Appears in International Parallel and Distributed Processing Symposium 2000

Design and Evaluation of I/O Strategies for Parallel Pipelined STAP Applications

Wei-keng Liao
Alok Choudhary

ECE Department
Northwestern University
Evanston, IL 60208

Donald Weiner
Pramod Varshney

EECS Department
Syracuse University
Syracuse, NY 13244

Abstract

This paper presents experimental results for a parallel pipeline STAP system with I/O task implementation using the parallel file systems on the Intel Paragon and the IBM SP. In our previous work, a parallel pipeline model was designed for radar signal processing applications on parallel computers. Based on this model, we implemented a real STAP application which demonstrated the performance scalability of this model in terms of throughput and latency. In this paper, we study the effect on system performance when the I/O task is incorporated in the parallel pipeline model. There are two alternatives for I/O implementation: embedding I/O in the pipeline or having a separate I/O task. From these two I/O implementations, we discovered that the latency may be improved when the structure of the pipeline is reorganized by merging multiple tasks into a single task. All the performance results shown in this paper demonstrated the scalability of parallel I/O implementation on the parallel pipeline STAP system.

1. Introduction

In this paper we build upon our earlier work where we devised strategies for high performance parallel pipeline implementations, in particular, for Space-Time Adaptive Processing (STAP) applications [2, 5]. A modified Pulse Repetition Interval (PRI)-staggered post-Doppler STAP algorithm [1, 6, 7] was implemented based on the parallel pipeline model and scalable performance was obtained both on the Intel Paragon and the IBM SP. Normally, this parallel pipeline system does not include disk I/O costs, since most radar applications require signal processing in real time. Thus far we have assumed that the signal data collected by radar is directly delivered to the pipeline system, as shown in the overall radar signal processing system of Figure 1.

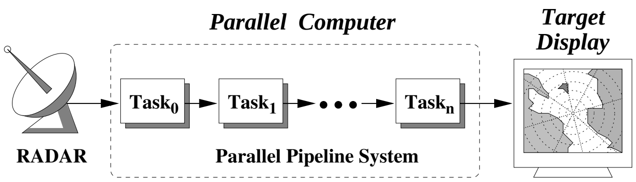

Figure 1. Data flow of a radar and signal processing system using parallel computers.

In practice, the I/O can be done either directly from a radar or through disk file systems. In this work we focus on the I/O implementation of the parallel pipeline STAP algorithm when I/O is carried out through a disk file system. Using existing parallel file systems, we investigate the impact of I/O on the overall pipeline system performance. Two designs of I/O are employed: in the first design the I/O is embedded in the pipeline without changing the task structure and in the other a separate task is created to perform I/O operations. With different I/O strategies, we ran the parallel pipeline STAP system portably and measured the performance on the Intel Paragon at California Institute of Technology and on the IBM SP at Argonne National Laboratory (ANL.) The parallel file systems on both the Intel Paragon and the IBM SP contain multiple stripe directories for applications to access disk files efficiently. On the Paragon, two PFS file systems with different stripe factors were tested and the results were analyzed to assess the effects of the size of the stripe factor on the STAP pipeline system. On the IBM SP, the performance results were obtained by using the native parallel file system, PIOFS, which has 80 stripe directories.

Comparing the two parallel file systems with different stripe sizes on the Paragon, we found that an I/O bottleneck results when a file system with smaller stripe size is used. Once a bottleneck appears in a pipeline, the throughput which is determined by the task with maximum exe-

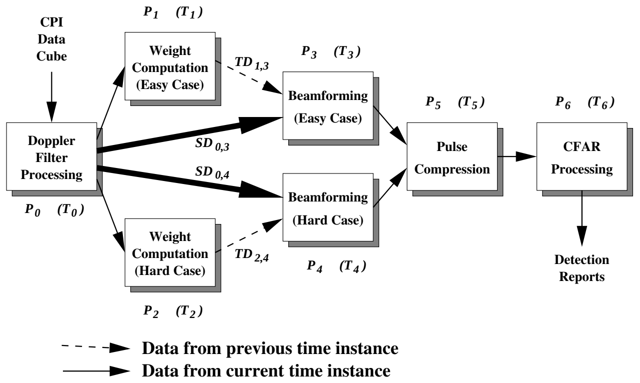

Figure 2. Implementation of parallel pipelined STAP. Arrows connecting task blocks represent data transfer between tasks.

cution time degrades significantly. On the other hand, the latency is not significantly affected by the bottleneck problem. This is because the latency depends on all the tasks in the pipeline rather than the task with the maximum execution time. Furthermore, when evaluating the performance results of the two I/O designs, we observed that the latency can be improved by merging two tasks in the pipeline. In this paper, we also examine the possibility of improving latency by reorganizing the task structure of the STAP pipeline system.

The rest of the paper is organized as follows: in Section 2, we briefly describe our previous work, the parallel pipeline implementation on a STAP algorithm. The parallel file systems tested in this work are described in Section 3. The I/O design and implementation are presented in Section 4 and their performance results are given in Section 5. Section 6 presents the results when tasks are combined.

2. Parallel pipeline STAP system

In our previous work [2, 5], we described the parallel pipelined implementation of a PRI-staggered post-Doppler STAP algorithm. The parallel pipeline system consists of seven tasks: 1) Doppler filter processing, 2) easy weight computation, 3) hard weight computation, 4) easy beamforming, 5) hard beamforming, 6) pulse compression, and 7) CFAR processing. The design of the parallel pipelined STAP algorithm is shown in Figure 2.

