Parallelizing ASP and SOR algorithms

Experiences in parallelizing well known algorithms using MPI

Sander van Loo and Erik van Zijst
January 20, 2003

Vrije Universiteit,
Faculty of Sciences, Department of Mathematics and Computer Science,
Amsterdam, the Netherlands

{sander,erik}@marketxs.com

1. Abstract

This text describes our experiences in parallelizing well known algorithms such as ASP and SOR using the Message Passing Interface (MPI). Execution time and speedup are measured on the DAS multicomputer at the Vrije Universiteit in Amsterdam. It is a wide-area distributed cluster designed by the Advanced School for Computing and Imaging (ASCI). Currently, DAS consists of only one cluster of 64 nodes that are interconnected using a fast (1.28 Gb/s) Myrinet network. Both applications are run on different numbers of processors with varying problem sizes. SOR experiments are done with two different distributions.

2. All-pairs Shortest Paths

2.1. Introduction

The goal of the All-pairs Shortest Paths (ASP) problem is to find the shortest path between any pair of nodes in a graph.

2.2. Distribution

The parallel ASP application uses a row wise distribution of the distance table. The master process (rank 0) initializes the distance table and distributes the data to the other processors. The total number of rows is divided by the number of processors. Every processor is assigned N/P contiguous rows. Any rows not assigned to processors because N cannot be equally divided by P are assigned to the master processor. This results in a slight load-imbalance but can be ignored if N >> P.

2.3. Parallelization

The processor owning row k broadcasts this row to the other processors using MPI_Bcast at the start of iteration. Other processors calculate the rank of the processor owning row k and also use MPI_Bcast with the owner's rank as root to receive row k. All processors use row k to update their local rows for iteration k. At the end of the iteration the processors synchronize using MPI_Barrier.

```
for (k = 0; k < n; k++)
{
  if (k >= lb && k < ub) // i have row k
  {
    MPI_Bcast(rowk, n, MPI_INT, my_rank, MPI_COMM_WORLD);
  }
  else // i do not have row k
  {
    MPI_Bcast(rowk, n, MPI_INT, root, MPI_COMM_WORLD);
  }
  for (i = lb; i < ub; i++)
  {
    if (i != k)
    {
      for (j = 0; j < table->cols; j++)
      {
        table->storage[i - lb][j] = minimum(table->storage[i - lb][j],
                                    table->storage[i - lb][k] + rowk[j]);
      }
    }
  }
  MPI_Barrier(MPI_COMM_WORLD);
}
```

Code above is simplified. Debugging statements and root calculation are left out.

2.4. Performance

Performance measurements are done with 1, 2, 4, 8, 16 and 32 processors. Also three different problem sizes are used to examine the result of the problem size on the performance of the parallel algorithm. All processors synchronize before entering the ASP computation. Execution time is calculated as the time difference between the entry and exit of the ASP computation. Since all processors also synchronize at end of iteration execution time of all processors is equal. All processors compute an equal number of iterations, only the master processor may need to operate on a small number of additional rows possibly slowing down the slave processors.

		Processors					
		1	**2**	**4**	**8**	**16**	**32**
	100	0,321	0,186	0,125	0,108	0,107	0,117
N	**1000**	345	174	88,3	45,7	26,2	15,4
	4000	22024	11015	5524	2779	1410	728

Table 1 Execution time of the ASP computation in seconds.

With a problem size of 100 the overhead of communication between processors is clearly killing performance when many processors are used. However, as the problem size increases the efficiency improves to an eventual speedup of 30.25 on 32 processors.

		Processors					
		1	**2**	**4**	**8**	**16**	**32**
	100	1	1,73	2,57	2,97	3,00	2,74
N	**1000**	1	1,98	3,91	7,55	13,17	22,40
	4000	1	2,00	3,99	7,93	15,62	30,25

Table 2 Speedup of the ASP computation.

