring channel where each process communicates to the next process, as shown in figure 1a.

In order to manage communications, a number of tags were used to differentiate messages, namely:

- **TAG_REQUEST_TASKS:** used to request tags to other processes
- **TAG_SEND_TASKS:** used when sending a task after receiving a TAG_REQUEST_TASKS
- **TAG_TOKEN:** used to propagate the token
- **TAG_STOP:** used to stop execution
- **TAG_REDUCTION:** used when agglomerating results

Initially, tasks are allocated statically according to the approach used in the static implementation. After each task completed, the process uses *MPI_Iprobe()* to check if a message was received. If it received a message bearing the tag TAG_REQUEST_TASKS, before beginning work on the next tasks, it sends a 2 integer buffer depicting an interval of tasks to send to the requesting process. The interval of tasks is half of the tasks remaining for the sender process. This message is sent with the tag TAG_SEND_TASKS. It then proceeds to jump the interval of tasks ahead so as not to repeat work given to another process.

After finishing all of his tasks, it checks for a received message with *MPI_Iprobe()*. If it did not receive a message, then it will request more tasks from the next process in the ring. Only after receiving more tasks will it start work on new tasks.

Termination

In order for the program to terminate, a Dual-Pass Ring Termination Algorithm was implemented. It consists of passing a token through the processes in the ring. This token can be colored white or black. The processes also have a color. All processes start with the color white. Process 0 first sends the token with the color white after it finished processing its initial batch of tasks.

A process changes color to black whenever it receives a request for tasks and sends task to the requested process, and the requesting process's rank is lower than the sending process's. A white process forwards the token without changing it's color. A black process always changes the token to black before forwarding it. The tag used to send a token is the TAG_TOKEN.

The termination condition occurs when process 0 receives a white token, which means all other processes are not sending tasks, therefore, no tasks are in transit. When this happens process 0 sends a message with the tag TAG_STOP to all other processes, and when they finish processing their current batch of tasks, they cease requesting more tasks,

breaking out of the loop and ending their processing. At this point, process 0 receives all other processes results to merge into his own.

Ultimately, even though the termination works, most of the processes ended up repeating work done either by other tasks or by themselves, resulting in an incorrect final result. Furthermore, the added communication culminated in an increase of the execution time, making the static implementation preferable.

Hybrid solution with both OpenMP and MPI

Because of the simplicity of our MPI solution, a hybrid solution was easy to implement, and had significant impact on the program performance.

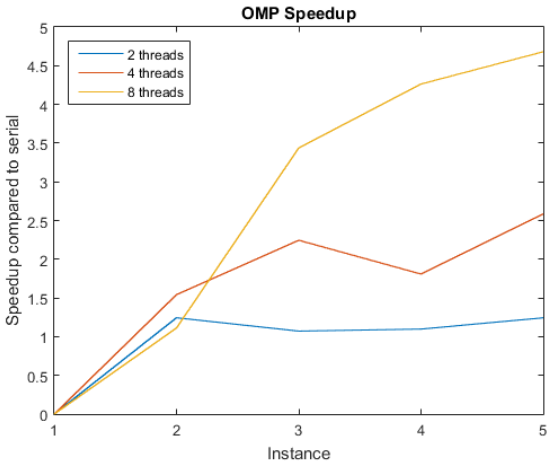
For this version, the for cycle associated with each process tasks was parallelized with openMP directive for (as in the single openMP solution).

With this addition, dynamic parallelization is permitted for each process, and as a result, it is more efficient than the single MPI solution, giving better speedups.

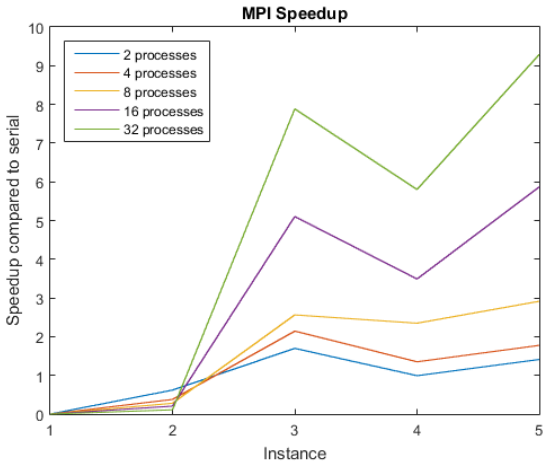
Results

As can be seen by analyzing the following graphs, when comparing the MPI solution with the OMP one, although the OMP one achieves better speedups for up to 8 threads vs 8 processes, MPI is able to scale better and achieve higher speedups with 16 and 32 processes. These are, however, far from ideal, as can be seen in figure 2b, where with 32 processes it is achieved a speedup of 9 at most.