The input data set for the pipeline is obtained from a phased array radar and is formed in terms of a coherent processing interval (CPI). Each CPI data set is a 3-dimensional complex data cube. The output of the pipeline is a report on the detection of possible targets. Each task i, $0 \leq i < 7$, is parallelized by evenly partitioning its work load among P_i compute nodes. The execution time associated with task i is T_i. For the computation of the weight vectors for the current CPI data cube, data cubes from previous CPIs are used as input data. This introduces temporal data dependency. Temporal data dependencies are represented by arrows with dashed lines in Figure 2 where $TD_{i,j}$ represents temporal data dependency of task j on data from task i. In a similar manner, spatial data dependencies $SD_{i,j}$ can be defined and are indicated by arrows with solid lines.

Throughput and latency are two important measures for performance evaluation on a pipeline system.

$$throughput = \frac{1}{\max\limits_{0 \leq i < 7} T_i}. \tag{1}$$

$$latency = T_0 + \max(T_3, T_4) + T_5 + T_6. \tag{2}$$

The temporal data dependency does not affect the latency because weight computation tasks use data from the previous time instance rather than the current CPI. The filtered CPI data cube sent to the beamforming task does not wait for the completion of its weight computation. This explains why Equation (2) does not contain T_1 and T_2.

3. Parallel file systems

We used the parallel I/O library developed by Intel Paragon and IBM SP systems to perform read operations. The Intel Paragon OSF/1 operating system provides a special file system type called PFS, for Parallel File System, which gives applications high-speed access to a large amount of disk storage [4]. In this work, two PFS file systems at Caltech were tested : one has 16 stripe directories (stripe factor 16) and the other has a stripe factor of 64. We used the Intel Paragon NX library to implement the I/O of the parallel pipeline STAP system. Subroutine `gopen()` was used to open CPI files globally with a non-collected I/O mode, `M_ASYNC`, because it offers better performance and causes less system overhead. In addition, we used asynchronous I/O function calls: `iread()` and `ireadoff()` in order to overlap I/O with the computation and communication.

The IBM AIX operating system provides a parallel file system called Parallel I/O File System (PIOFS) [3]. There are a total of 80 slices (striped directories) in the ANL PIOFS file system. IBM PIOFS supports existing C read, write, open and close functions. However, unlike the Paragon NX library, asynchronous parallel read/write subroutines are not supported on IBM PIOFS. The overall performance of the STAP pipeline system will be limited by the inability to overlap I/O operations with computation and communication.

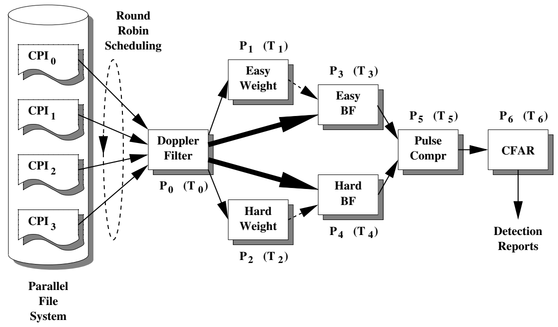

Figure 3. I/O task is embedded in the Doppler filter processing task.

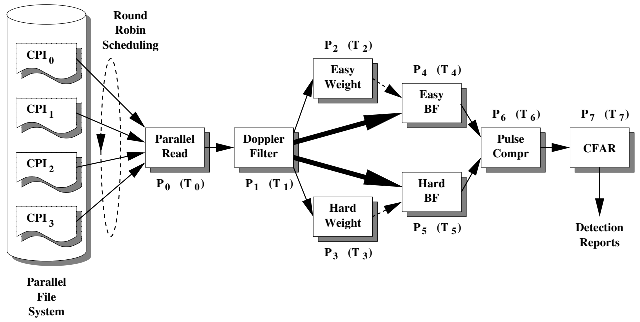

Figure 4. A separate I/O task for reading CPI data is added to the STAP pipeline system.

4. Design and implementation

A total of four CPI data sets stored as four files in the parallel file systems were used on both the Caltech Paragon and the ANL SP. During each of the following steps after the initialization, only nodes assigned to the first task perform read operations from the parallel file system. We assume that the radar writes its collected CPI data into these four files in a round-robin manner and, similarly, the STAP pipeline system reads the four files in a round-robin fashion but at times that are different from the times at which the radar writes. In this manner, the problem of data inconsistency for read/write operations between the radar and the pipeline system can be minimized.

All nodes allocated to the first task (the I/O nodes) of the pipeline read exclusive portions of each CPI file with proper offsets. The read length and file offset for all the read operations are set only during the STAP pipeline system's initialization and is not changed afterward. Therefore, in each of the following iterations, only one read function call is needed. On the Paragon, since asynchronous read subroutines were used, an additional subroutine waiting for the read's completion was also required in each iteration.