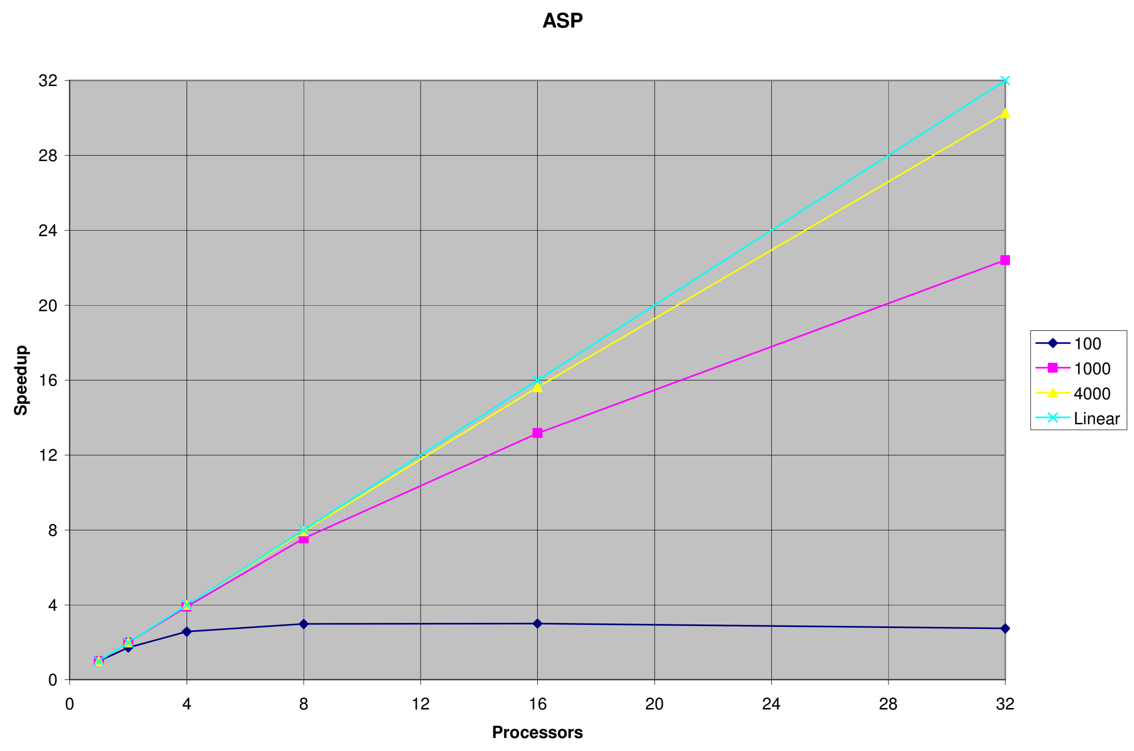

Figure 1 Speedup of the ASP computation.

3. Successive Over-relaxation (row distributed)

3.1. Introduction

Successive Over-relaxation is an iterative algorithm for solving Laplace equations on a grid. The sequential algorithm works as follows. For all non-boundary points of the grid, SOR first calculates the average value of the four neighbors of the point:

$$av(r,c) = \frac{g[r+1,c+1] + g[r+1,c-1] + g[r-1,c+1] + g[r-1,c-1]}{4} \qquad (1)$$

Then the new value of the point is determined using the following correction:

$$g[r,c] = g[r,c] + \omega(av(r,c) - g[r,c]) \qquad (2)$$

w is known as the relaxation parameter and a suitable value can be calculated in the following way:

$$\omega = \frac{2}{1 + \sqrt{1 - (\frac{\cos\frac{\pi}{totalcolumns} + \cos\frac{\pi}{totalrows}}{2})^2}} \qquad (3)$$

The entire process terminates if during the last iteration no point in the grid has changed more than a certain quantity.

