

On-line Scheduling of MPI-2 Programs with Hierarchical Work Stealing

Guilherme P. Pezzi
Márcia C. Cera
Elton N. Mathias
Nicolas Maillard
Philippe O. A. Navaux

19 th International Symposium on Computer Architecture and High Performance Computing - SBAC-PAD 2007



Claim

- It is hard to **schedule** MPI-2 programs **efficiently**
 - There are recursive and dynamic creation of processes and tasks
- But you can do it with **Hierarchical Work Stealing**
- This improves the **performance** of the application

2

Message Passing Interface

- MPI - Message Passing Interface
 - Standard for HPC programming
- MPI-1 (1994)
 - Static processes creation
- MPI-2 (1998)
 - Extend MPI-1 features
 - Dynamic process creation (`MPI_Comm_spawn()`)
 - Remote memory access
 - Parallel I/O
 - Recently implemented
 - LAMMPI (2000), MPICH (2005), OpenMPI (2006)

3

Message Passing Interface

- How to program with dynamic processes?
 - Recursive-like programming model
 - Divide and Conquer (D&C)
- On-line scheduling of dynamic programs
 - Work Stealing
 - Efficient to schedule D&C programs, used in Cilk and Satin
 - How to adapt the Work Stealing algorithm to MPI-2 context?
 - Using an **Hierarchical** structure
- This work presents an on-line scheduling strategy based in Work Stealing, called **Hierarchical Work Stealing**

4

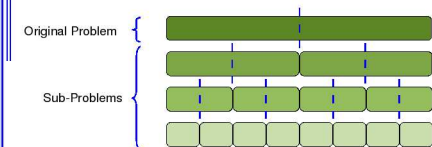
Outline

- Motivation
- MPI-2 Dynamic Programs
 - Parallel D&C
 - Scheduling D&C with Work Stealing
- Hierarchical Work Stealing
- Experimental Evaluation
- Conclusion

5

MPI-2 Dynamic Programs

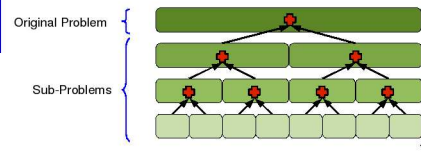
- Divide and Conquer (D&C) Model
 - Divide Phase:
 - Problem is recursively **divided** into sub-problems until their solutions become trivial



6

MPI-2 Dynamic Programs

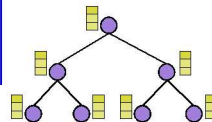
- Divide and Conquer (D&C) Model
 - Conquer Phase:
 - The sub-problems solution are **merged** to compute the original problem solution



7

Scheduling D&C with Work Stealing

- Work Stealing [Blumofe and Leiserson, 1998]

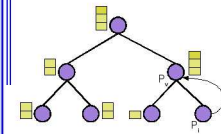


- Work Stealing works with Strict computations
- Each processor maintains a deque (double ended queue)
- Processor pushes a task to compute on top of the deque

8

Scheduling D&C with Work Stealing

- Work Stealing [Blumofe and Leiserson, 1998]

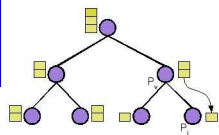


- Processor P_i gets idle and its deque is empty: starts stealing
- Processor P_i chooses a victim P_v

9

Scheduling D&C with Work Stealing

- Work Stealing [Blumofe and Leiserson, 1998]

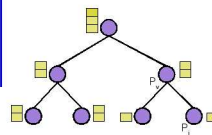


- The processor P_i receives the stealing request
 - It pops a task from the bottom of its deque
 - It sends the task to the stealing processor P_i

10

Scheduling D&C with Work Stealing

- Work Stealing [Blumofe and Leiserson, 1998]



- Devised in Cilk
- Adapted in Satin
 - Scheduling in cluster of clusters
- How to implement it in MPI-2?
 - Communication in Parent/children relationship
- Hierarchical Work Stealing (HWS)**

11

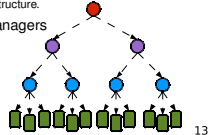
Outline

- Motivation
- MPI-2 Dynamic Programs
- Hierarchical Work Stealing
 - Hierarchical Work Stealing Algorithm
- Experimental Evaluation
- Conclusion

12

Outline of the HWS algorithm

- The CPUs will run:
 - Workers** processes
 - They perform the computation of the tasks.
 - Managers** processes
 - They handle the deque and route the stealing requests;
 - They are organized in a tree structure.
- Internal node in the tree = managers
- Leaves = workers.



