DRAFT

Implementation and Scalability of Fortran 90D Intrinsic Functions on Distributed Memory Machines *

Ishfaq Ahmad   Rajesh Bordawekar   Zeki Bozkus
Alok Choudhary   Geoffrey Fox   Kanchana Parasuram
Ravi Ponnusamy   Sanjay Ranka   Rajeev Thakur

Technical Report SCCS-256
Northeast Parallel Architectures Center
111 College Place
Syracuse, NY 13244-4100

Abstract

We are developing a Fortran 90D compiler, which converts Fortran 90D code into Fortran 77 node programs for distributed memory machines. This paper presents the performance results of Fortran 90D intrinsic functions on Intel iPSC/860 and iPSC/2 hypercubes. We discuss the implementation of these intrinsic functions and show that our implementations are scalable.

*This work was sponsored by DARPA under contract # DABT63-91-C-0028. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

1 Introduction

It is widely recognized that massively parallel MIMD distributed memory machines can provide enormous computing power. But this power has not yet been fully harnessed because of the difficulty in programming these machines and the lack of portability between them. Hence it is necessary to have a machine-independent parallel programming model for distributed memory machines.

Fortran 90 is the most popular parallel programming language for SIMD machines. But, it is not just suitable for SIMD machines; with some extensions, it can also represent a class of problems called *synchronous* problems [4]. Fortran 90D, a version of Fortran 90 enhanced with a rich set of data decomposition specifications, is a language designed for this purpose. The data decomposition specifications indicate how arrays should be aligned with respect to one another, both within and across array dimensions, and also how arrays should be distributed among the processors of the parallel machine. These specifications in Fortran 90D are the same as in Fortran 77D [5, 7]. The main advantage of Fortran 90D is that it uses high level data structures explicitly (as arrays) and so the problem architecture is clear and not hidden in values of pointers and DO loop indices [3]. Compilers can effectively map Fortran 90D into all parallel architectures suitable for synchronous problems, including MIMD & SIMD parallel machines, systolic arrays and heterogeneous networks [3]. Compiler methods for converting Fortran 77D programs into distributed memory node programs are discussed in [8, 9]. We are developing a Fortran 90D compiler, which converts Fortran 90D code into Fortran 77 plus message passing node programs for a distributed memory machine [12].

Fortran 90D has many intrinsic functions. It is necessary to build a library of intrinsic functions which can be called from the node programs of a distributed memory machine. This paper discusses the implementation and scalability of several of these intrinsic functions. In order that the intrinsic functions are portable, we have implemented them using EXPRESS, a portable parallel programming environment developed by Parasoft Corp. [10, 11].

2 EXPRESS

EXPRESS provides routines for interprocessor communication as well as tools for debugging and performance analysis. A detailed discussion of the functionality of EXPRESS is given in [1]. The most important feature of EXPRESS is that it is portable. Programs written using EXPRESS can be run without any modifications on a number of machines such as NCUBE, Intel iPSC/2 and iPSC/860 hypercubes, Intel Touchstone Delta, transputer arrays, BBN Butterfly and also on networks of workstations. The languages supported by EXPRESS are FORTRAN and C.

One feature of EXPRESS which we have extensively used is a set of function calls collectively known as KXGRID. Their purpose is to take a user specification of a problem domain and perform a mapping to the underlying processor topology. EXPRESS then provides the physical processor numbers that may be required for use in communication calls. The user does not need to know the exact location of the processes or which nodes they have to communicate with – all this is handled transparently. The architecture of the parallel machine is effectively hidden from the user, which makes programming easier and portable. The user has to specify the dimensionality of the problem and the number of processors to be assigned to each dimension. EXPRESS then creates a *virtual grid* and maps the physical processor numbers into coordinates in the grid. The user can then assume that the processors are configured as a grid and use EXPRESS routines to find the physical number of a processor ¿from its location in the grid and vice-versa.

3 Fortran 90D Intrinsic Functions

The intrinsic functions that we have implemented fall into four main categories :-

- *Array Reduction Functions*: ALL, ANY, COUNT, MAXVAL, MINVAL, PRODUCT, SUM.
- *Array Manipulation Functions*: CSHIFT, EOSHIFT, TRANSPOSE.
- *Array Location Functions*: MAXLOC, MINLOC.
- *Array Construction Functions*: SPREAD.

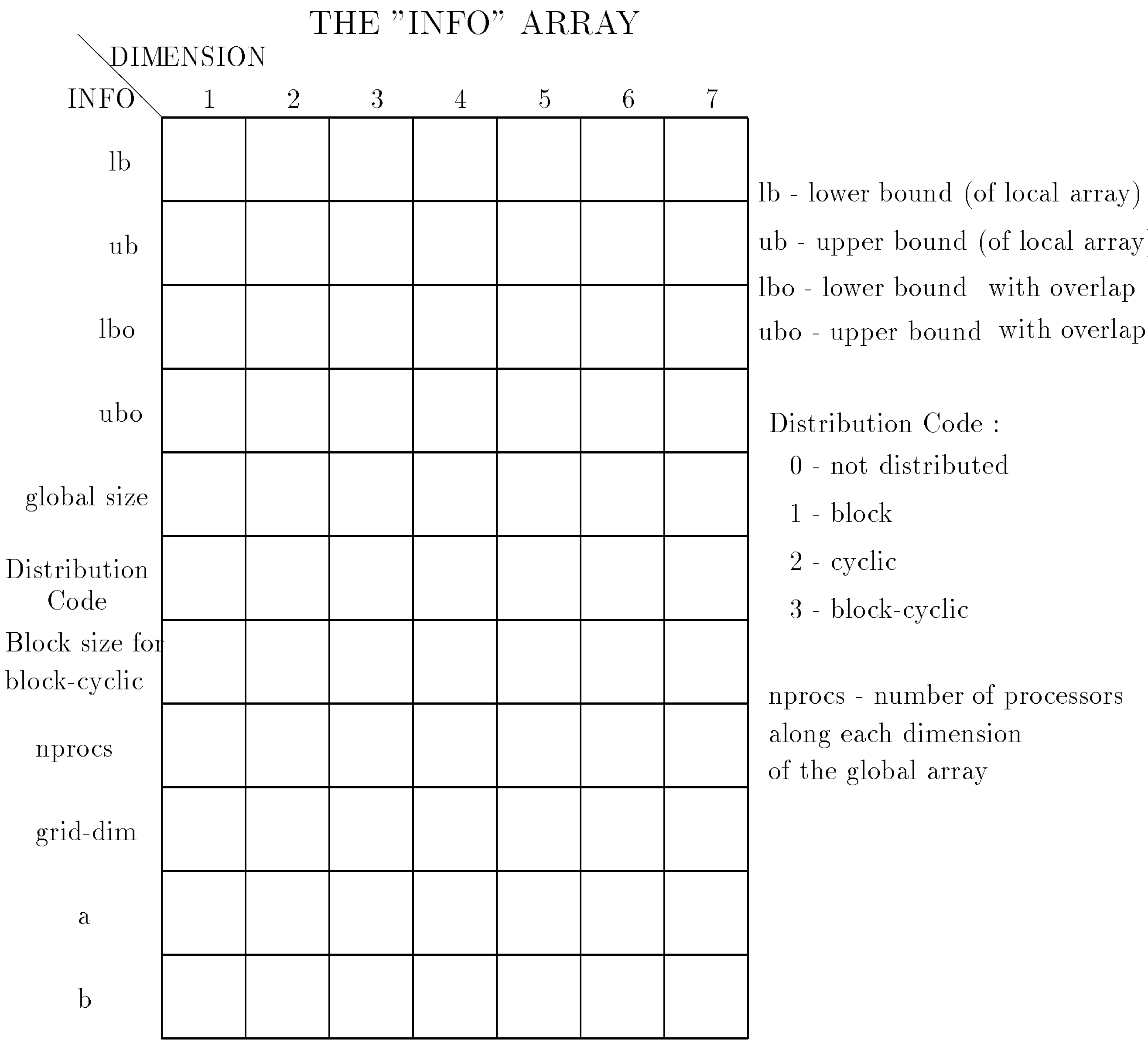

Figure 1: Array Specifications Passed to Intrinsic Functions

- *Vector and Matrix Multiplication Functions*: DOT_PRODUCT, MATMUL.

For each of these functions, we have written Fortran 77 routines which can be called from the node programs of a distributed memory machine. The Fortran 90D compiler will detect calls to intrinsic functions in the Fortran 90D program and replace them with calls to these routines. When an array is passed as an argument to an intrinsic function, it is also necessary to provide some other information such as its size, distribution among the nodes of the distributed memory machine etc. All this information is stored in an array "INFO" and passed as another argument to the intrinsic function. The contents of the "INFO" array are shown in figure 1. Rows 1 and 2 contain the lower and upper bounds of the local array (excluding overlap area) in each dimension. The lower and upper bounds in each dimension including overlap area are stored in rows 3 and 4. The number of elements in each dimension of the global array is given in row 5. Row 6 contains information regarding the distribution of the array. If the distribution is block-cyclic, it is also necessary to specify the block size. This is given in row 7. If none of the dimensions have block-cyclic distribution, this row can be ignored. Row 8 specifies the number of processors assigned to each dimension of the array. Rows 9, 10 and 11 contain array alignment information. Row 9 indicates the dimension of the grid along which each dimension of the array is aligned. If the alignment statement is of the form ALIGN X(i) with Y(a*i+b) then the value of 'a' is stored in row 10. If there is any other form of alignment, 0 is stored in row 10 and that alignment is declared as a function which can be called from the intrinsic. The value of 'b' defined above is stored in row 11

We have written separate routines for one-and two dimensional arrays and also in cases where some of the arguments are optional, because Fortran 77 does not support optional arguments. The compiler has to call the appropriate routine depending on the number of dimensions of the array and the optional arguments supplied.

- *Syntax*: **ALL(MASK, DIM), ANY(MASK, DIM), COUNT(MASK, DIM)**
- *Optional Arguments*: DIM
- *Description (ALL)*: Determines whether all values are true in MASK along dimension DIM.
- *Description (ANY)*: Determines whether any value is true in MASK along dimension DIM.
- *Description (COUNT)*: Count the number of true elemnts of MASK along dimension DIM.

  1. MASK: must be of type logical and must be conformable with ARRAY.
  2. DIM (optional): must be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY.

For ALL and ANY, the result is of type logical with the same kind of type parameters as MASK. For COUNT, the result type is integer. It is scalar if DIM is absent or ARRAY has rank one; otherwise the result is an array of rank $n-1$ and of shape $(d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of ARRAY.

Figure 2: FORTRAN 90D Specification for ALL, ANY and COUNT

4 Array Reduction Functions

4.1 ALL

The Fortran 90D specification for ALL is given in figure 2. For a one-dimensional array, each processor performs an AND operation on the local array and generates a local result. Then all processors perform a global AND operation using the EXPRESS routine KXCOMB to find the global AND of the local variables; the result of the global operation is left in each participating processor. In the case of two-dimensional arrays, if the DIM argument is not specified, the problem is essentially the same as that for a one-dimensional array. If DIM is specified, the result is a one-dimensional array. Each processor performs the AND operation along each row (if reduction dimension is 2) or column (if reduction dimension is 1) of the local array. The result of this local computation is that each processor generates a vector of TRUE and FALSE values. Depending upon the reduction dimension and processor grid configuration, a global operation may be required to take a global AND across the vectors generated by each processor. The result of this global operation is itself a vector which is left in each participating processor. A global operation can be performed in this case because the EXPRESS routine KXCOMB allows us to specify the list of processors participating in the global operation. It is not necessary that all processors have to participate in the global operation. The timings for different array sizes and processor configurations are given in tables 1, 2, 3 and 4.