The parallel implementation of SOR is based on the Red/Black SOR algorithm. The grid is treated as a checkerboard and each iteration is split into two phases, red and black. During the red phase only the red points of the grid are updated. Red points only have black neighbors and no black points are changed during the red phase. During the black phase, the black points are updated in a similar way. Using the Red/Black SOR algorithm the grid can be partitioned among the available processors, each processor receiving a number of adjacent rows. All processors can now update different points of the same color in parallel. Before a processor starts the update of a certain color, it exchanges the border points of the opposite color with its neighbor. After each iteration, the processors must decide if the calculation must continue. Each processor determines if it wants to continue and sends its vote to the first processor. The calculation is terminated only if all the processors agree to stop. This means a processor may continue the calculation after the termination condition is reached locally.

3.2. Distribution

The parallel SOR application uses a row wise distribution of the grid. The master process (rank 0) initializes the grid and distributes the data to the other processors. The total number of rows is divided by the number of processors. Every processor is assigned N/P contiguous rows. Any rows not assigned to processors because N cannot be equally divided by P are assigned to the master processor. This results in a slight load-imbalance but can be ignored if N >> P.

3.3. Parallelization

At start of a phase in each iteration every processor sends its boundary rows (if present) to its neighbor processors using MPI_Send. After sending out its boundary rows it receives boundary rows from its neighbor processors with MPI_Recv. At end of iteration all processors synchronize their local maximum differences using the MPI_Allreduce call in conjunction with MPI_MAX. This results in global maximum of all processors to be present on each individual processor so that each processor can check if it should terminate.

```
do
{
  maxdiff = 0.0;
  for ( phase = 0; phase < 2 ; phase++)
  {
    // send lower boundary to previous processor
    if (my_rank != 0)
      MPI_Send(table->storage[offset], table->cols, MPI_DOUBLE, my_rank - 1, 0,
               MPI_COMM_WORLD);
    // send upper boundary to next processor
    if (my_rank != (processors - 1))
      MPI_Send(table->storage[ub - lb - 1 + offset], table->cols, MPI_DOUBLE,
               my_rank + 1, 0, MPI_COMM_WORLD);

    // receive next processor's lower boundary
    if (my_rank != (processors - 1))
      MPI_Recv(table->storage[ub - lb + offset], table->cols, MPI_DOUBLE,
               my_rank + 1, 0, MPI_COMM_WORLD, &status);
    // receive previous processor's upper boundary
    if (my_rank != 0)
      MPI_Recv(table->storage[0], table->cols, MPI_DOUBLE, my_rank - 1, 0,
               MPI_COMM_WORLD, &status);

    for ( i = lb + (lb == 0 ? 1 : 0) ; i < ub - (ub == n ? 1 : 0) ; i++ )
    {
      for ( j = 1 + (even(i) ^ phase); j < table->cols - 1 ; j += 2 )
      {
        Gnew = stencil(table->storage, i - lb + offset, j);
        diff = abs_d(Gnew - table->storage[i - lb + offset][j]);
        if ( diff > maxdiff )
          maxdiff = diff;
        table->storage[i - lb + offset][j] = table->storage[i - lb + offset][j] +
                                             omega * (Gnew - table->storage[i - lb + offset][j]);
      }
    }
  }
  MPI_Allreduce(&maxdiff, &global_maxdiff, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
  iteration++;
}
while (global_maxdiff > stopdiff);
```

Offset is used to accommodate row lb – 1 (the boundary rows).

3.4. Performance

Performance measurements are done on 1, 4, 9, 16, 25 and 36 processors. Each time the computation was performed with three different problem sizes. A fourth problem size was tried (N = 4000) but this required more than 120 MB of memory on the master node (while total available memory on each node is only 128 MB) leaving little room for the operating system and other processes. On a single node this resulted in very bad performance due to trashing. When using more processors the entire memory is accessed for initialization and distribution of the data, but in the actual computation only a small part of the allocated memory is actually used leaving the unused part of the grid safely swapped to disk. The smaller the grid partition the larger the part of the grid that can be swapped out. Only when collecting the results from other processors the swapped out part of the grid needs to be accessed again. This memory swapping problem distorts the speedup measurements of this problem size and the results are therefore left out of the graphical results.