4.1. I/O task implementation

Two designs for the I/O task were implemented in the STAP pipeline system. The first one, shown in Figure 3, embeds the parallel I/O in the first task of the pipeline, i.e. in the Doppler filter processing task. The Doppler filter processing task now consists of three phases, reading CPI data from files, computation, and sending phases. The second I/O implementation creates a new task for reading CPI data and this task is added to the beginning of the pipeline. Figure 4 shows the structure of the overall pipeline system with this implementation. The only job of this I/O task is to read CPI data from the files and deliver it to the Doppler filter processing task.

5. Performance results

Performance results are given for the two I/O implementations on the parallel pipeline STAP system. For each implementation, parallel file systems on the Paragon and the SP were tested. On both machines, the stripe unit for the parallel file systems is 64K bytes. The size of each CPI data file is 8M bytes that results in 128 stripe units distributed across all stripe directories in all the parallel file systems.

5.1. I/O embedded in the first task

In the first I/O implementation on the Paragon, the Doppler filter processing task reads its input from CPI files using asynchronous read calls. Table 1 shows the timing results for this implementation on two PFS and a PIOFS file systems. For each parallel file system, three cases of node assignments to all tasks in the pipeline system are given, each doubles the number of nodes of another. Using the Paragon PFS with 16 stripe directories, the throughput scales well in the first two cases, but degrades when the total number of nodes goes up to 224. In this case, we observe that the timing results of the receive phase in the first task are relatively higher than the other two phases, the compute and send phases. The I/O operations for reading CPI data files here become a bottleneck for the pipeline system and this bottleneck forces the rest of the following tasks to wait for their input data from their previous tasks.

When using the PFS with 64 stripe directories, both throughput and latency showed linear speedups. In the first two cases with 56 and 112 nodes, the results of throughput and latency are approximately the same for both file systems with 16 and 64 stripe directories. However, in the case with 224 nodes, we observe that the I/O bottleneck is relieved by using 64 stripe directories. Therefore, the efficiency of I/O operations plays an important role in the overall performance of the pipeline system.

Table 1. Performance results with the I/O embedded in the Doppler filter processing task.

Paragon PFS stripe factor = 16

case 1: total number of nodes = 56 — Time in seconds

	nodes	recv	comp	send	total
Doppler filter	12	.0101	.2566	.0916	.3584
easy weight	3	.1317	.2214	.0002	.3534
hard weight	28	.0684	.2838	.0003	.3525
easy BF	3	.1451	.1921	.0003	.3375
hard BF	4	.1596	.1756	.0002	.3354
pulse compr	4	.1070	.1979	.0298	.3347
CFAR	2	.1983	.1361	-	.3343
throughput		2.9560			
latency		0.9804			

case 2: total number of nodes = 112 — Time in seconds

	nodes	recv	comp	send	total
Doppler filter	24	.0178	.1292	.0663	.2134
easy weight	6	.0856	.1110	.0002	.1968
hard weight	56	.0483	.1423	.0059	.1965
easy BF	6	.0939	.0958	.0003	.1901
hard BF	8	.0906	.0885	.0003	.1795
pulse compr	8	.0648	.0993	.0150	.1792
CFAR	4	.1107	.0683	-	.1790
throughput		5.4996			
latency		0.5171			

case 3: total number of nodes = 224 — Time in seconds

	nodes	recv	comp	send	total
Doppler filter	48	.0871	.0619	.0317	.1807
easy weight	12	.1056	.0557	.0002	.1616
hard weight	112	.0905	.0724	.0009	.1639
easy BF	12	.1080	.0482	.0003	.1565
hard BF	16	.1030	.0509	.0003	.1542
pulse compr	16	.0983	.0502	.0078	.1562
CFAR	8	.1217	.0343	-	.1561
throughput		6.2708			
latency		0.3292			

Paragon PFS stripe factor = 64

case 1: total number of nodes = 56

	nodes	recv	comp	send	total
Doppler filter	12	.0314	.2461	.0916	.3691
easy weight	3	.1262	.2216	.0002	.3480
hard weight	28	.0628	.2840	.0003	.3471
easy BF	3	.1397	.1921	.0003	.3321
hard BF	4	.1537	.1756	.0002	.3295
pulse compr	4	.1011	.1977	.0298	.3286
CFAR	2	.1920	.1363	-	.3282
throughput		3.0111			
latency		0.9787			

case 2: total number of nodes = 112

	nodes	recv	comp	send	total
Doppler filter	24	.0107	.1280	.0557	.1944
easy weight	6	.0787	.1111	.0020	.1917
hard weight	56	.0453	.1427	.0039	.1919
easy BF	6	.0860	.0959	.0003	.1823
hard BF	8	.0878	.0885	.0003	.1766
pulse compr	8	.0615	.0995	.0151	.1761
CFAR	4	.1077	.0682	-	.1759
throughput		5.6068			
latency		0.5143			

case 3: total number of nodes = 224

	nodes	recv	comp	send	total
Doppler filter	48	.0069	.0673	.0309	.1052
easy weight	12	.0510	.0559	.0002	.1071
hard weight	112	.0355	.0733	.0019	.1106
easy BF	12	.0526	.0483	.0003	.1013
hard BF	16	.0471	.0515	.0003	.0989
pulse compr	16	.0407	.0503	.0080	.0990
CFAR	8	.0642	.0343	-	.0985
throughput		10.0262			
latency		0.2871			