Table 1: ALL, 1 dim. array, any decomposition

Array Size	Processors	iPSC/860 time (ms)			iPSC/2 time (ms)		
		t_{calc}	t_{comm}	t_{total}	t_{calc}	t_{comm}	t_{total}
64	1	0.095	0.000	0.095	0.149	0.149	0.290
	2	0.082	0.313	0.395	0.088	0.971	1.509
	4	0.056	0.413	0.469	0.058	2.794	2.852
	8	0.041	0.560	0.601	0.051	4.183	4.234
	16	0.027	0.977	1.004	0.048	5.622	5.670
	32	0.019	1.035	1.054	0.034	7.084	7.158
256	1	0.325	0.000	0.325	0.539	0..063	0.602
	2	0.313	0.313	0.625	0.276	0.802	1.708
	4	0.113	0.313	0.426	0.158	2.795	2.953
	8	0.078	0.625	0.703	0.096	4.142	4.238
	16	0.039	1.016	1.055	0.069	5.611	5.680
	32	0.026	2.016	2.042	0.084	7.057	7.141
1k	1	0.625	0.000	0.625	2.078	0.070	2.148
	2	0.313	0.313	0.312	1.050	1.440	2.490
	4	0.156	0.469	0.625	0.542	2.800	3.342
	8	0.078	0.703	0.781	0.285	4.199	4.484
	16	0.156	0.898	1.055	0.179	5.643	5.822
	32	0.098	1.094	1.191	0.128	7.061	7.189
4k	1	1.250	0.625	1.875	8.258	0.062	8.320
	2	0.625	0.313	0.938	4.141	3.465	5.609
	4	0.156	0.781	0.938	2.083	3.849	4.932
	8	0.234	0.703	0.938	1.059	4.220	5.279
	16	0.195	0.859	1.055	0.560	5.630	6.190
	32	0.098	1.113	1.211	0.329	7.082	7.411
16k	1	5.838	0.156	5.994	33.038	0.123	33.161
	2	3.125	0.313	3.438	16.529	1.715	18.244
	4	1.406	0.625	2.031	8.260	3.034	11.294
	8	0.781	0.703	1.484	4.152	4.281	8.433
	16	0.508	0.781	1.289	2.099	5.691	7.790
	32	0.234	1.094	1.328	1.088	7.160	8.248
64k	1	22.500	0.156	22.656	146.797	0.082	146.879
	2	11.875	0.000	11.875	73.423	1.617	75.040
	4	5.781	0.469	6.250	33.042	3.462	36.504
	8	2.969	0.781	3.750	16.539	4.576	21.115
	16	1.602	0.898	2.500	8.270	5.891	14.161
	32	0.801	1.094	1.895	4.178	7.172	11.350
256k	1	90.625	0.000	90.625	587.212	0.375	587.300
	2	45.625	0.000	45.625	293.624	1.636	295.260
	4	22.813	0.625	23.438	146.811	3.073	149.884
	8	11.484	0.703	12.188	73.434	4.455	77.889
	16	5.938	0.781	6.719	33.039	5.252	39.391
	32	3.008	1.055	4.063	16.562	7.473	24.035

Table 2: ALL, 2 dim. array, reduction to scalar, any decomposition

Array Size	Processors	iPSC/860 time (ms)			iPSC/2 time (ms)		
		t_{calc}	t_{comm}	t_{total}	t_{calc}	t_{comm}	t_{total}
16x16	1	0.313	0.000	0.313	1.015	0.017	1.032
	2	0.234	0.273	0.508	0.537	1.416	1.952
	4	0.204	0.469	0.673	0.306	2.744	3.051
	8	0.186	0.674	0.859	0.188	4.157	4.345
	16	0.176	0.879	1.055	0.136	5.538	5.674
	32	0.112	1.069	1.182	0.083	6.954	7.037
32x32	1	0.781	0.000	0.781	3.922	0.002	3.924
	2	0.469	0.273	0.742	2.005	1.430	3.434
	4	0.313	0.488	0.801	1.061	2.789	3.850
	8	0.254	0.674	0.928	0.585	4.149	4.734
	16	0.225	0.869	1.094	0.352	5.542	5.894
	32	0.212	1.067	1.279	0.243	6.960	7.203
64x64	1	2.500	0.000	2.500	15.649	0.058	15.708
	2	1.328	0.273	1.602	7.948	1.584	9.532
	4	0.781	0.430	1.211	4.075	2.859	6.934
	8	0.498	0.674	1.172	2.140	4.195	6.335
	16	0.342	0.894	1.235	1.175	5.582	6.756
	32	0.273	1.079	1.353	0.689	6.952	7.641
128x128	1	9.375	0.000	9.375	63.199	0.019	63.318
	2	4.766	0.313	5.078	31.815	1.682	33.497
	4	2.500	0.488	2.988	16.087	3.035	19.122
	8	1.377	0.664	2.041	8.256	4.385	12.641
	16	0.811	0.889	1.699	4.327	5.651	9.977
	32	0.515	1.108	1.624	2.368	6.993	9.361
256x256	1	36.641	0.000	36.641	277.757	0.066	277.823
	2	18.477	0.273	18.750	136.235	1.703	137.937
	4	9.375	0.469	9.844	64.816	3.312	68.129
	8	4.844	0.693	5.537	32.701	4.467	37.168
	16	2.559	0.913	3.472	16.750	5.806	22.556
	32	1.416	1.104	2.520	8.757	7.181	15.938
512x512	1	145.469	0.000	145.469	1162.460	0.057	1162.517
	2	73.008	0.273	73.281	581.671	1.674	583.345
	4	36.719	0.508	37.227	289.704	3.147	292.851
	8	18.564	0.674	19.238	139.779	4.623	144.402
	16	9.487	0.913	10.400	66.420	6.290	72.710
	32	4.971	1.099	6.069	33.970	7.258	41.229
1kx1k	1	580.625	0.000	580.625	—	—	—
	2	290.703	0.273	290.977	2531.377	1.884	2533.261
	4	145.664	0.527	146.191	1264.844	3.319	1268.163
	8	73.164	0.674	73.838	631.298	5.283	636.581
	16	36.934	0.928	37.861	310.402	6.381	316.782
	32	18.818	1.104	19.922	144.727	9.022	153.749

Table 3: ALL, 2 dim. array, reduction along a dimension, array size = 256×256

Processors	Decomposition	Reduction Along	iPSC/860 time (ms)			iPSC/2 time (ms)		
			t_{calc}	t_{comm}	t_{total}	t_{calc}	t_{comm}	t_{total}
4	(any decomp, X)	row	10.237	3.676	13.913	76.939	16.232	93.171
		col	9.704	0.071	9.775	77.652	0.397	78.04
	(X, any decomp)	row	10.115	0.064	10.180	73.863	0.365	74.228
		col	9.807	3.642	13.448	76.905	14.864	91.769
	(any decomp, any decomp)							
	2×2	row	10.157	1.220	11.377	75.205	5.583	80.788
		col	9.738	1.215	10.953	77.054	5.500	82.554
8	(any decomp, X)	row	5.287	5.456	10.744	39.265	21.819	61.084
		col	4.943	0.069	5.012	38.861	0.297	39.158
	(X, any decomp)	row	5.149	0.059	5.208	36.311	0.292	36.603
		col	5.066	5.437	10.503	38.783	21.750	60.533
	(any decomp, any decomp)							
	4×2	row	5.209	2.325	7.533	38.004	9.542	47.546
		col	4.961	0.895	5.856	38.564	3.908	42.47
	2×4	row	5.169	0.895	6.064	37.160	3.802	40.962
		col	4.995	2.322	7.316	38.475	9.534	48.008
16	(any decomp, X)	row	2.813	7.276	10.088	20.469	29.708	50.176
		col	2.564	0.067	2.631	19.464	0.297	19.761
	(X, any decomp)	row	2.665	0.059	2.724	17.950	0.301	18.252
		col	2.695	7.262	9.957	19.955	29.599	49.554
	(any decomp, any decomp)							
	4×4	row	2.694	1.695	4.389	18.802	7.075	25.876
		col	2.589	1.703	4.292	19.286	7.224	26.510
	8×2	row	2.734	3.442	6.177	19.427	14.253	33.680
		col	2.572	0.802	3.374	19.316	3.232	22.548
	2×8	row	2.674	0.830	3.505	18.292	3.256	21.548
		col	2.624	3.458	6.082	19.429	14.362	33.791
32	(any decomp, X)	row	1.575	9.116	10.691	11.047	37.312	48.359
		col	1.374	0.066	1.440	9.772	0.265	10.037
	(X, any decomp)	row	1.423	0.059	1.482	8.885	0.268	9.153
		col	1.509	9.105	10.614	10.677	37.314	47.991
	(any decomp, any decomp)							
	16×2	row	1.497	4.614	6.111	10.116	19.341	29.457
		col	1.378	0.344	1.722	9.689	2.030	11.719
	2×16	row	1.428	0.341	1.769	9.059	2.023	11.081
		col	1.438	4.610	6.049	10.008	19.335	29.343
	8×4	row	1.457	2.518	3.975	9.608	10.546	20.154
		col	1.385	1.484	2.869	9.674	5.695	15.369
	4×8	row	1.437	1.500	2.938	9.269	5.726	14.995
		col	1.403	2.526	3.929	9.743	10.567	20.310

Table 4: ALL, 2 dim. array, reduction along a dimension, array size = 512×512

Processors	Decomposition	Reduction Along	iPSC/860 time (ms)			iPSC/2 time (ms)		
			t_{calc}	t_{comm}	t_{total}	t_{calc}	t_{comm}	t_{total}
4	(any decomp, X)	row	40.040	6.754	46.794	335.621	29.026	364.647
		col	38.155	0.074	38.229	326.044	0.336	326.380
	(X, any decomp)	row	39.805	0.065	39.870	313.776	0.322	314.097
		col	38.364	6.744	45.108	320.851	27.979	348.830
	(any decomp, any decomp)							
	2×2	row	39.887	1.882	41.768	322.807	7.651	330.458
		col	38.225	1.864	40.089	322.767	7.342	330.109
8	(any decomp, X)	row	20.277	10.384	30.661	167.180	44.101	211.28
		col	19.177	0.074	19.251	163.074	0.305	163.379
	(X, any decomp)	row	20.003	0.063	20.066	153.338	0.342	153.680
		col	19.422	10.355	29.776	160.657	42.717	203.374
	(any decomp, any decomp)							
	4×2	row	20.122	3.669	23.791	161.154	14.986	176.140
		col	19.212	1.224	20.437	161.418	4.901	166.319
	2×4	row	20.042	1.222	21.264	156.563	5.192	161.755
		col	19.282	3.662	22.944	160.471	14.451	174.922
16	(any decomp, X)	row	10.391	13.992	24.383	79.608	60.366	139.974
		col	9.686	0.072	9.758	78.554	0.402	78.957
	(X, any decomp)	row	10.097	0.060	10.157	72.355	0.398	72.752
		col	9.949	14.011	23.960	77.488	58.700	136.187
	(any decomp, any decomp)							
	4×4	row	10.157	2.323	12.480	75.220	10.259	85.478
		col	9.738	2.334	12.072	77.060	10.037	87.097
	8×2	row	10.236	5.475	15.711	76.955	23.029	99.984
		col	9.703	0.905	10.609	77.659	4.372	82.031
	2×8	row	10.117	0.903	11.019	73.861	4.209	78.070
		col	9.807	5.478	15.285	76.892	22.943	99.835
32	(any decomp, X)	row	5.442	17.609	23.051	41.345	73.240	114.585
		col	4.935	0.070	5.004	39.318	0.316	39.634
	(X, any decomp)	row	5.140	0.059	5.199	35.755	0.312	36.066
		col	5.207	17.544	22.751	39.847	73.197	113.044
	(any decomp, any decomp)							
	16×2	row	5.287	7.273	12.560	39.263	29.711	68.974
		col	4.944	0.840	5.783	38.868	3.319	42.187
	2×16	row	5.149	0.827	5.976	36.310	3.361	39.671
		col	5.065	7.270	12.335	38.788	29.658	68.446
	8×4	row	5.209	3.454	8.663	38.013	14.242	52.255
		col	4.961	1.713	6.673	38.576	7.187	45.763
	4×8	row	5.168	1.697	6.866	37.163	7.110	44.273
		col	4.994	3.483	8.477	38.497	14.200	52.697