		Processors					
		1	4	9	16	25	36
N	1000	23,8	6,24	2,81	1,84	1,09	1,41
	2000	49,8	12,9	5,71	3,31	2,16	1,98
	3000	78	20,2	8,86	5,32	3,32	2,58
	4000	2025	39,2	15,0	7,40	5,41	3,93

Table 3 Execution time in seconds for the row distributed SOR computation. Note the extreme difference between the execution time for N = 3000 and N = 4000 on a single CPU.

Though less efficient than the ASP algorithm the parallel SOR displays reasonable efficiency when the problem size is large enough. Communication is less efficient than in the ASP algorithm. The ASP algorithm broadcasts only a single row (and performs synchronization) per iteration while SOR needs to send a maximum of two rows per phase (there are two phases) and single MPI_Allreduce per iteration. Send operations between different processors can be performed in parallel provided enough bandwidth is available.

		Processors					
		1	4	9	16	25	36
N	1000	1	3,81	8,47	12,9	21,8	16,9
	2000	1	3,86	8,72	15,0	23,1	25,2
	3000	1	3,86	8,80	14,7	23,5	30,2
	4000	1	51,66	135,00	273,6	374,3	515,3

Table 4 Speedup of the row distributed SOR computation. The super linear speedup for N = 4000 is caused by thrashing when running on a single processor.

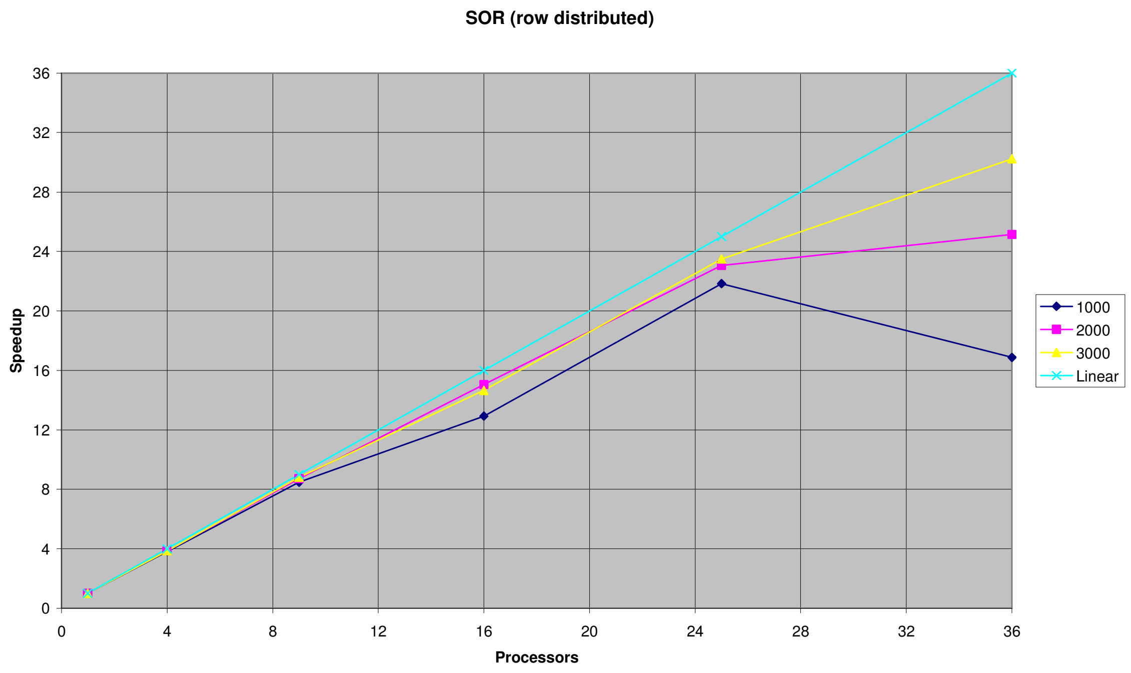

Figure 2 Speedup of a row-distributed SOR computation.