SP PIOFS stripe factor = 80

case 1: total number of nodes = 18

	nodes	recv	comp	send	total
Doppler filter	6	.1172	.0734	.1966	.3872
easy weight	1	.2717	.1070	.0001	.3788
hard weight	7	.1590	.2194	.0002	.3786
easy BF	1	.2927	.0829	.0001	.3757
hard BF	1	.2595	.1177	.0002	.3775
pulse compr	1	.2230	.1545	.0001	.3776
CFAR	1	.2941	.0828	-	.3770
throughput		2.6715			
latency		1.2353			

case 2: total number of nodes = 30

	nodes	recv	comp	send	total
Doppler filter	8	.1109	.0543	.1031	.2683
easy weight	1	.1471	.1045	.0002	.2518
hard weight	14	.1523	.1072	.0002	.2597
easy BF	2	.2189	.0412	.0001	.2602
hard BF	2	.1999	.0606	.0001	.2606
pulse compr	2	.1801	.0777	.0001	.2579
CFAR	1	.1801	.0801	-	.2602
throughput		3.8319			
latency		0.7810			

case 3: total number of nodes = 60

	nodes	recv	comp	send	total
Doppler filter	16	.1044	.0304	.0474	.1823
easy weight	2	.1314	.0547	.0001	.1862
hard weight	28	.1303	.0566	.0002	.1871
easy BF	4	.1571	.0219	.0002	.1792
hard BF	4	.1492	.0298	.0002	.1792
pulse compr	4	.1370	.0396	.0001	.1767
CFAR	2	.1399	.0403	-	.1802
throughput		5.5364			
latency		0.5004			

On the other hand, a linear speedup was obtained for the latency results. The I/O bottleneck problem does not affect the latency significantly. We can observe that in the case with 224 nodes, the latency of using 16 stripe directories is slightly greater than using 64 stripe directories. This can be explained by examining the throughput and latency equations, (1) and (2).Unlike the throughput depending on the maximum of the execution times, the latency is determined by the sum of the execution times of all the tasks except for the tasks with temporal dependency. Therefore, even though the execution time of the Doppler filter processing task is increased, the delay does not contribute much to the latency. Comparing the results of using two PFS file systems, the latency did not degrade significantly and still scaled well in the case with 224 nodes. Figure 5 shows the performance results of this I/O design in bar charts.

Detailed timing results for the IBM SP at ANL are also given in Table 1. The stripe factor of the PIOFS file system is 80. Because PIOFS does not provide asynchronous read/write subroutines, the I/O operations do not overlap with computation and communication in the Doppler filter processing task. Hence, the performance results for throughput and latency on the SP did not show the scalability as on the Paragon, even though the SP has faster CPUs.

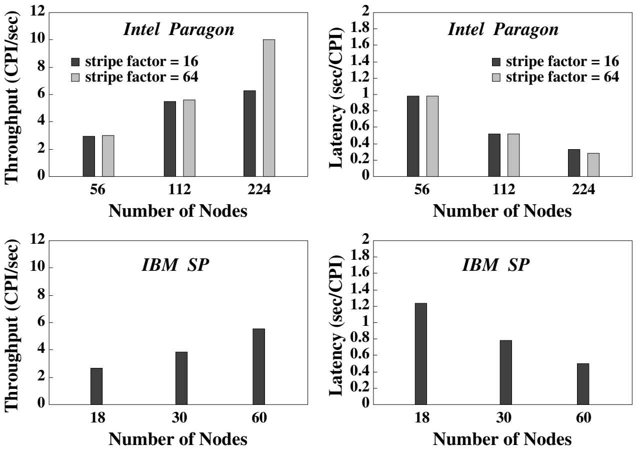

Figure 5. Results corresponds to Table 1.

5.2. A new I/O task

In the second I/O task implementation, a new task is added to the beginning of the pipeline. This new task only performs the operations of reading CPI files and distributing CPI data to its successor task, Doppler filter processing task. The STAP pipeline system then has a total of 8 tasks.

Table 2. Performance results with the I/O implemented as a separate task.

Paragon PFS stripe factor = 16

case 1: total number of nodes = 60 Time in seconds

	nodes	recv	comp	send	total
Parallel read	4	.0191	-	.3997	.4187
Doppler filter	12	.0122	.3240	.2375	.5738
easy weight	3	.2032	.2217	.0002	.4252
hard weight	28	.1390	.2846	.0003	.4239
easy BF	3	.2210	.1911	.0003	.4124
hard BF	4	.2327	.1753	.0003	.4083
pulse compr	4	.1800	.1977	.0295	.4072
CFAR	2	.2706	.1362	-	.4068
throughput		2.4127			
latency		1.9186			

case 2: total number of nodes = 120 Time in seconds

	nodes	recv	comp	send	total
Parallel read	8	.0559	-	.1604	.2163
Doppler filter	24	.0254	.1221	.0839	.2313
easy weight	6	.0920	.1110	.0004	.2034
hard weight	56	.0526	.1432	.0045	.2003
easy BF	6	.1003	.0960	.0003	.1966
hard BF	8	.0918	.0928	.0003	.1849
pulse compr	8	.0727	.0999	.0151	.1877
CFAR	4	.1185	.0683	-	.1867
throughput		5.3883			
latency		0.9226			

case 3: total number of nodes = 240 Time in seconds

	nodes	recv	comp	send	total
Parallel read	16	.1269	-	.0276	.1545
Doppler filter	48	.0833	.0463	.0245	.1541
easy weight	12	.0891	.0558	.0002	.1451
hard weight	112	.0749	.0724	.0004	.1477
easy BF	12	.0975	.0485	.0003	.1463
hard BF	16	.0924	.0516	.0003	.1443
pulse compr	16	.0869	.0502	.0077	.1448
CFAR	8	.1104	.0343	-	.1447
throughput		6.8438			
latency		0.3890			