4.2 ANY

The Fortran 90D specification for ANY is given in figure 2. The implementation has been done in the same manner as in the case of ALL. The only difference is that each processor performs an OR operation instead of AND. Similarly in the global operation OR operation is performed instead of AND. The timings for different array sizes and processor configurations are given in tables 5, 6, 7 and 8.

Table 5: ANY, 1 dim. array, any decomposition

Array Size	Processors	iPSC/860 time (ms)			iPSC/2 time (ms)		
		t_{calc}	t_{comm}	t_{total}	t_{calc}	t_{comm}	t_{total}
64	1	0.125	0.000	0.125	0.150	0.062	0.212
	2	0.079	0.117	0.196	0.090	1.432	1.532
	4	0.049	0.469	0.508	0.054	2.830	2.884
	8	0.034	0.625	0.659	0.051	4.183	4.234
	16	0.023	0.742	0.765	0.038	5.632	5.670
	32	0.016	0.859	1.075	0.028	7.089	7.107
256	1	0.325	0.000	0.325	0.540	0.060	0.600
	2	0.156	0.156	0.312	0.280	1.441	1.721
	4	0.078	0.469	0.547	0.160	2.810	2.970
	8	0.048	0.625	0.673	0.096	4.142	4.238
	16	0.027	0.898	0.925	0.073	5.637	5.710
	32	0.018	1.035	1.053	0.089	7.081	7.170
1k	1	0.625	0.000	0.625	2.080	0.058	2.138
	2	0.356	0.078	0.434	1.062	1.448	2.510
	4	0.156	0.469	0.625	0.540	2.829	3.369
	8	0.078	0.625	0.703	0.285	4.199	4.484
	16	0.057	0.898	1.955	0.164	5.662	5.822
	32	0.038	1.113	1.151	0.130	7.109	7.239
4k	1	1.875	0.000	1.875	8.261	0.060	8.321
	2	0.938	0.313	1.250	4.139	1.479	5.618
	4	0.469	0.625	1.094	2.090	2.857	4.947
	8	0.156	0.703	0.859	1.059	4.220	5.279
	16	0.117	0.781	0.898	0.554	5.684	6.238
	32	0.039	0.996	1.035	0.318	0.093	7.441
16k	1	6.250	0.000	6.250	33.018	0.120	33.138
	2	3.125	0.156	3.281	16.511	1.661	18.270
	4	1.719	0.256	1.975	8.257	3.076	11.333
	8	0.781	0.703	1.484	4.152	4.281	8.433
	16	0.469	0.820	1.289	2.094	5.742	7.836
	32	0.332	0.977	1.309	1.088	8.160	8.248
64k	1	23.125	0.000	23.125	146.802	0.089	146.891
	2	11.562	0.156	11.718	73.430	1.701	75.131
	4	6.094	0.256	6.350	33.022	3.470	36.492
	8	2.969	0.703	3.672	16.539	5.576	21.115
	16	1.445	0.859	2.305	8.291	5.911	14.202
	32	0.820	1.055	1.875	4.173	7.197	11.370
256k	1	92.188	0.000	92.188	587.192	0.086	587.278
	2	46.250	0.234	46.484	293.611	1.551	295.262
	4	23.125	0.313	23.438	146.802	3.173	149.975
	8	11.563	0.781	12.344	73.434	4.455	77.889
	16	5.781	0.977	6.758	33.029	6.340	39.381
	32	2.949	1.172	4.121	16.532	7.493	24.035

Table 6: ANY, 2 dim. array, reduction to scalar, any decomposition

Array Size	Processors	iPSC/860 time (ms)			iPSC/2 time (ms)		
		t_{calc}	t_{comm}	t_{total}	t_{calc}	t_{comm}	t_{total}
16x16	1	0.313	0.000	0.313	1.016	0.025	1.041
	2	0.234	0.234	0.469	0.553	1.394	1.947
	4	0.204	0.449	0.653	0.298	2.770	3.068
	8	0.156	0.654	0.810	0.192	4.151	4.343
	16	0.106	0.869	1.975	0.130	5.566	5.697
	32	0.057	1.040	1.097	0.080	6.975	7.055
32x32	1	0.781	0.000	0.781	3.875	0.044	3.919
	2	0.469	0.234	0.703	2.000	1.457	3.458
	4	0.352	0.410	0.762	1.071	2.767	3.838
	8	0.273	0.635	0.908	0.596	4.136	4.732
	16	0.220	0.830	1.050	0.362	5.558	5.920
	32	0.190	1.052	1.243	0.241	6.960	7.201
64x64	1	2.578	0.000	2.578	15.422	0.064	15.486
	2	1.406	0.234	1.641	7.813	1.612	9.425
	4	0.801	0.430	1.230	4.027	2.884	6.912
	8	0.488	0.645	1.133	2.137	4.176	6.313
	16	0.342	0.864	1.206	1.189	5.551	6.741
	32	0.286	1.035	1.321	0.716	6.979	7.695
128x128	1	9.766	0.000	9.766	62.447	0.124	62.571
	2	4.922	0.234	5.156	31.305	1.671	32.976
	4	2.598	0.449	3.047	15.888	3.041	18.930
	8	1.406	0.654	2.061	8.161	4.389	12.550
	16	0.845	0.825	1.670	4.304	5.704	10.008
	32	0.535	1.047	1.58	2.382	7.035	9.417
256x256	1	38.125	0.000	38.125	271.327	0.053	271.380
	2	19.180	0.273	19.453	134.142	1.877	136.019
	4	9.766	0.469	10.234	64.081	3.495	67.577
	8	5.039	0.645	5.684	32.310	4.578	36.888
	16	2.661	0.859	3.521	16.560	5.786	22.346
	32	1.472	1.047	2.520	8.720	7.248	15.968
512x512	1	151.719	0.000	151.719	1140.368	0.025	1140.393
	2	76.055	0.352	76.406	568.598	1.730	570.329
	4	38.262	0.488	38.750	282.705	3.330	286.035
	8	19.365	0.664	20.029	138.067	5.449	143.516
	16	9.873	0.869	10.742	65.990	6.299	72.289
	32	5.149	1.069	6.218	33.691	7.877	41.568
1kx1k	1	606.250	0.000	606.250	—	—	—
	2	303.477	0.234	303.711	2458.421	1.635	2460.056
	4	152.051	0.527	152.578	1225.214	3.093	1228.307
	8	76.406	0.674	77.080	608.574	4.447	613.021
	16	38.530	0.859	39.390	300.198	5.819	306.017
	32	19.612	1.062	20.674	142.588	7.320	149.907

Table 7: ANY, 2 dim. array, reduction along a dimension, array size = 256×256

Processors	Decomposition	Reduction Along	iPSC/860 time (ms)			iPSC/2 time (ms)		
			t_{calc}	t_{comm}	t_{total}	t_{calc}	t_{comm}	t_{total}
4	(any decomp, X)	row	10.313	0.078	10.391	72.847	15.318	88.165
		col	9.766	0.078	9.844	73.841	0.393	74.235
	(X, any decomp)	row	10.234	0.156	10.391	69.592	0.370	69.963
		col	10.000	0.156	10.156	72.706	15.387	88.093
	(any decomp, any decomp)							
	2×2	row	10.234	0.078	10.313	70.995	5.609	76.604
		col	9.844	0.078	9.922	72.932	5.567	78.499
8	(any decomp, X)	row	5.273	0.078	5.352	37.186	22.707	59.893
		col	4.961	0.117	5.078	36.935	0.309	37.244
	(X, any decomp)	row	5.352	0.039	5.391	34.275	0.285	34.560
		col	5.195	0.156	5.352	36.616	22.630	59.246
	(any decomp, any decomp)							
	4×2	row	5.234	0.117	5.352	35.873	9.738	45.611
		col	5.000	0.156	5.156	36.490	3.956	40.446
	2×4	row	5.313	0.039	5.352	34.978	3.934	38.911
		col	5.039	0.234	5.273	36.382	9.760	46.142
16	(any decomp, X)	row	2.813	0.098	2.910	19.422	30.581	50.0041
		col	2.578	0.078	2.656	18.514	0.289	18.803
	(X, any decomp)	row	2.793	0.117	2.910	16.939	0.290	17.229
		col	2.793	0.078	2.871	18.894	30.556	49.450
	(any decomp, any decomp)							
	4×4	row	2.773	0.039	2.813	17.719	7.226	24.945
		col	2.617	0.117	2.734	18.232	7.297	25.530
	8×2	row	2.773	0.039	2.813	18.357	14.732	33.088
		col	2.598	0.098	2.695	18.288	3.238	21.527
	2×8	row	2.813	0.078	2.891	17.277	3.256	20.533
		col	2.676	0.137	2.813	18.349	14.858	33.207
32	(any decomp, X)	row	1.568	0.103	1.670	10.515	38.411	48.927
		col	1.371	0.102	1.473	9.293	0.267	9.561
	(X, any decomp)	row	1.423	0.101	1.524	8.363	0.264	8.627
		col	1.503	0.101	1.604	10.114	38.398	48.512
	(any decomp, any decomp)							
	16×2	row	1.493	0.101	1.594	9.616	19.758	29.374
		col	1.376	0.100	1.476	9.173	2.032	11.204
	2×16	row	1.429	0.100	1.529	8.551	2.028	10.579
		col	1.435	0.102	1.537	9.481	19.732	29.213
	8×4	row	1.456	0.102	1.557	9.104	10.723	19.828
		col	1.384	0.101	1.485	9.147	5.761	14.908
	4×8	row	1.437	0.102	1.539	8.762	5.745	14.507
		col	1.401	0.101	1.502	9.212	10.765	19.977