3.5. Verification

The results of this parallelized SOR version are compared against the sequential version. Since the parallel version uses the same input data as the sequential version with equal problem size both the number of iterations and the maximum difference should be equal. This was the case for all problem sizes and number of processors tested.

4. Successive Over-relaxation (block distributed)

4.1. Introduction

Successive Over-relaxation is an iterative algorithm for solving Laplace equations on a grid. The sequential algorithm works as follows. For all non-boundary points of the grid, SOR first calculates the average value of the four neighbors of the point:

$$av(r,c) = \frac{g[r+1,c+1] + g[r+1,c-1] + g[r-1,c+1] + g[r-1,c-1]}{4} \quad (1)$$

Then the new value of the point is determined using the following correction:

$$g[r,c] = g[r,c] + \omega(av(r,c) - g[r,c]) \quad (2)$$

w is known as the relaxation parameter and a suitable value can be calculated in the following way:

$$\omega = \frac{2}{1 + \sqrt{1 - (\frac{\cos\frac{\pi}{totalcolumns} + \cos\frac{\pi}{totalrows}}{2})^2}} \quad (3)$$

The entire process terminates if during the last iteration no point in the grid has changed more than a certain quantity.

The parallel implementation of SOR is based on the Red/Black SOR algorithm. The grid is treated as a checkerboard and each iteration is split into two phases, red and black. During the red phase only the red points of the grid are updated. Red points only have black neighbors and no black points are changed during the red phase. During the black phase, the black points are updated in a similar way. Using the Red/Black SOR algorithm the grid can be partitioned among the available processors, each processor receiving a number of adjacent rows. All processors can now update different points of the same color in parallel. Before a processor starts the update of a certain color, it exchanges the border points of the opposite color with its neighbor. After each iteration, the processors must decide if the calculation must continue. Each processor determines if it wants to continue and sends its vote to the first processor. The calculation is terminated only if all the processors agree to stop. This means a processor may continue the calculation after the termination condition is reached locally.

4.2. Distribution

The block distributed version of the parallel SOR application partitions the grid not in sections of contiguous rows but in rectangular blocks. Each processor is assigned a block of N/sqrt(P) by N/sqrt(P) elements. To simplify the implementation the application only accepts problem sizes for which N can be equally divided by sqrt(P). The master process (rank 0) initializes the grid and distributes the data to the other processors.

4.3. Parallelization

At start of a phase in each iteration every processor sends its boundary rows (if present) to its neighbor processors using MPI_Send. After sending out its boundary rows it receives boundary rows from its neighbor processors with MPI_Recv. At end of iteration all processors synchronize their local maximum differences using the MPI_Allreduce call in conjunction with MPI_MAX. This results in global maximum of all processors to be present on each individual processor so that each processor can check if it should terminate. The implementation is not much different from the row distributed version. The main difference is that each processor sends a maximum of 4 messages, two rows and two columns.

4.4. Performance

Performance measurements are done on 1, 4, 9, 16, 25 and 36 processors. Each time the computation was performed with three different problem sizes. N was changed in a number of cases to ensure that N+2 can be equally divided by sqrt(P). For N = 1000 N was set to 1002 when running on 16 processors and to 1003 when running on 25 processors. For N = 2000 N was set to 2002 for 9, 16 and 36 processors and set to 2003 for 25 processors. For N = 3000 N was set to 3001 for 9 processors, 3002 for 16 processors, 3003 for 25 processors and 3004 for 36 processors.

		Processors					
		1	4	9	16	25	36
	1000	29,2	7,58	3,51	2,02	1,32	0,83
N	2000	61,1	16,0	7,26	4,04	2,62	1,84
	3000	95,7	25,1	11,3	6,31	4,10	2,70

Table 5 Execution time in seconds of the block distributed SOR application.