Paragon PFS stripe factor = 64

case 1: total number of nodes = 60

nodes	recv	comp	send	total
4	.0628	-	.3391	.4019
12	.0085	.2670	.1755	.4510
3	.1425	.2217	.0002	.3645
28	.0763	.2847	.0003	.3613
3	.1621	.1914	.0003	.3537
4	.1740	.1759	.0002	.3501
4	.1213	.1980	.0296	.3489
2	.2125	.1362	-	.3488
	2.8234			
	1.7309			

case 2: total number of nodes = 120

nodes	recv	comp	send	total
8	.0362	-	.1685	.2047
24	.0280	.1084	.0786	.2151
6	.0816	.1111	.0024	.1951
56	.0461	.1438	.0003	.1903
6	.0914	.0959	.0003	.1877
8	.0891	.0908	.0003	.1802
8	.0672	.0999	.0151	.1822
4	.1131	.0683	-	.1815
	5.5262			
	0.9137			

case 3: total number of nodes = 240

nodes	recv	comp	send	total
16	.0171	-	.0617	.0788
48	.0073	.0502	.0290	.0864
12	.0503	.0558	.0002	.1063
112	.0305	.0724	.0029	.1057
12	.0491	.0489	.0004	.0984
16	.0417	.0540	.0004	.0961
16	.0393	.0502	.0078	.0973
8	.0629	.0343	-	.0972
	10.2111			
	0.5193			

SP PIOFS stripe factor = 80

case 1: total number of nodes = 20

nodes	recv	comp	send	total
2	.1787	-	.1413	.3200
6	.0045	.0724	.2548	.3316
1	.2269	.1047	.0001	.3317
7	.1165	.2150	.0013	.3329
1	.0641	.0822	.2082	.3545
1	.0416	.1179	.1874	.3469
1	.1459	.1538	.0656	.3653
1	.2926	.0801	-	.3727
	2.6670			
	2.6715			

case 2: total number of nodes = 34

nodes	recv	comp	send	total
4	.1230	-	.0594	.1823
8	.0264	.0549	.0913	.1726
1	.0639	.1043	.0001	.1683
14	.0598	.1090	.0003	.1692
2	.0576	.0415	.0814	.1805
2	.0593	.0596	.0579	.1768
2	.0278	.0784	.0803	.1864
1	.1092	.0804	-	.1896
	5.2819			
	1.2766			

case 3: total number of nodes = 68

nodes	recv	comp	send	total
8	.1100	-	.0185	.1285
16	.0455	.0283	.0631	.1369
2	.0901	.0535	.0001	.1437
28	.0839	.0554	.0001	.1395
4	.1158	.0208	.0035	.1401
4	.0813	.0483	.0089	.1385
4	.1008	.0391	.0054	.1453
2	.1074	.0404	-	.1478
	6.5063			
	0.6531			

Table 2 gives the performance results for this I/O design. Corresponding to Table 1, all tasks have the same numbers of nodes assigned, except for the I/O task. Similarly, the I/O bottleneck problem occurs when using the PFS with 16 stripe directories and the problem is solved by using the PFS with 64 stripe directories. The bar charts shown in Figure 6 represent the throughput and latency results of Table 2.

Comparing the two I/O designs, we observe that the throughput results are approximately the same . However, the latency results for the separate I/O task design are worse than the embedded one. This phenomenon can be explained by examining the new throughput and latency equations:

$$throughput_8 = \frac{1}{\max\limits_{0 \le i < 8} T_i} \tag{3}$$

and

$$latency_8 = T_0 + T_1 + \max(T_4, T_5) + T_6 + T_7, \tag{4}$$

where T_i is the execution time for the task i. The throughput of a pipeline system is determined by the task with the maximum execution time among all the tasks. Compared to Table 1, the throughput results have no significant change because the tasks with the maximum execution time are the same for every corresponding pair in all cases. The latency, on the other hand, is the sum of the execution times of all the tasks except for the tasks with temporal data dependency. In

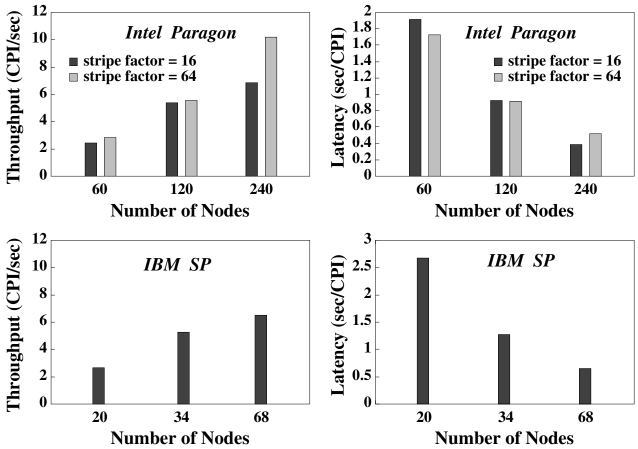

Figure 6. Results corresponds to Table 2.