Using OMP in conjunction with MPI achieves the best results, with a speedup reaching up to 20. It can be seen, however, that the higher the number of processes used for MPI, the lower is the influence of OMP.



(a) Speedup of OMP solutions compared to serial.



(b) Speedup of MPI solutions compared to serial.

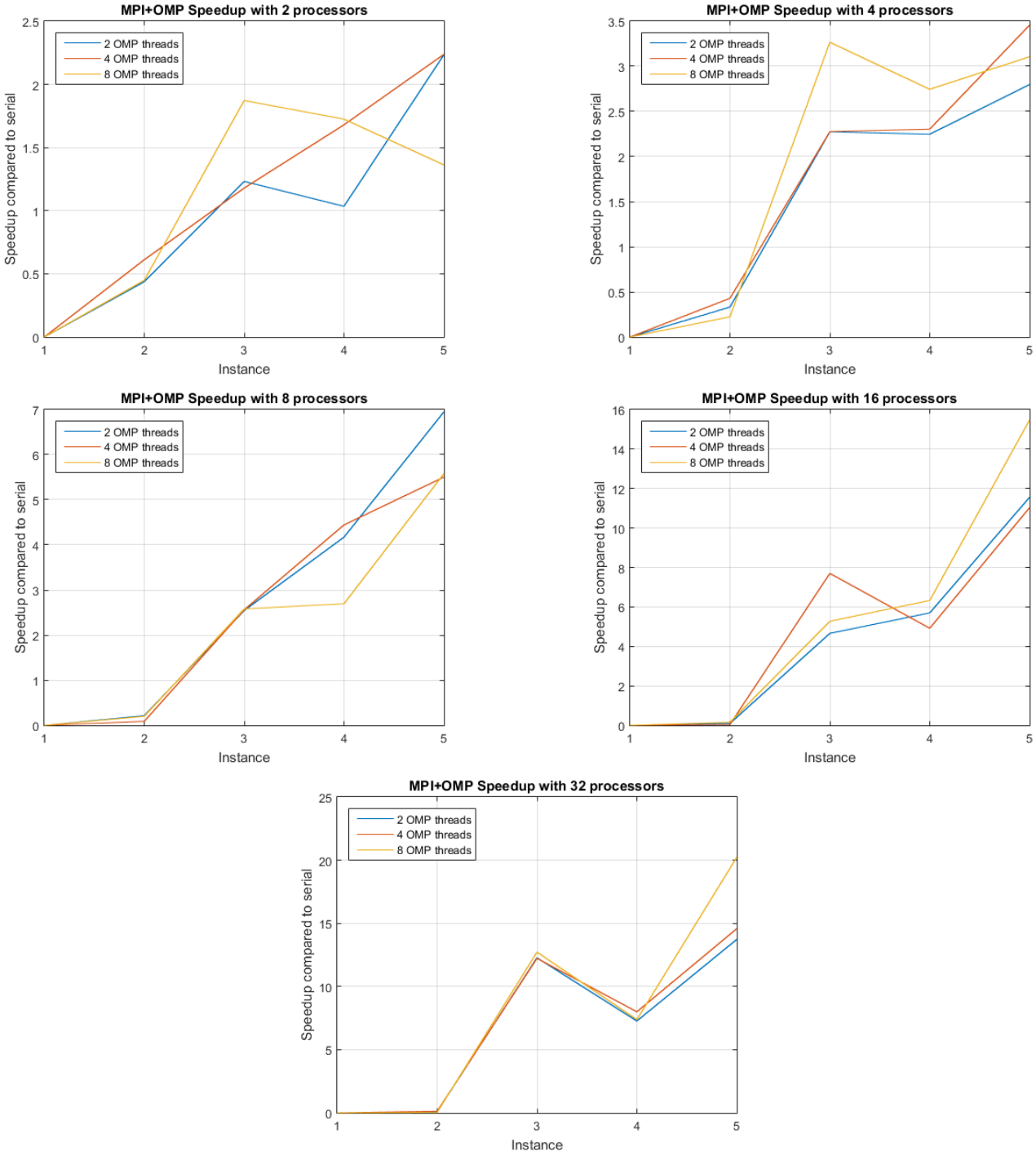


Figure 3: Requesting tasks.