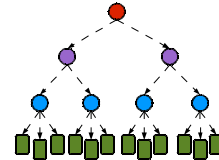
13

HWS Algorithm

```

1  LEAF then
2    recv(parent, tasks);
3    partial-result = Compute(tasks);
4    send(parent, partial-result);
5  else
6    if ROOT then
7      deque.push(new_tasks);
8    else
9      recv(parent, tasks);
10     deque.push(tasks);
11   end if
12   if #processors > x then
13     create-managers();
14   else
15     create-leaves();
16   end if
17   if ROOT then
18     return(result);
19   else
20     send(parent, partial-result)
21   end if

```



14

HWS Algorithm

```

1  LEAF then
2    recv(parent, tasks);
3    partial-result = Compute(tasks);
4    send(parent, partial-result);
5  else
6    if ROOT then
7      deque.push(new_tasks);
8    else
9      recv(parent, tasks);
10     deque.push(tasks);
11   end if
12   if #processors > x then
13     create-managers();
14   else
15     create-leaves();
16   end if
17   if ROOT then
18     return(result);
19   else
20     send(parent, partial-result)
21   end if

```



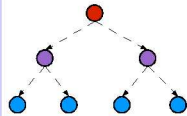
15

HWS Algorithm

```

1  for i = 1 ... O do
2    spawn (child[i]);
3    tasks = deque(pop);
4    send(child[i], tasks);
5    irrecv(child[i], partial-result);
6  end for
7  while there are children do
8    wait-any();
9    if !deque.empty() then
10     tasks = deque(pop);
11     send(src, tasks);
12     irrecv(src, partial-result);
13   else
14     if father-has-tasks then
15       send(parent, partial-result);
16       recv(parent, tasks);
17     end if
18   end if

```



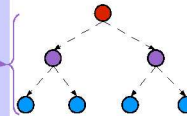
16

HWS Algorithm

```

1  for i = 1 ... O do
2    spawn (child[i]);
3    tasks = deque(pop);
4    send(child[i], tasks);
5    irrecv(child[i], partial-result);
6  end for
7  while there are children do
8    wait-any();
9    if !deque.empty() then
10     tasks = deque(pop);
11     send(src, tasks);
12     irrecv(src, partial-result);
13   else
14     if father-has-tasks then
15       send(parent, partial-result);
16       recv(parent, tasks);
17     end if

```



Stealing Request

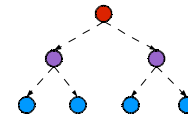
17

HWS Algorithm

```

1  LEAF then
2    recv(parent, tasks);
3    partial-result = Compute(tasks);
4    send(parent, partial-result);
5  else
6    if ROOT then
7      deque.push(new_tasks);
8    else
9      recv(parent, tasks);
10     deque.push(tasks);
11   end if
12   if #processors > x then
13     create-managers();
14   else
15     create-leaves();
16   end if
17   if ROOT then
18     return(result);
19   else
20     send(parent, partial-result)
21   end if

```



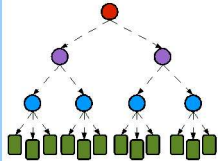
18

HWS Algorithm

```

for i in processors do
  spawn (child[i]);
  tasks = deque(pop);
  send(child[i], tasks);
  irecv(child[i], partial-result);
end for
while there are children do
  wait-any();
  tasks = deque(pop);
  if task != 0 then
    spawn(child);
    send(src, tasks);
    irecv(src, partial-result);
  else
    if father-has-tasks then
      send(parent, partial-result);
      recv(parent, tasks);
    end if
  end if
end if