Table 8: ANY, 2 dim. array, reduction along a dimension, array size = 512×512

Processors	Decomposition	Reduction Along	iPSC/860 time (ms)			iPSC/2 time (ms)		
			t_{calc}	t_{comm}	t_{total}	t_{calc}	t_{comm}	t_{total}
4	(any decomp, X)	row	40.025	0.103	40.128	316.874	28.887	345.761
		col	38.148	0.103	38.251	309.193	0.308	309.501
	(X, any decomp)	row	39.802	0.101	39.903	297.000	0.299	297.298
		col	38.351	0.100	38.451	303.651	28.754	332.405
	(any decomp, any decomp)							
	2×2	row	39.876	0.103	39.979	304.875	7.621	312.497
		col	38.215	0.103	38.318	309.639	7.606	317.245
8	(any decomp, X)	row	20.261	0.102	20.362	155.669	44.278	199.947
		col	19.180	0.078	19.258	154.646	0.316	154.962
	(X, any decomp)	row	20.430	0.039	20.469	144.798	0.309	145.108
		col	19.844	0.078	19.922	151.536	44.100	195.636
	(any decomp, any decomp)							
	4×2	row	20.195	0.078	20.273	151.231	14.874	166.105
		col	19.258	0.156	19.414	154.869	4.986	159.855
	2×4	row	20.234	0.117	20.352	147.942	5.027	152.968
		col	19.453	0.078	19.531	151.877	14.772	166.650
16	(any decomp, X)	row	10.410	0.117	10.527	75.310	60.533	135.843
		col	9.688	0.137	9.824	74.470	0.380	74.849
	(X, any decomp)	row	10.488	0.117	10.605	68.348	0.375	68.723
		col	10.254	0.137	10.391	73.200	60.574	133.774
	(any decomp, any decomp)							
	4×4	row	10.215	0.137	10.352	70.917	10.295	81.212
		col	9.785	0.176	9.961	72.919	10.345	83.264
	8×2	row	10.293	0.078	10.371	72.743	23.318	96.062
		col	9.766	0.117	9.883	73.845	4.386	78.231
	2×8	row	10.273	0.156	10.430	69.524	4.423	73.947
		col	10.020	0.020	10.039	72.710	23.317	96.027
32	(any decomp, X)	row	5.427	0.103	5.530	39.235	75.343	114.578
		col	4.933	0.102	5.035	37.245	0.302	37.547
	(X, any decomp)	row	5.139	0.101	5.240	33.693	0.288	33.981
		col	5.194	0.101	5.295	37.689	75.431	113.120
	(any decomp, any decomp)							
	16×2	row	5.278	0.103	5.381	37.200	30.585	67.785
		col	4.940	0.103	5.043	36.936	3.289	40.224
	2×16	row	5.149	0.101	5.249	34.280	3.288	37.568
		col	5.058	0.102	5.160	36.616	30.678	67.294
	8×4	row	5.204	0.101	5.305	35.848	14.322	50.170
		col	4.957	0.101	5.058	36.476	7.288	43.764
	4×8	row	5.167	0.101	5.268	34.965	7.291	42.255
		col	4.991	0.101	5.092	36.368	14.437	50.805

4.3 COUNT

The Fortran 90D specification for ALL is given in figure 2. The basic implementaion has been done in the same manner as in the case of ALL operatation. However, this function is different from ALL and ANY in the sense that the result is of type integer which is calculated by counting the number of TRUE elements in the logical array. The global operation in this case is a sum operation which adds the local results of all participating processors. For the COUNT operation, the timings for different array sizes and processor configurations are given in tables 9, 10, 11 and 12.

Table 9: COUNT, 1 dim. array, any decomposition

Array Size	Processors	iPSC/860 time (ms)			iPSC/2 time (ms)		
		t_{calc}	t_{comm}	t_{total}	t_{calc}	t_{comm}	t_{total}
64	1	0.092	0.000	0.092	0.149	0.070	0.219
	2	0.073	0.213	0.286	0.080	1.460	1.540
	4	0.052	0.313	0.365	0.062	2.833	2.895
	8	0.043	0.625	0.668	0.046	4.221	4.267
	16	0.039	0.664	0.703	0.051	5.720	5.771
	32	0.026	1.016	1.042	0.071	7.340	7.310
256	1	0.325	0.000	0.325	0.539	0.063	0.602
	2	0.213	0.313	0.526	0.280	1.460	1.740
	4	0.156	0.313	0.469	0.165	2.814	2.979
	8	0.076	0.703	0.779	0.096	4.225	4.321
	16	0.039	0.664	0.703	0.075	5.703	5.778
	32	0.029	1.035	1.064	0.084	7.113	7.297
1k	1	0.625	0.000	0.625	2.078	0.070	2.148
	2	0.313	0.313	0.625	1.090	1.450	2.540
	4	0.213	0.423	0.636	0.577	2.000	3.388
	8	0.156	0.625	0.781	0.296	3.228	4.524
	16	0.117	0.586	0.703	0.177	5.717	5.894
	32	0.096	0.957	1.053	0.132	7.199	7.331
4k	1	1.875	0.000	1.875	8.258	0.062	8.320
	2	0.925	0.313	1.238	4.268	1.485	5.753
	4	0.625	0.313	0.938	2.158	2.848	5.006
	8	0.156	0.859	1.016	1.097	4.246	5.343
	16	0.352	0.547	0.898	0.577	5.705	6.282
	32	0.176	1.035	1.211	0.329	7.133	7.532
16k	1	6.250	0.000	6.250	33.038	0.123	33.161
	2	3.438	0.313	3.750	16.990	1.740	18.730
	4	1.563	0.625	2.188	8.539	3.024	11.563
	8	0.625	1.016	1.641	4.278	4.283	8.565
	16	0.547	0.781	1.328	2.169	5.746	7.915
	32	0.391	1.016	1.406	1.126	7.214	8.340
64k	1	23.750	0.000	23.750	146.797	0.082	146.879
	2	11.875	0.313	12.188	73.626	1.675	75.301
	4	6.250	0.469	6.719	33.514	3.484	36.998
	8	3.125	0.938	4.063	17.024	4.580	21.604
	16	1.602	0.977	2.578	8.528	5.945	14.473
	32	0.938	1.074	2.012	4.305	7.310	11.615
256k	1	92.500	0.000	92.500	587.212	0.088	587.300
	2	46.563	0.313	46.875	293.811	1.640	295.451
	4	23.750	0.313	24.063	146.988	3.000	150.024
	8	11.875	0.625	12.500	73.614	4.502	78.116
	16	5.977	1.211	7.188	33.530	6.462	39.992
	32	3.262	1.113	4.375	17.049	7.588	24.637

Table 10: COUNT, 2 dim. array, reduction to scalar, any decomposition

Array Size	Processors	iPSC/860 time (ms)			iPSC/2 time (ms)		
		t_{calc}	t_{comm}	t_{total}	t_{calc}	t_{comm}	t_{total}
16x16	1	0.391	0.000	0.391	1.001	0.038	1.039
	2	0.273	0.234	0.508	0.537	1.415	1.952
	4	0.234	0.449	0.684	0.320	2.764	3.083
	8	0.195	0.625	0.820	0.191	4.147	4.338
	16	0.171	0.854	1.025	0.137	5.542	5.680
	32	0.098	1.035	1.133	0.072	6.951	7.022
32x32	1	0.781	0.000	0.781	3.858	0.079	3.937
	2	0.469	0.234	0.703	1.985	1.449	3.434
	4	0.332	0.449	0.781	1.068	2.778	3.846
	8	0.264	0.645	0.908	0.603	4.126	4.729
	16	0.225	0.854	1.079	0.379	5.513	5.892
	32	0.205	1.050	1.255	0.253	6.926	7.179
64x64	1	2.656	0.000	2.656	15.411	0.053	15.465
	2	1.367	0.273	1.641	7.836	1.595	9.432
	4	0.801	0.449	1.250	4.045	2.874	6.919
	8	0.508	0.625	1.133	2.140	4.183	6.322
	16	0.342	0.845	1.187	1.201	5.555	6.756
	32	0.278	1.045	1.323	0.727	6.930	7.656
128x128	1	9.766	0.078	9.844	62.338	0.145	62.483
	2	5.000	0.234	5.234	31.324	1.630	32.954
	4	2.637	0.449	3.086	15.899	3.038	18.937
	8	1.426	0.645	2.070	8.185	4.394	12.580
	16	0.840	0.835	1.675	4.330	5.629	9.959
	32	0.532	1.050	1.582	2.398	7.000	9.398
256x256	1	38.359	0.000	38.359	271.801	0.118	271.920
	2	19.336	0.234	19.570	134.662	1.779	136.442
	4	9.824	0.430	10.254	63.849	3.640	67.489
	8	5.029	0.664	5.693	32.278	4.417	36.695
	16	2.676	0.869	3.545	16.599	5.814	22.413
	32	1.475	1.069	2.544	8.761	7.163	15.924
512x512	1	152.188	0.000	152.188	1140.892	0.069	1140.960
	2	76.328	0.313	76.641	570.121	1.811	571.932
	4	38.418	0.430	38.848	283.859	3.531	287.390
	8	19.375	0.664	20.039	139.751	5.918	145.669
	16	9.922	0.854	10.776	65.857	7.626	73.483
	32	5.156	1.074	6.230	33.675	7.243	40.918
1kx1k	1	607.109	0.000	607.109	—	—	—
	2	303.945	0.273	304.219	2463.460	1.785	2465.245
	4	152.324	0.488	152.813	1230.500	3.369	1233.869
	8	76.445	0.654	77.100	613.680	5.453	619.133
	16	38.579	0.879	39.458	304.452	7.529	311.981
	32	19.619	1.072	20.691	148.419	9.521	157.940

Table 11: COUNT, 2 dim. array, reduction along a dimension, array size = 256×256

Processors	Decomposition	Reduction Along	iPSC/860 time (ms)			iPSC/2 time (ms)		
			t_{calc}	t_{comm}	t_{total}	t_{calc}	t_{comm}	t_{total}
4	(any decomp, X)	row	10.370	3.662	14.033	72.573	14.934	87.507
		col	9.819	0.081	9.900	73.830	0.382	74.212
	(X, any decomp)	row	10.663	0.067	10.731	69.485	0.369	69.8
		col	10.268	3.680	13.948	72.514	14.981	87.495
	(any decomp, any decomp)							
	2×2	row	10.432	1.204	11.636	70.849	5.539	76.387
		col	9.966	1.205	11.171	72.855	5.567	78.423
8	(any decomp, X)	row	5.356	5.458	10.814	36.962	22.073	59.035
		col	5.004	0.079	5.083	36.942	0.293	37.235
	(X, any decomp)	row	5.708	0.066	5.774	34.265	0.276	34.541
		col	5.535	5.457	10.992	36.447	21.932	58.379
	(any decomp, any decomp)							
	4×2	row	5.352	2.330	7.681	35.795	9.530	45.325
		col	5.080	0.917	5.997	36.446	3.875	40.321
	2×4	row	5.451	0.972	6.423	34.920	3.912	38.832
		col	5.230	2.339	7.569	36.270	9.552	45.822
16	(any decomp, X)	row	2.845	7.271	10.115	19.211	29.617	48.828
		col	2.593	0.079	2.671	18.520	0.289	18.810
	(X, any decomp)	row	3.010	0.065	3.075	16.930	0.279	17.209
		col	2.986	7.261	10.247	18.712	29.563	48.275
	(any decomp, any decomp)							
	4×4	row	2.833	1.712	4.546	17.687	7.249	24.936
		col	2.703	1.719	4.422	18.184	7.122	25.306
	8×2	row	2.804	3.453	6.257	18.283	14.418	32.701
		col	2.630	0.830	3.460	18.267	3.214	21.480
	2×8	row	2.947	0.858	3.806	17.263	3.223	20.486
		col	2.851	3.479	6.330	18.264	14.498	32.762
32	(any decomp, X)	row	1.591	9.111	10.701	10.319	37.178	47.497
		col	1.389	0.077	1.466	9.296	0.262	9.559
	(X, any decomp)	row	1.590	0.062	1.652	8.363	0.266	8.629
		col	1.651	9.101	10.752	9.945	37.138	47.083
	(any decomp, any decomp)							
	16×2	row	1.533	4.601	6.134	9.526	19.286	28.812
		col	1.407	0.358	1.766	9.165	2.034	11.199
	2×16	row	1.596	0.377	1.973	8.549	2.007	10.557
		col	1.579	4.624	6.202	9.392	19.243	28.635
	8×4	row	1.528	2.509	4.036	9.044	10.541	19.585
		col	1.443	1.488	2.931	9.129	5.690	14.819
	4×8	row	1.574	1.548	3.122	8.748	5.683	14.431
		col	1.516	2.540	4.055	9.171	10.524	19.695