Table 3. Performance results with pulse compression and CFAR tasks combined.

Paragon PFS stripe factor = 16

case 1: total number of nodes = 56 — Time in seconds

	nodes	recv	comp	send	total
Doppler filter	12	.0094	.2589	.0908	.3591
easy weight	3	.1307	.2230	.0002	.3540
hard weight	28	.0660	.2868	.0003	.3531
easy BF	3	.1449	.1930	.0003	.3382
hard BF	4	.1616	.1756	.0003	.3375
PC + CFAR	6	.1517	.1863	-	.3380
throughput		2.9243			
latency		0.7913			

case 2: total number of nodes = 112 — Time in seconds

	nodes	recv	comp	send	total
Doppler filter	24	.0194	.1294	.0656	.2145
easy weight	6	.0831	.1111	.0002	.1944
hard weight	56	.0468	.1427	.0046	.1940
easy BF	6	.0914	.0958	.0003	.1874
hard BF	8	.0892	.0887	.0004	.1784
PC + CFAR	12	.0869	.0935	-	.1804
throughput		5.5340			
latency		0.4221			

case 3: total number of nodes = 224 — Time in seconds

	nodes	recv	comp	send	total
Doppler filter	48	.0953	.0623	.0323	.1900
easy weight	12	.1056	.0558	.0003	.1617
hard weight	112	.0930	.0726	.0004	.1661
easy BF	12	.1116	.0484	.0003	.1603
hard BF	16	.1063	.0513	.0004	.1579
PC + CFAR	24	.1079	.0513	-	.1592
throughput		6.1478			
latency		0.2948			

Paragon PFS stripe factor = 64

case 1: total number of nodes = 56

nodes	recv	comp	send	total
12	.0319	.2485	.0915	.3718
3	.1265	.2218	.0002	.3485
28	.0631	.2839	.0003	.3473
3	.1400	.1921	.0003	.3324
4	.1533	.1756	.0003	.3292
6	.1449	.1860	-	.3309
3.0027				
0.7957				

case 2: total number of nodes = 112

nodes	recv	comp	send	total
24	.0104	.1301	.0528	.1933
6	.0774	.1111	.0002	.1887
56	.0438	.1427	.0022	.1886
6	.0853	.0959	.0003	.1815
8	.0869	.0886	.0004	.1759
12	.0838	.0936	-	.1773
5.6029				
0.4197				

case 3: total number of nodes = 224

nodes	recv	comp	send	total
48	.0071	.0676	.0306	.1054
12	.0522	.0559	.0002	.1083
112	.0347	.0730	.0031	.1108
12	.0533	.0482	.0004	.1018
16	.0481	.0512	.0003	.0997
24	.0489	.0514	-	.1003
9.8853				
0.2392				

SP PIOFS stripe factor = 80

case 1: total number of nodes = 18

nodes	recv	comp	send	total
6	.1320	.0728	.1894	.3942
1	.2844	.1023	.0001	.3868
7	.1738	.2131	.0002	.3870
1	.3039	.0823	.0001	.3862
1	.2677	.1182	.0002	.3862
2	.2683	.1194	-	.3877
2.5754				
0.9388				

case 2: total number of nodes = 30

nodes	recv	comp	send	total
8	.1105	.0550	.1055	.2710
1	.1711	.1026	.0002	.2739
14	.1570	.1077	.0002	.2649
2	.2225	.0417	.0001	.2644
2	.2051	.0608	.0002	.2661
3	.1878	.0793	-	.2671
3.7492				
0.6255				

case 3: total number of nodes = 60

nodes	recv	comp	send	total
16	.1044	.0279	.0462	.1786
2	.1350	.0515	.0002	.1867
28	.1238	.0568	.0002	.1808
4	.1582	.0210	.0002	.1794
4	.1485	.0300	.0003	.1787
6	.1397	.0414	-	.1810
5.5356				
0.4207				

the design with a separate I/O task, the latency contains one more term than the embedded I/O design. Therefore, the latency results become worse in this implementation.

6. Task combination

From the comparison of performance results for the two I/O task implementations, we notice that the structure of the STAP pipeline system can be reorganized to improve the latency. The first implementation that embeds I/O in the Doppler filter processing task can be viewed as combining the first two tasks of the second implementation that uses a separate task for I/O. As shown in Section 5.2, the first I/O implementation has a better latency performance, while the throughput results are approximately the same.

6.1. Improving latency

We investigate whether the latency can be further improved by combining multiple tasks of the pipeline into a single task. We consider Table 1 as an example and combine the last two tasks, the pulse compression and CFAR processing tasks, into a single task. In order to make a fair comparison, the number of nodes assigned to this single task is equal to the sum of the nodes assigned to the two original tasks and the total number of nodes allocated to the