```



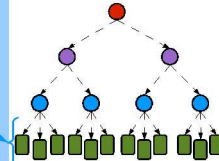
19

HWS Algorithm

```

for i in processors do
  spawn (child[i]);
  tasks = deque(pop);
  send(child[i], tasks);
  irecv(child[i], partial-result);
end for
while there are children do
  wait-any();
  tasks = deque(pop);
  if task != 0 then
    spawn(child);
    send(src, tasks);
    irecv(src, partial-result);
  else
    if father-has-tasks then
      send(parent, partial-result);
      recv(parent, tasks);
    end if
  end if
end if

```



Stealing Request

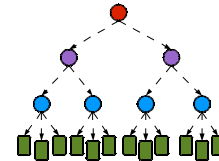
20

HWS Algorithm

```

if ROOT then
  recv(parent, tasks);
  partial-result = Compute(tasks);
  send(parent, partial-result);
else
  if ROOT then
    deque.push(new_tasks);
  else
    recv(parent, tasks);
    deque.push(tasks);
  end if
  if #processors > #tasks then
    create-managers();
  else
    create-leaves();
  end if
  if ROOT then
    return(result);
  else
    send(parent, partial-result);
  end if
end if

```



21

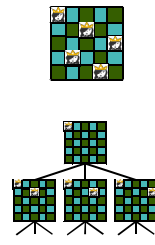
Outline

- Motivation
- MPI-2 Dynamic Programs
- Hierarchical Work Stealing
- Experimental Evaluation
 - The N-Queens Application
 - Performance: MPI vs. Satin
 - Scheduling in Dynamic Environment
 - Hierarchical Performance
- Conclusion

22

N-Queens Application

- Placing N queens in a $N \times N$ chessboard
 - No queen may capture any other
- Backtracking algorithm
 - Placing recursively and exhaustively the queens, row by row

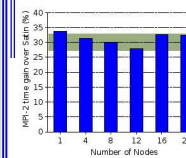


23

Performance of MPI with HWS vs. Satin

- Run-time of N-Queens with MPI-2 vs. Satin: sequential case

	MPI run-time (sec.)		Satin run-time (sec.)	
N	Average	Std. Dev.	Average	Std. Dev.
16	11.91	0.01	15.74	0.01
17	76.79	0.28	108.75	0.06
18	552.88	0.14	790.75	0.42
19	4252.88	0.57	6085.02	0.38

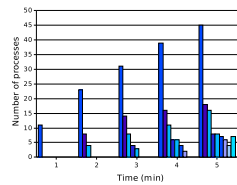


- Run-time gain of the N-Queens with MPI-2 over Satin: the parallel case ($N=18$):
 - MPI is better than Satin.
 - Maintains the same scalability.

24

Scheduling in Dynamic Environment

- Number of processes on processor vs. time. A new node is added each minute.

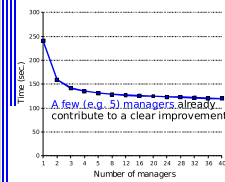


- In a distributed platform:
 - MPI-2 with HWS uses dynamically the CPUs

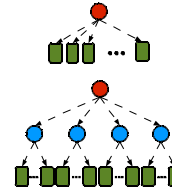
25

Impact of the tree structure

- For a fixed number of processors (40), different numbers of managers can be chosen



A few (e.g. 5) managers already... contribute to a clear improvement.



26

Outline

- Motivation
- MPI-2 Dynamic Programs
- Hierarchical Work Stealing
- Experimental Evaluation
- Conclusion

27

Conclusion

- MPI-2 can be used to implement D&C programs
 - Dynamic process creation to implement recursive calls – the divide phase
- Problem: How to schedule tasks in a dynamic and distributed way?
 - We propose Hierarchical Work Stealing
 - Work Stealing offer distributed efficient scheduling devised by Cilk
 - Hierarchical to follow the parent/children communication relationship
 - HWS evaluation:
 - Better performance than Satin
 - Can be used in dynamic environments

28

On-line Scheduling of MPI-2 Programs with Hierarchical Work Stealing

Guilherme P. Pezzi
Márcia C. Cera
Elton N. Mathias
Nicolas Maillard
Philippe O. A. Navaux

19th International Symposium on Computer Architecture and High Performance Computing – SBAC-PAD 2007