Table 12: COUNT, 2 dim. array, reduction along a dimension, array size = 512×512

Processors	Decomposition	Reduction Along	iPSC/860 time (ms)			iPSC/2 time (ms)		
			t_{calc}	t_{comm}	t_{total}	t_{calc}	t_{comm}	t_{total}
4	(any decomp, X)	row	40.302	6.956	47.257	316.039	27.488	343.527
		col	38.380	0.081	38.461	309.103	0.297	309.400
	(X, any decomp)	row	40.894	0.071	40.964	296.850	0.329	297.179
		col	39.270	7.181	46.451	303.291	27.387	330.679
	(any decomp, any decomp)							
	2×2	row	40.425	1.909	42.334	304.519	7.367	311.886
		col	38.677	1.908	40.585	309.529	7.334	316.863
8	(any decomp, X)	row	20.409	10.508	30.917	154.717	42.034	196.751
		col	19.297	0.080	19.377	154.583	0.331	154.915
	(X, any decomp)	row	21.112	0.066	21.178	144.708	0.333	145.041
		col	20.353	10.654	31.008	151.231	42.023	193.254
	(any decomp, any decomp)							
	4×2	row	20.396	3.674	24.070	150.811	14.225	165.036
		col	19.446	1.216	20.662	154.807	4.876	159.683
	2×4	row	20.598	1.245	21.843	147.808	4.926	152.734
		col	19.747	3.672	23.419	151.706	14.187	165.894
16	(any decomp, X)	row	10.450	14.051	24.50	74.980	58.132	133.112
		col	9.743	0.081	9.824	74.452	0.397	74.849
	(X, any decomp)	row	11.162	0.067	11.229	68.328	0.386	68.714
		col	10.845	14.137	24.982	72.859	58.206	131.065
	(any decomp, any decomp)							
	4×4	row	10.426	2.333	12.759	70.859	9.987	80.846
		col	9.963	2.357	12.320	72.847	9.951	82.798
	8×2	row	10.368	5.568	15.935	72.558	22.451	95.009
		col	9.817	0.930	10.747	73.832	4.306	78.139
	2×8	row	10.652	1.005	11.656	69.480	4.363	73.843
		col	10.257	5.471	15.728	72.509	22.438	94.947
32	(any decomp, X)	row	5.467	17.635	23.102	38.902	72.472	111.37
		col	4.965	0.077	5.042	37.248	0.291	37.539
	(X, any decomp)	row	5.792	0.065	5.857	33.692	0.288	33.980
		col	5.760	17.696	23.457	37.331	72.524	109.855
	(any decomp, any decomp)							
	16×2	row	5.352	7.274	12.626	36.960	29.522	66.482
		col	5.000	0.833	5.833	36.943	3.246	40.19
	2×16	row	5.673	0.899	6.572	34.262	3.272	37.534
		col	5.506	7.269	12.775	36.445	29.584	66.028
	8×4	row	5.342	3.465	8.806	35.787	14.294	50.081
		col	5.071	1.728	6.799	36.436	7.140	43.576
	4×8	row	5.433	1.782	7.215	34.926	7.130	42.055
		col	5.216	3.495	8.711	36.270	13.807	50.077

- *Syntax*: **MAXVAL(ARRAY, DIM, MASK), MINVAL(ARRAY, DIM, MASK), PRODUCT(ARRAY, DIM, MASK), SUM(ARRAY, DIM, MASK)**
- *Optional Arguments*: DIM, MASK
- *Description (MAXVAL)*: Determines the maximum value of the elements of ARRAY along dimension DIM corresponding to the true elements of MASK.
- *Description (MINVAL)*: Determines the minimum value of the elements of ARRAY along dimension DIM corresponding to the true elements of MASK.
- *Description (PRODUCT)*: Determines the product of all the elements along dimension DIM corresponding to the true elements of MASK.
- *Description (SUM)*: Determines the sum of all the elements along dimension DIM corresponding to the true elements of MASK.
- *Arguments*:
  1. ARRAY: must be of type integer or real. It must not be scalar.
  2. DIM (optional): must be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY.
  3. MASK (optional): must be of type logical and must be conformable with ARRAY.

The result is of the same type and type parameter as ARRAY. It is scalar if DIM is absent or ARRAY has rank one; otherwise the result is an array of rank $n - 1$ and of shape $(d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of ARRAY.

Figure 3: FORTRAN 90D Specification for MAXVAL, MINVAL, PRODUCT and SUM

4.4 MAXVAL and MINVAL

The Fortran 90D specification for MAXVAL is given in figure 3. For a one-dimensional array, each processor calculates the maximum value in the local array and then all processors perform a global maximum operation to find the maximum element among all local arrays. In the case of two-dimensional arrays, if the DIM argument is not specified, the problem is essentially the same as that for a one-dimensional array. If DIM is specified, the result is a one-dimensional array. Each processor determines the maximum value along each row (or column) of the local array. A global maximum operation may or may not be necessary depending on whether the rows (or columns) are distributed or not. If the rows (or columns) of the array are not distributed, the maximum values determined by each processor are the maximum values along the rows (or columns). If the rows (or columns) are distributed, then those processors which share a particular row (or column) of the array, perform a global maximum operation to determine the maximum value along the row (or column). Thus, a separate global operation takes place in every row (or column) of the grid. If the MASK array is specified, then the maximum of only those elements which correspond to the true values of MASK, is calculated.

The timings for different array sizes and processor configurations are given in tables 13, 14, 15, 16 and 17. Graphs of speedup versus number of processors for different array sizes are given in figures 4 and 5. We see that the speedup is higher for larger array sizes. This is because as the number of processors increases, the computation time decreases almost linearly, but the communication time increases. For large array sizes, the computation time is much higher than the communication time and hence as the number of processors increases, the speedup increases. MINVAL has been implemented in the same way as MAXVAL, except that instead of finding the maximum value, the minimum value is determined.

Table 13: MAXVAL, 1 dim. array, any decomposition

Array Size	Processors	iPSC/860 time (ms)			iPSC/2 time (ms)		
		t_{calc}	t_{comm}	t_{total}	t_{calc}	t_{comm}	t_{total}
64	1	0.025	0.000	0.025	0.180	0.000	0.180
	2	0.013	0.402	0.415	0.090	1.420	1.510
	4	0.007	0.798	0.805	0.055	2.795	2.850
	8	0.005	1.800	1.805	0.030	4.197	4.227
	16	0.003	1.668	1.671	0.024	5.629	5.653
	32	0.002	2.175	2.177	0.017	7.057	7.074
256	1	0.090	0.000	0.090	0.620	0.000	0.620
	2	0.046	0.410	0.456	0.310	1.420	1.730
	4	0.024	0.801	0.825	0.170	2.770	2.940
	8	0.013	1.808	1.821	0.090	4.172	4.262
	16	0.007	1.676	1.683	0.050	5.596	5.646
	32	0.005	2.164	2.169	0.032	7.052	7.085
1K	1	0.364	0.000	0.364	2.460	0.000	2.460
	2	0.183	0.411	0.594	1.240	1.420	2.660
	4	0.091	0.831	0.922	0.620	2.785	3.405
	8	0.048	1.825	1.873	0.317	4.185	4.502
	16	0.024	1.661	1.685	0.164	5.591	5.755
	32	0.013	2.133	2.146	0.089	7.035	7.124
4K	1	1.616	0.000	1.616	9.800	0.000	9.800
	2	0.735	0.504	1.239	4.910	1.440	6.350
	4	0.364	0.838	1.202	2.455	2.795	5.250
	8	0.183	1.842	2.025	1.235	4.187	5.422
	16	0.091	1.651	1.742	0.624	5.612	6.236
	32	0.046	2.131	2.177	0.320	7.041	7.361
16K	1	6.594	0.000	6.594	39.22	0.000	39.22
	2	3.291	0.507	3.798	19.62	1.720	21.34
	4	1.614	0.925	2.539	9.800	2.870	12.67
	8	0.735	1.972	2.707	4.900	4.252	9.152
	16	0.364	1.690	2.054	2.457	5.641	8.099
	32	0.183	2.175	2.358	1.236	7.064	8.300
64K	1	26.44	0.000	26.44	171.2	0.000	171.2
	2	13.20	0.537	13.74	85.62	1.690	87.31
	4	6.597	0.938	7.535	41.52	2.950	44.47
	8	3.290	1.975	5.265	19.61	4.510	24.12
	16	1.615	1.762	3.377	9.799	5.741	15.54
	32	0.735	2.233	2.967	4.904	7.111	12.02
256K	1	105.9	0.000	105.9	685.1	0.000	685.1
	2	52.94	0.554	53.49	342.6	1.630	344.2
	4	26.44	0.977	27.42	171.2	3.135	174.3
	8	13.20	2.038	15.24	85.62	4.450	90.07
	16	6.596	1.769	8.365	39.22	6.236	45.46
	32	3.290	2.236	5.526	19.61	7.408	27.02