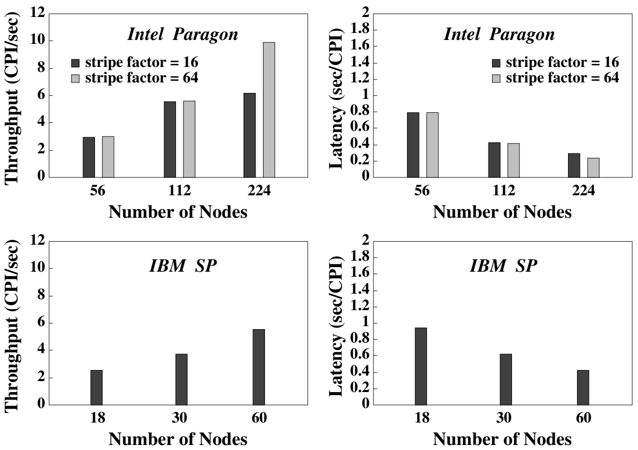

Figure 7. Results corresponds to Table 3.

whole pipeline system is kept the same. Table 3 gives the timing results corresponding to Table 1 with the same total number of nodes assigned to the pipeline system and Figure 7 shows the corresponding results in bar charts. The comparison of performance results of the STAP pipeline system with and without task combining is given in Figure 8. We observe that the latency improves for all cases on all parallel file systems when the last two tasks are combined.

Before task combination, the latency equation for the

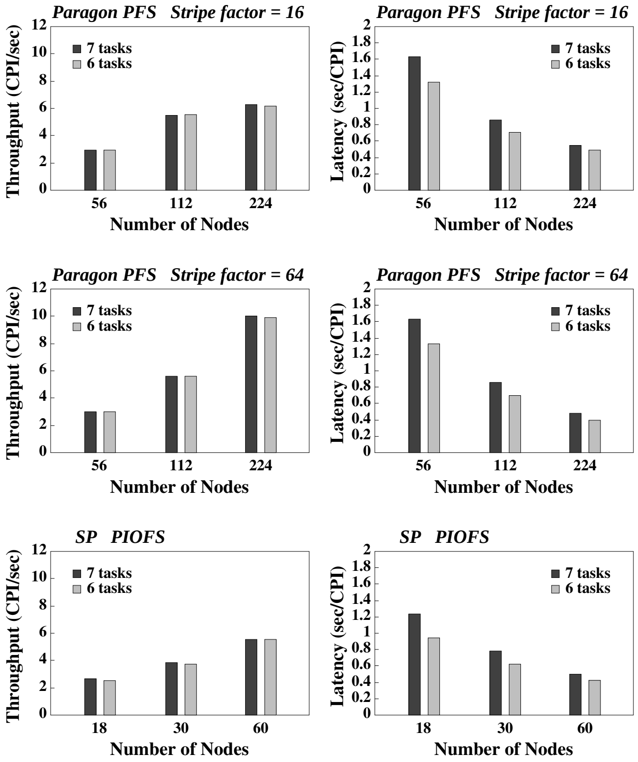

Figure 8. Performance comparison of the pipeline system with and without task combining.

STAP pipeline system with 7 tasks is

$$latency_7 = T_0 + \max(T_3, T_4) + T_5 + T_6. \quad (5)$$

Let W_5 and W_6 be the workloads for tasks 5 and 6, respectively. The execution times for task 5 and 6 are

$$T_5 = \frac{W_5}{P_5} + C_5 + V_5 \quad \text{and} \quad T_6 = \frac{W_6}{P_6} + C_6 + V_6 \quad (6)$$

where C_i and V_i represent the communication time and the other parallelization overhead for task i, respectively. Similarly, let T_{5+6} be the execution time of the task that combines tasks 5 and 6 running on $P_5 + P_6$ nodes:

$$T_{5+6} = \frac{W_5 + W_6}{P_5 + P_6} + C_{5+6} + V_{5+6}. \quad (7)$$

By subtracting Equation (6) from Equation (7), we have

$$\begin{aligned} T_{5+6} - (T_5 + T_6) &= \frac{W_5 + W_6}{P_5 + P_6} - \frac{W_5}{P_5} - \frac{W_6}{P_6} \\ &+ C_{5+6} - C_5 - C_6 \\ &+ V_{5+6} - V_5 - V_6 \end{aligned} \quad (8)$$

Table 4. Percentage of latency improvement when the Pulse compression and CFAR tasks are combined into a single task.

Paragon: PFS

# nodes	56	112	224
16 stripe dir	19.3%	18.4%	10.4%
64 stripe dir	18.7%	18.4%	16.7%

SP: PIOFS

# nodes	18	30	60
80 stripe dir	24.0%	19.9%	15.9%

where

$$\frac{W_5 + W_6}{P_5 + P_6} - \frac{W_5}{P_5} - \frac{W_6}{P_6} = \frac{-W_5 P_6^2 - W_6 P_5^2}{P_5 P_6 (P_5 + P_6)} < 0. \quad (9)$$

Communication for the combined task occurs only when receiving data from tasks 3 and 4. Prior to the task combination, the same communication takes place in the receive phase of task 5. The difference is the number of nodes used between the two tasks. Since $P_{5+6} > P_5$, the data size for each received message from tasks 3 and 4 to the combined task is smaller than that for task 5. Besides, in task 5, C_5 includes the communication cost of sending messages from task 5 to task 6 which does not occur in the combined task. Hence, we have

$$C_{5+6} < C_5. \quad (10)$$

The remaining overhead, V_i, is due to parallelization of task i. Since the operations in tasks 5 and 6 are sets of individual subroutines which require no communication within each single task, parallelization is carried out by evenly partitioning these subroutines among the nodes assigned. Due to this computational structure, the overhead for these two tasks becomes negligible compared to their communication costs. From Equations (8), (9), and (10) we can conclude that

$$T_{5+6} < T_5 + T_6. \quad (11)$$

Therefore, the new latency equation of the STAP pipeline system with the last two tasks combined becomes

$$latency_6 = T_0 + \max(T_3, T_4) + T_{5+6} < latency_7. \quad (12)$$

Combining the last two tasks, therefore, reduces the latency.