The block distributed version of SOR suffers from less communication overhead than the row distributed version when the number of processors is large. Each processor sends a maximum of 4 messages of size N/sqrt(P) elements (two columns, two rows). Send operations between different processors can be performed in parallel provided that enough bandwidth is available. In the row distributed version each processor sends a maximum of 2 messages of size N elements to other processors. This means that if sqrt(P) > 2 the block distributed version actually sends less elements per processor than the row distributed version (there is of course additional communication overhead that can be contributed to message headers etc). The effects of this strategy become clear when looking at the speedup of the 36 processor run with N = 1000. The row distributed version only achieves a speedup of 16.9 while the block distributed version achieves a speedup of 35.2 when running on 36 processors. The break even point appears to lie in the neighborhood of 25 processors.

		Processors					
		1	4	9	16	25	36
	1000	1	3,85	8,32	14,5	22,1	35,2
N	2000	1	3,82	8,42	15,1	23,3	33,2
	3000	1	3,81	8,47	15,2	23,3	35,4

Table 6 Speedup of the block distributed SOR application.

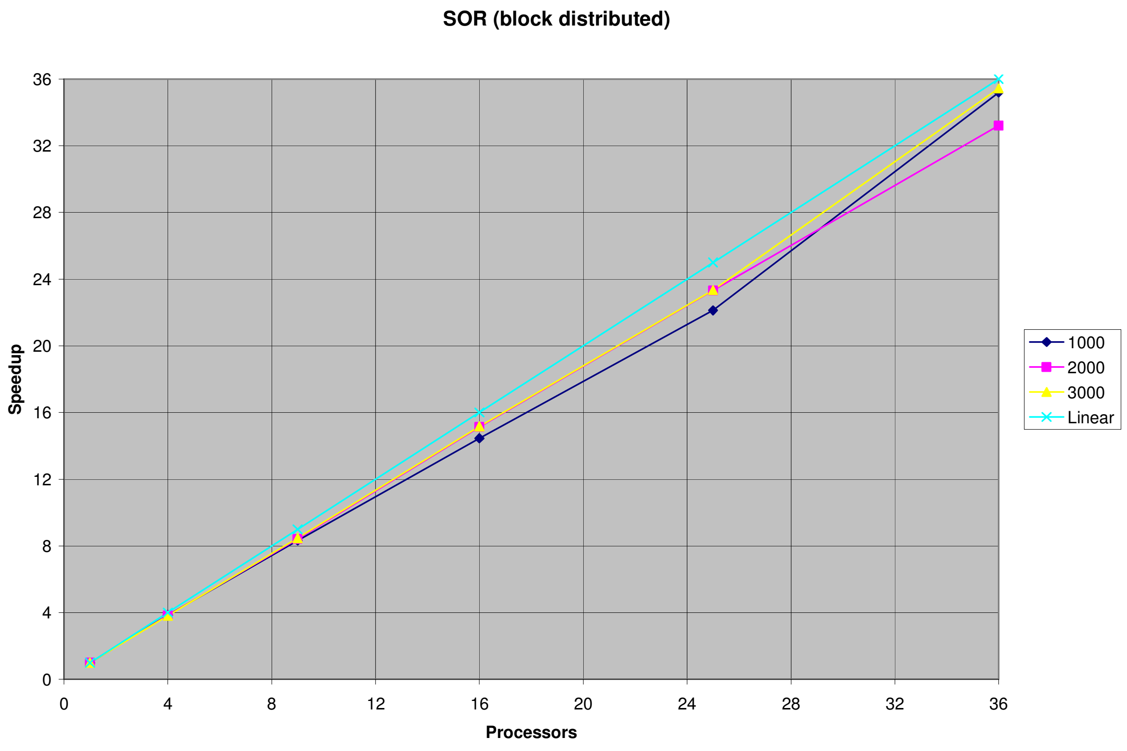

Figure 3 Speedup of the block distributed SOR application.

4.5. Verification

The results of this parallelized SOR version are compared against the sequential version. Since the parallel version uses the same input data as the sequential version with equal problem size both the number of iterations and the maximum difference should be equal. This was the case for all problem sizes and number of processors tested.