Table 14: MAXVAL, 2 dim. array, reduction to scalar, any decomposition

Array Size	Processors	iPSC/860 time (ms)			iPSC/2 time (ms)		
		t_{calc}	t_{comm}	t_{total}	t_{calc}	t_{comm}	t_{total}
16×16	1	0.162	0.000	0.162	1.500	0.000	1.500
	2	0.086	0.413	0.499	0.770	1.410	2.180
	4	0.048	0.807	0.854	0.415	2.750	3.165
	8	0.027	1.210	1.237	0.235	4.167	4.402
	16	0.018	1.727	1.745	0.150	5.592	5.742
	32	0.011	2.216	2.227	0.088	7.029	7.117
32×32	1	0.633	0.000	0.633	5.880	0.000	5.880
	2	0.324	0.424	0.748	2.930	1.410	4.340
	4	0.169	0.836	1.005	1.500	2.740	4.240
	8	0.090	1.253	1.343	0.797	4.145	4.942
	16	0.052	1.681	1.733	0.450	5.576	6.026
	32	0.032	2.193	2.225	0.276	6.964	7.240
64×64	1	2.587	0.000	2.587	23.12	0.000	23.12
	2	1.270	0.507	1.777	11.51	1.440	12.95
	4	0.644	0.851	1.495	5.780	2.765	8.545
	8	0.334	1.241	1.575	2.952	4.160	7.112
	16	0.179	1.692	1.871	1.552	5.560	7.112
	32	0.101	2.188	2.289	0.872	6.966	7.838
128×128	1	10.33	0.000	10.33	92.78	0.000	92.78
	2	5.192	0.533	5.725	46.04	1.650	47.69
	4	2.608	0.930	3.538	22.88	2.810	25.69
	8	1.292	1.334	2.626	11.49	4.220	15.71
	16	0.665	1.759	2.425	5.861	5.621	11.48
	32	0.354	2.184	2.538	3.076	6.966	10.04
256×256	1	41.24	0.000	41.24	387.3	0.000	387.3
	2	20.65	0.534	21.18	190.8	1.640	192.4
	4	10.38	0.944	11.32	91.77	3.300	95.07
	8	5.230	1.346	6.576	45.66	4.405	50.06
	16	2.648	1.761	4.409	22.92	5.680	28.60
	32	1.334	2.258	3.592	11.68	7.071	18.75
512×512	1	164.9	0.000	164.9	1555	0.000	1555
	2	82.50	0.573	83.07	773.6	1.610	775.3
	4	41.33	0.994	42.32	381.0	3.005	384.0
	8	20.75	1.393	22.14	189.3	4.372	193.7
	16	10.45	1.808	12.26	91.16	6.209	97.37
	32	5.311	2.238	7.549	45.80	7.285	53.08
$1K \times 1K$	1	658.8	0.000	658.8	—	—	—
	2	329.8	0.588	330.4	3108	1.620	3109
	4	165.0	1.101	166.1	1546	2.985	1549
	8	82.67	1.403	84.07	761.7	4.412	766.1
	16	41.51	1.838	43.35	378.3	5.876	384.2
	32	20.90	2.269	23.17	188.4	7.266	195.7

Table 15: MAXVAL, 2 dim. array, reduction along a dimension, array size = 64×64

Processors	Decomposition	Reduction Along	iPSC/860 time (ms)			iPSC/2 time (ms)		
			t_{calc}	t_{comm}	t_{total}	t_{calc}	t_{comm}	t_{total}
4	(any decomp, X)	row	0.654	0.000	0.654	8.470	0.000	8.470
		col	0.675	2.123	2.798	8.495	6.525	15.02
	(X, any decomp)	row	0.685	2.131	2.816	8.550	6.515	15.06
		col	0.667	0.000	0.667	8.780	0.000	8.780
	(any decomp, any decomp)							
2×2		row	0.771	0.984	1.755	10.27	2.650	12.92
		col	0.769	0.977	1.746	11.62	2.675	14.29
8	(any decomp, X)	row	0.327	0.000	0.327	4.187	0.000	4.187
		col	0.351	3.225	3.576	4.200	9.847	14.05
	(X, any decomp)	row	0.354	3.239	3.593	4.310	9.845	14.15
		col	0.334	0.000	0.334	4.597	0.000	4.597
	(any decomp, any decomp)							
4×2		row	0.417	0.503	0.920	5.180	1.730	6.910
		col	0.427	1.874	2.301	5.272	5.240	10.51
2×4		row	0.431	1.867	2.298	5.945	5.240	11.19
		col	0.415	0.508	0.923	6.000	1.702	7.702
16	(any decomp, X)	row	0.166	0.000	0.166	2.091	0.000	2.091
		col	0.190	4.453	4.643	2.120	13.30	15.42
	(X, any decomp)	row	0.190	4.467	4.657	2.176	13.30	15.48
		col	0.169	0.000	0.169	2.499	0.000	2.499
	(any decomp, any decomp)							
4×4		row	0.253	0.982	1.235	2.815	3.360	6.175
		col	0.249	0.973	1.222	2.866	3.352	6.219
8×2		row	0.241	0.469	0.710	3.847	1.561	5.409
		col	0.274	2.847	3.121	3.960	7.944	11.90
2×8		row	0.276	2.838	3.114	3.986	8.052	12.04
		col	0.238	0.471	0.709	3.924	1.556	5.480
32	(any decomp, X)	row	0.086	0.000	0.086	1.060	0.000	1.060
		col	0.110	5.730	5.840	1.109	16.72	17.83
	(X, any decomp)	row	0.108	5.733	5.841	1.115	16.75	17.87
		col	0.088	0.000	0.088	1.453	0.000	1.453
	(any decomp, any decomp)							
16×2		row	0.155	0.446	0.601	1.571	1.477	3.047
		col	0.229	3.974	4.203	2.075	10.89	12.97
2×16		row	0.233	3.964	4.197	2.149	10.91	13.06
		col	0.152	0.451	0.603	1.794	1.472	3.266
8×4		row	0.165	0.912	1.077	2.175	3.061	5.236
		col	0.184	1.507	1.691	2.288	5.035	7.323
4×8		row	0.187	1.533	1.720	2.290	5.044	7.334
		col	0.162	0.921	1.083	2.198	3.048	5.246

Table 16: MAXVAL, 2 dim. array, reduction along a dimension, array size = 256×256

Processors	Decomposition	Reduction Along	iPSC/860 time (ms)			iPSC/2 time (ms)		
			t_{calc}	t_{comm}	t_{total}	t_{calc}	t_{comm}	t_{total}
4	(any decomp, X)	row	14.19	0.000	14.19	147.8	0.000	147.8
		col	10.97	3.810	14.78	146.4	14.42	160.8
	(X, any decomp)	row	14.52	3.864	18.38	161.3	14.49	175.8
		col	11.05	0.000	11.05	161.3	0.000	161.3
	(any decomp, any decomp)							
2×2		row	14.43	1.339	15.77	161.7	5.505	167.2
		col	11.77	1.307	13.08	184.6	5.255	189.9
8	(any decomp, X)	row	7.070	0.000	7.070	70.42	0.000	70.42
		col	5.506	5.837	11.343	69.60	21.37	90.98
	(X, any decomp)	row	7.327	5.921	13.25	79.80	21.46	101.3
		col	5.523	0.000	5.523	80.66	0.000	80.66
	(any decomp, any decomp)							
4×2		row	7.258	1.043	8.301	74.52	3.785	78.31
		col	5.948	2.665	8.613	79.71	9.345	89.06
2×4		row	7.317	2.701	10.02	80.98	9.410	90.39
		col	5.926	1.075	7.001	92.57	3.807	96.38
16	(any decomp, X)	row	3.436	0.000	3.436	34.15	0.000	34.15
		col	2.775	7.885	10.66	33.73	28.46	62.19
	(X, any decomp)	row	3.546	7.943	11.49	39.09	28.39	67.48
		col	2.752	0.000	2.752	40.33	0.000	40.33
	(any decomp, any decomp)							
4×4		row	3.585	2.169	5.754	37.07	6.677	43.75
		col	3.004	2.173	5.177	37.00	6.706	43.70
8×2		row	3.578	0.982	4.560	36.75	2.851	39.60
		col	3.035	4.117	7.152	45.59	13.67	59.26
2×8		row	3.629	4.104	7.733	40.76	13.81	54.57
		col	2.988	0.983	3.971	46.41	2.915	49.32
32	(any decomp, X)	row	1.322	0.000	1.322	16.81	0.000	16.81
		col	1.386	9.997	11.38	16.54	35.43	51.97
	(X, any decomp)	row	1.427	10.03	11.46	18.72	35.43	54.15
		col	1.333	0.000	1.333	20.18	0.000	20.18
	(any decomp, any decomp)							
16×2		row	1.457	0.581	2.038	17.48	1.746	19.23
		col	1.582	5.570	7.152	18.10	18.40	36.50
2×16		row	1.561	5.649	7.210	20.25	18.45	38.70
		col	1.489	0.557	2.046	20.34	1.756	22.10
8×4		row	1.470	1.934	3.404	20.66	5.325	25.99
		col	1.527	3.445	4.972	28.96	10.07	39.03
4×8		row	1.500	3.407	4.907	21.30	10.03	31.33
		col	1.501	1.904	3.406	29.30	5.352	34.66

Table 17: MAXVAL, 2 dim. array, reduction along a dimension, array size = $1K \times 1K$

Processors	Decomposition	Reduction Along	iPSC/860 time (ms)			iPSC/2 time (ms)		
			t_{calc}	t_{comm}	t_{total}	t_{calc}	t_{comm}	t_{total}
4	(any decomp, X)	row	264.2	0.000	264.2	3079	0.000	3079
		col	176.8	15.42	192.2	3043	54.56	3097
	(X, any decomp)	row	375.8	17.21	393.0	4088	55.66	4144
		col	186.5	0.000	186.5	3901	0.000	3901
	(any decomp, any decomp)							
2×2		row	307.9	4.095	311.1	3917	13.90	3931
		col	186.6	3.970	190.6	3849	13.82	3863
8	(any decomp, X)	row	122.5	0.000	122.5	1323	0.000	1323
		col	87.64	23.27	110.9	1304	83.17	1387
	(X, any decomp)	row	178.1	24.55	202.6	2034	84.78	2119
		col	93.24	0.000	93.24	1951	0.000	1951
	(any decomp, any decomp)							
4×2		row	130.3	2.156	132.4	1539	7.067	1546
		col	93.40	7.831	101.2	1522	27.59	1549
2×4		row	148.9	8.001	156.9	1956	27.88	1983
		col	93.37	2.016	95.39	1925	7.170	1932
16	(any decomp, X)	row	57.33	0.000	57.33	606.4	0.000	606.4
		col	43.77	31.67	75.44	596.6	112.3	709.0
	(X, any decomp)	row	63.03	32.30	95.34	1007	114.6	1121
		col	46.61	0.000	46.61	975.3	0.000	975.3
	(any decomp, any decomp)							
4×4		row	59.21	4.197	63.40	768.8	13.99	782.8
		col	46.76	3.892	50.65	761.2	13.82	775.0
8×2		row	58.00	1.490	59.49	661.9	4.701	666.6
		col	46.82	11.93	58.75	751.1	41.79	792.9
2×8		row	61.04	12.22	73.26	975.2	42.57	1017
		col	46.72	1.368	48.09	962.6	4.625	967.2
32	(any decomp, X)	row	28.29	0.000	28.29	288.8	0.000	288.8
		col	21.93	40.25	62.18	284.0	141.2	425.2
	(X, any decomp)	row	30.26	40.03	70.29	492.4	144.9	637.4
		col	23.31	0.000	23.31	487.7	0.000	487.7
	(any decomp, any decomp)							
16×2		row	28.53	1.095	29.62	303.1	3.442	306.6
		col	23.56	16.30	39.85	299.2	56.83	356.1
2×16		row	29.68	16.19	45.87	484.8	57.69	542.4
		col	23.41	1.083	24.49	481.5	3.440	485.0
8×4		row	28.69	2.675	31.36	331.8	8.977	340.8
		col	23.48	6.028	29.50	474.6	21.24	495.8
4×8		row	29.12	3.983	35.10	385.8	21.01	406.8
		col	23.42	2.691	26.11	481.1	8.979	490.0