Table 4 gives the percentage of improvement in latency when the last two tasks are combined. These improvements were made without adding any extra nodes to the pipeline system. We observe that the percentage decreases as the number of nodes goes up. Normally, scalability of the parallelization tends to decrease when more processors are used.

This also explains the trend for the percentage improvement shown in Table 4. Notice that the tasks that can be combined to improve the latency do not include tasks with temporal data dependency. It is because only those tasks with spatial data dependency contribute to the latency.

6.2. Improving throughput

The throughput results, on the other hand, do not change significantly when the two tasks are combined. This is because the task with the maximum execution time among all the tasks is still the maximum in the new pipeline system. Assuming that T_{max} is the maximum execution time before task combination:

$$T_{max} = \max_{0 \leq i < 7} T_i \geq max(T_5, T_6)$$

From Equations (6) and (7), the execution time of the new combined task becomes

$$\begin{aligned} T_{5+6} &\approx \frac{P_5 T_5 + P_6 T_6}{P_5 + P_6} \\ &\leq \frac{P_5\ max(T_5, T_6) + P_6\ max(T_5, T_6)}{P_5 + P_6} \\ &= max(T_5, T_6) \end{aligned} \quad (13)$$

and the new maximum execution time

$$\begin{aligned} T'_{max} &= max(T_0, T_1, T_2, T_3, T_4, T_{5+6}) \\ &\leq max(T_0, T_1, T_2, T_3, T_4, T_5, T_6) \ = \ T_{max}. \end{aligned}$$

Therefore, the throughput will not decrease after task combination because

$$throughput_6 = \frac{1}{T'_{max}} \geq \frac{1}{T_{max}} = throughput_7. \quad (14)$$

Both latency and throughput can be improved simultaneously when one of the combined tasks determines the throughput of the pipeline system. Suppose that either task 5 or task 6 has the maximum execution time among all the 7 tasks in the STAP pipeline system, that is,

$$T_{max} = max(T_5, T_6) > \max_{0 \leq i \leq 4} T_i. \quad (15)$$

Notice that none of these two tasks has temporal data dependency. From Equation (12), we have latency improvement when tasks 5 and 6 are combined. From Equations (14) and (15), the throughput is increased. The reduction of execution time of both tasks 5 and 6 contributes to the latency as well as to the throughput. Therefore, not only the throughput can be increased, but the latency can be also reduced. Note that in our experiment results shown in the previous section, the task with the maximum execution time is neither task 5 nor task 6, that is, $T_{max} > \max(T_5, T_6)$.

7. Conclusions

In this work, we studied the effects of parallel I/O implementation for a modified PRI-staggered post-Doppler STAP algorithm. The parallel pipeline STAP system was run portably on Intel Paragon and IBM SP using the existing parallel file systems. On the Paragon, we found that a pipeline bottleneck can result when using a parallel file system with a relatively smaller stripe factor. With a larger stripe factor, a parallel file system can deliver higher efficiency of I/O operations and, therefore, improve the throughput performance.

This paper presented two I/O designs which are incorporated into the parallel pipeline STAP system. One embedded I/O in the original pipeline and the other used a separate I/O task. By comparing the results of these designs, we found that the task structure of the pipeline can be reorganized to further improve the latency. Without adding any compute nodes, we obtained performance improvement in the latency when the last two tasks were combined. We also analyzed the possibility of further improvement by examining the throughput and latency equations. The performance results demonstrate that the parallel pipeline STAP system scaled well even with a more complicated I/O implementation.

8. Acknowledgments

This work was supported by Air Force Materials Command under contract F30602-97-C-0026. We acknowledge the use of the Intel Paragon at California Institute of Technology and the IBM SP at Argonne National Laboratory.

References

[1] R. Brown and R. Linderman. Algorithm Development for an Airborne Real-Time STAP Demonstration. In *Proceedings of the IEEE National Radar Conference*, 1997.

[2] A. Choudhary, W. Liao, D. Weiner, P. Varshney, R. Linderman, and M. Linderman. Design, Implementation and Evaluation of Parallel Pipelined STAP on Parallel Computers. *International Parallel Processing Symposium*, 1998.

[3] IBM Corp. *IBM AIX Parallel I/O File System: Installation, Administration, and Use*, October 1996.

[4] Intel Corporation. *Paragon System User's Guide*, Apr. 1996.

[5] W. Liao, A. Choudhary, D. Weiner, and P. Varshney. Multi-Threaded Design and Implementation of Parallel Pipelined STAP on Parallel Computers with SMP Nodes. *International Parallel Processing Symposium*, 1999.

[6] M. Linderman and R. Linderman. Real-Time STAP Demonstration on an Embedded High Performance Computer. In *Proceedings of the IEEE National Radar Conference*, 1997.

[7] M. Little and W. Berry. Real-Time Multi-Channel Airborne Radar Measurements. In *Proceedings of the IEEE National Radar Conference*, 1997.