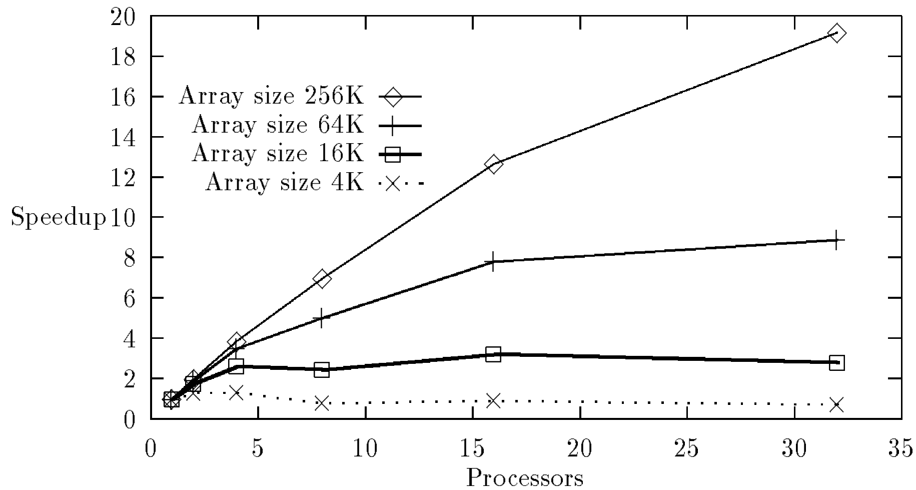

Figure 4: MAXVAL on iPSC/860 for one-dimensional array

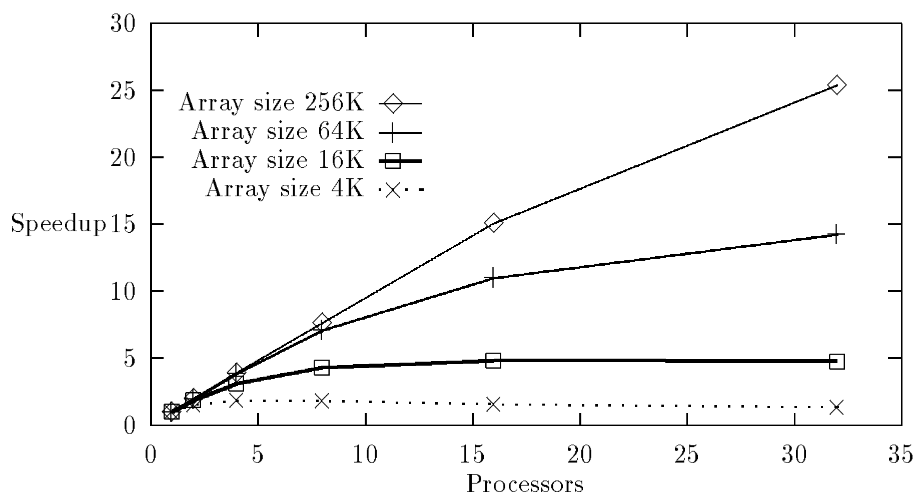

Figure 5: MAXVAL on iPSC/2 for one-dimensional array

4.5 PRODUCT

The Fortran 90D specification for PRODUCT is given in figure 3. PRODUCT is implemented in the same way as MAXVAL, except that each processor finds the product of all elements in the local array and then all processors perform a global multiplication operation.

The timings for different array sizes and processor configurations are given in tables 18, 19, 20 and 21.

Table 18: 1D PRODUCT Reduction, any decomposition

Array Size	Processors	iPSC/860 time (ms)			iPSC/2 time (ms)		
		t_{calc}	t_{comm}	t_{total}	t_{calc}	t_{comm}	t_{total}
64	1	0.110	0.000	0.110	0.198	0.045	0.243
	2	0.070	0.120	0.190	0.094	1.415	1.509
	4	0.065	0.313	0.378	0.063	2.766	2.829
	8	0.058	0.625	0.683	0.053	4.145	4.198
	16	0.056	0.703	0.759	0.051	5.601	5.652
	32	0.048	0.996	1.134	0.042	7.222	7.262
256	1	0.625	0.000	0.625	0.650	0.046	0.696
	2	0.313	0.120	0.433	0.333	1.405	1.738
	4	0.213	0.313	0.526	0.187	2.756	2.943
	8	0.156	0.625	0.781	0.120	4.138	4.258
	16	0.117	0.781	0.898	0.084	5.578	5.662
	32	0.078	1.094	1.172	0.092	7.000	7.092
1k	1	0.625	0.000	0.625	2.552	0.029	2.581
	2	0.313	0.120	0.313	1.288	1.414	2.702
	4	0.156	0.469	0.625	0.660	2.764	3.424
	8	0.096	0.625	0.721	0.354	4.153	4.507
	16	0.069	0.898	0.967	0.199	5.587	5.786
	32	0.046	0.957	1.003	0.132	7.120	7.252
4k	1	1.875	0.000	1.875	10.041	0.128	10.169
	2	0.625	0.625	1.250	5.046	1.575	6.620
	4	0.313	0.781	1.094	2.539	2.841	5.380
	8	0.234	0.703	0.938	1.290	4.175	5.465
	16	0.174	0.781	1.955	0.671	5.582	6.253
	32	0.117	1.094	1.211	0.354	7.020	7.374
16k	1	6.250	0.000	6.250	40.288	0.102	40.390
	2	3.125	0.625	3.750	20.114	1.687	21.802
	4	1.563	0.781	2.344	10.072	3.077	13.150
	8	0.859	0.703	1.563	5.040	4.360	9.400
	16	0.430	0.938	1.367	2.551	5.693	8.243
	32	0.195	1.172	1.367	1.142	7.214	8.356
64k	1	24.375	0.000	24.375	173.023	0.104	173.126
	2	11.875	0.625	12.500	86.535	1.623	88.158
	4	6.250	0.625	6.875	40.270	3.409	43.679
	8	3.203	0.703	3.906	20.119	4.536	24.655
	16	1.602	0.898	2.500	10.083	5.887	15.970
	32	0.781	1.172	1.953	5.329	7.189	12.518
256k	1	98.125	0.000	98.125	692.102	0.106	692.208
	2	49.375	0.625	50.000	346.069	1.622	347.691
	4	24.531	0.625	25.156	173.031	3.072	176.103
	8	12.344	0.781	13.125	86.541	4.414	90.954
	16	6.250	0.938	7.188	40.269	6.321	46.589
	32	3.145	1.094	4.238	19.129	7.627	24.756

- *Syntax*: **CSHIFT(ARRAY, SHIFT, DIM)**
- *Optional Arguments*: MASK
- *Description*: Performs a circular shift on an array expression of rank one or performs a circular shift on all the complete rank one sections along a given dimension of an array expression of rank two or greater. Elements shifted out at one end of a section are shifted in at the other end. Different sections may be shifted by different amounts and in different directions.
- *Arguments*:

  1. ARRAY: may be of any type. It must not be scalar.
  2. SHIFT: must be of type integer and must be scalar if ARRAY has rank one; otherwise, it must be scalar or of rank $(d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of the array.
  3. DIM (optional): must be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY. If DIM is omitted, it is as if it were present with the value 1.

The result is of the same type and type parameter as ARRAY and has the shape of ARRAY.

Figure 6: FORTRAN 90D Specification for CSHIFT

4.6 SUM

The Fortran 90D specification for SUM is given in figure 3. SUM is also implemented in the same way as MAXVAL, except that each processor finds the sum of all elements in the local array and then all processors perform a global sum operation.

The timings for different array sizes and processor configurations are given in tables 22, 23, 24, 25 and 26.

Table 19: 2D PRODUCT reduction to scalar, any decomposition

Array Size	Processors	iPSC/860 time (ms)			iPSC/2 time (ms)		
		t_{calc}	t_{comm}	t_{total}	t_{calc}	t_{comm}	t_{total}
16x16	1	0.313	0.000	0.313	1.541	0.071	1.612
	2	0.273	0.234	0.508	0.815	1.406	2.222
	4	0.234	0.449	0.684	0.461	2.753	3.215
	8	0.195	0.645	0.840	0.276	4.132	4.408
	16	0.105	0.850	0.955	0.183	5.536	5.720
	32	0.066	0.957	1.023	0.085	6.999	7.084
32x32	1	0.781	0.000	0.781	6.073	0.033	6.105
	2	0.508	0.273	0.781	3.130	1.441	4.571
	4	0.332	0.449	0.781	1.643	2.780	4.424
	8	0.244	0.645	0.889	0.900	4.112	5.012
	16	0.220	0.850	1.069	0.527	5.543	6.071
	32	0.195	1.055	1.250	0.337	6.957	7.294
64x64	1	2.656	0.000	2.656	24.342	0.049	24.391
	2	1.406	0.273	1.680	12.320	1.558	13.877
	4	0.801	0.449	1.250	6.303	2.860	9.163
	8	0.527	0.635	1.162	3.286	4.167	7.453
	16	0.342	0.850	1.191	1.776	5.595	7.371
	32	0.254	1.055	1.309	1.030	6.970	8.000
128x128	1	10.156	0.000	10.156	97.553	0.180	97.733
	2	5.195	0.234	5.430	48.941	1.645	50.586
	4	2.715	0.449	3.164	24.723	3.052	27.775
	8	1.455	0.664	2.119	12.638	4.382	17.020
	16	0.854	0.850	1.704	6.583	5.659	12.242
	32	0.586	1.035	1.621	3.558	7.008	10.566
256x256	1	39.844	0.000	39.844	411.472	0.054	411.526
	2	20.078	0.234	20.313	204.142	1.647	205.789
	4	10.195	0.449	10.645	99.211	3.589	102.799
	8	5.234	0.645	5.879	49.985	4.442	54.427
	16	2.769	0.850	3.618	25.508	5.824	31.332
	32	1.504	1.152	2.656	13.281	7.179	20.460
512x512	1	158.516	0.000	158.516	1706.084	0.046	1706.130
	2	79.492	0.234	79.727	851.365	1.640	853.005
	4	39.941	0.488	40.430	423.907	3.064	426.972
	8	20.186	0.674	20.859	208.944	4.473	213.417
	16	10.313	0.859	11.172	101.550	6.743	108.294
	32	5.332	1.055	6.387	51.592	7.254	58.846
1kx1k	1	632.969	0.000	632.969	—	—	—
	2	316.797	0.234	317.031	3595.223	1.709	3596.932
	4	158.730	0.449	159.180	1793.584	3.022	1796.606
	8	79.697	0.654	80.352	892.737	4.385	897.122
	16	40.171	0.889	41.060	442.290	5.799	448.089
	32	20.430	1.055	21.484	214.154	7.296	221.450

Table 20: PRODUCT, 2 dim. array, reduction along a dimension, array size = 256×256

Processors	Decomposition	Reduction Along	iPSC/860 time (ms)			iPSC/2 time (ms)		
			t_{calc}	t_{comm}	t_{total}	t_{calc}	t_{comm}	t_{total}
4	(any decomp, X)	row	10.215	0.313	10.527	107.780	15.356	123.135
		col	10.088	0.059	10.146	108.730	0.404	109.135
	(X, any decomp)	row	10.156	0.039	10.195	104.716	0.359	105.076
		col	10.273	0.303	10.576	107.688	15.254	122.942
	(any decomp, any decomp)							
	2×2	row	10.088	0.107	10.195	106.084	5.553	111.637
		col	10.078	0.107	10.186	108.049	5.553	113.603
8	(any decomp, X)	row	5.469	0.254	5.723	54.647	22.346	76.993
		col	5.322	0.049	5.371	54.382	0.281	54.663
	(X, any decomp)	row	5.361	0.034	5.396	51.686	0.297	51.984
		col	5.493	0.259	5.752	54.150	22.241	76.391
	(any decomp, any decomp)							
	4×2	row	5.259	0.122	5.381	53.439	9.604	63.043
		col	5.225	0.054	5.278	54.028	3.916	57.944
	2×4	row	5.239	0.063	5.303	52.564	3.867	56.431
		col	5.264	0.127	5.391	53.845	9.734	63.579
16	(any decomp, X)	row	3.118	0.198	3.315	28.193	29.927	58.120
		col	2.969	0.037	3.005	27.233	0.286	27.519
	(X, any decomp)	row	2.954	0.037	2.991	25.856	0.278	26.134
		col	3.093	0.205	3.298	27.619	29.896	57.516
	(any decomp, any decomp)							
	4×4	row	2.781	0.066	2.847	26.508	7.151	33.659
		col	2.783	0.063	2.847	26.962	7.214	34.177
	8×2	row	2.869	0.107	2.976	27.170	14.401	41.571
		col	2.810	0.044	2.854	27.060	3.234	30.294
	2×8	row	2.805	0.049	2.854	25.993	3.283	29.277
		col	2.864	0.100	2.964	27.110	14.444	41.554
32	(any decomp, X)	row	1.908	0.183	2.091	14.931	37.698	52.629
		col	1.754	0.028	1.782	13.646	0.276	13.922
	(X, any decomp)	row	1.753	0.038	1.791	12.751	0.266	13.018
		col	1.906	0.178	2.084	14.533	37.625	52.159
	(any decomp, any decomp)							
	16×2	row	1.655	0.104	1.759	14.016	19.418	33.434
		col	1.584	0.039	1.624	13.569	2.032	15.601
	2×16	row	1.588	0.040	1.628	13.018	1.989	15.007
		col	1.654	0.101	1.755	13.849	19.481	33.330
	8×4	row	1.545	0.068	1.614	13.489	10.623	24.112
		col	1.523	0.051	1.575	13.509	5.765	19.274
	4×8	row	1.525	0.055	1.580	13.132	5.768	18.899
		col	1.549	0.068	1.617	13.591	10.611	24.202

Table 21: PRODUCT, 2 dim. array, reduction along a dimension, array size = 512×512

Processors	Decomposition	Reduction Along	iPSC/860 time (ms)			iPSC/2 time (ms)		
			t_{calc}	t_{comm}	t_{total}	t_{calc}	t_{comm}	t_{total}
4	(any decomp, X)	row	38.408	10.020	48.428	460.216	29.088	489.304
		col	38.252	0.068	38.320	450.346	0.290	450.637
	(X, any decomp)	row	38.174	0.068	38.242	439.006	0.325	439.331
		col	38.398	10.068	48.467	443.467	28.020	471.487
	(any decomp, any decomp)							
	2×2	row	38.242	5.137	43.379	446.917	7.835	454.752
		col	38.252	5.137	43.389	446.855	7.393	454.248
8	(any decomp, X)	row	19.463	12.026	31.489	228.907	44.385	273.292
		col	19.253	0.078	19.331	225.219	0.296	225.516
	(X, any decomp)	row	19.180	0.073	19.253	215.346	0.353	215.699
		col	19.458	12.061	31.519	222.543	43.189	265.732
	(any decomp, any decomp)							
	4×2	row	19.302	5.317	24.619	223.031	15.355	238.387
		col	19.238	2.876	22.114	223.467	4.924	228.391
	2×4	row	19.238	2.856	22.095	219.008	5.301	224.309
		col	19.297	5.337	24.634	221.790	14.442	236.232
16	(any decomp, X)	row	9.983	14.751	24.734	110.527	59.273	169.799
		col	9.749	0.090	9.839	109.377	0.413	109.789
	(X, any decomp)	row	9.697	0.051	9.749	103.176	0.433	103.609
		col	9.976	14.800	24.775	108.585	58.934	167.519
	(any decomp, any decomp)							
	4×4	row	9.756	3.137	12.893	106.071	10.070	116.141
		col	9.753	3.159	12.913	108.060	10.045	118.106
	8×2	row	9.832	6.289	16.121	107.842	23.007	130.850
		col	9.705	1.741	11.445	108.751	4.316	113.067
	2×8	row	9.707	1.726	11.433	104.711	4.317	109.028
		col	9.834	6.279	16.113	107.688	22.897	130.584
32	(any decomp, X)	row	5.239	18.030	23.269	56.865	73.793	130.658
		col	4.999	0.068	5.067	54.702	0.302	55.005
	(X, any decomp)	row	4.938	0.062	5.000	51.549	0.278	51.827
		col	5.233	18.032	23.265	55.163	73.852	129.015
	(any decomp, any decomp)							
	16×2	row	5.083	7.682	12.765	54.660	30.114	84.774
		col	4.948	1.244	6.191	54.375	3.303	57.678
	2×16	row	4.944	1.235	6.179	51.698	3.350	55.048
		col	5.081	7.679	12.760	54.137	30.094	84.232
	8×4	row	5.002	3.879	8.882	53.442	14.500	67.942
		col	4.965	2.128	7.092	54.035	7.203	61.238
	4×8	row	4.965	2.108	7.073	52.552	7.191	59.742
		col	5.004	3.888	8.892	53.831	14.554	68.385

Table 22: SUM, 1 dim. array, any decomposition

Array Size	Processors	iPSC/860 time (ms)			iPSC/2 time (ms)		
		t_{calc}	t_{comm}	t_{total}	t_{calc}	t_{comm}	t_{total}
64	1	0.021	0.000	0.021	0.160	0.000	0.160
	2	0.011	0.400	0.411	0.080	1.410	1.490
	4	0.007	0.809	0.816	0.045	2.805	2.850
	8	0.005	1.799	1.804	0.030	4.200	4.230
	16	0.003	1.647	1.650	0.027	5.601	5.629
	32	0.002	2.146	2.148	0.020	7.076	7.096
256	1	0.080	0.000	0.080	0.540	0.000	0.540
	2	0.042	0.405	0.447	0.280	1.410	1.690
	4	0.021	0.805	0.826	0.155	2.785	2.940
	8	0.012	1.800	1.812	0.082	4.162	4.245
	16	0.007	1.649	1.656	0.046	5.585	5.631
	32	0.005	2.155	2.160	0.034	7.042	7.076
1K	1	0.313	0.000	0.313	2.140	0.000	2.140
	2	0.157	0.418	0.575	1.080	1.410	2.490
	4	0.080	0.799	0.879	0.540	2.795	3.335
	8	0.040	1.824	1.864	0.282	4.187	4.470
	16	0.021	1.754	1.775	0.149	5.577	5.726
	32	0.012	2.214	2.226	0.081	7.045	7.126
4K	1	1.406	0.000	1.406	8.500	0.000	8.500
	2	0.631	0.501	1.132	4.260	1.440	5.700
	4	0.313	0.849	1.162	2.135	2.820	4.955
	8	0.157	1.843	2.000	1.072	4.215	5.287
	16	0.079	1.727	1.806	0.546	5.582	6.129
	32	0.040	2.225	2.265	0.279	7.039	7.319
16K	1	5.718	0.000	5.718	34.04	0.000	34.04
	2	2.859	0.503	3.362	17.02	1.700	18.72
	4	1.408	0.913	2.321	8.515	2.905	11.42
	8	0.631	1.984	2.615	4.260	4.245	8.505
	16	0.312	1.814	2.125	2.135	5.630	7.765
	32	0.156	2.189	2.345	1.076	7.064	8.141
64K	1	22.89	0.000	22.89	148.2	0.000	148.2
	2	11.44	0.531	11.97	74.12	1.690	75.81
	4	5.720	0.928	6.648	34.03	3.415	37.45
	8	2.858	1.969	4.826	17.02	4.522	21.54
	16	1.408	1.763	3.171	8.510	5.721	14.23
	32	0.632	2.289	2.921	4.258	7.097	11.35
256K	1	91.49	0.000	91.49	593.1	0.000	593.1
	2	45.76	0.556	46.32	296.5	1.600	298.1
	4	22.89	0.954	23.85	148.2	3.040	151.3
	8	11.44	2.009	13.45	74.12	4.492	78.61
	16	5.719	1.796	7.515	34.04	6.219	40.26
	32	2.857	2.264	5.121	17.02	7.395	24.42

Table 23: SUM, 2 dim. array, reduction to scalar, any decomposition

Array Size	Processors	iPSC/860 time (ms)			iPSC/2 time (ms)		
		t_{calc}	t_{comm}	t_{total}	t_{calc}	t_{comm}	t_{total}
16×16	1	0.130	0.000	0.130	1.400	0.000	1.400
	2	0.071	0.408	0.479	0.700	1.430	2.130
	4	0.039	0.814	0.853	0.385	2.775	3.160
	8	0.023	1.219	1.242	0.222	4.167	4.390
	16	0.016	1.667	1.683	0.141	5.596	5.737
	32	0.009	2.154	2.163	0.081	7.097	7.177
32×32	1	0.505	0.000	0.505	5.440	0.000	5.440
	2	0.259	0.413	0.672	2.750	1.410	4.160
	4	0.136	0.835	0.971	1.400	2.775	4.175
	8	0.074	1.227	1.302	0.740	4.147	4.887
	16	0.046	1.670	1.715	0.421	5.561	5.982
	32	0.029	2.144	2.173	0.259	6.968	7.227
64×64	1	2.118	0.000	2.118	21.80	0.000	21.80
	2	1.016	0.500	1.516	10.86	1.450	12.31
	4	0.517	0.839	1.356	5.490	2.785	8.275
	8	0.271	1.239	1.510	2.785	4.177	6.962
	16	0.148	1.670	1.818	1.466	5.574	7.040
	32	0.087	2.176	2.263	0.823	6.984	7.807
128×128	1	8.486	0.000	8.486	88.16	0.000	88.16
	2	4.270	0.498	4.768	43.61	1.700	45.31
	4	2.143	0.928	3.071	21.71	2.905	24.61
	8	1.039	1.336	2.376	10.95	4.217	15.16
	16	0.539	1.708	2.247	5.555	5.610	11.16
	32	0.292	2.190	2.483	2.907	6.992	9.899
256×256	1	33.89	0.000	33.89	367.5	0.000	367.5
	2	16.96	0.556	17.51	182.0	1.680	183.7
	4	8.541	0.951	9.492	87.18	3.395	90.57
	8	4.315	1.329	5.644	43.42	4.432	47.86
	16	2.188	1.759	3.947	21.88	5.681	27.56
	32	1.085	2.244	3.329	11.09	7.047	18.13
512×512	1	135.3	0.000	135.3	1485	0.000	1485
	2	67.80	0.609	68.41	735.1	1.650	736.8
	4	33.97	0.981	34.95	364.0	3.090	367.1
	8	17.08	1.384	18.46	181.0	4.425	185.5
	16	8.635	1.762	10.40	86.82	6.202	93.02
	32	4.403	2.261	6.664	43.77	7.242	51.01
$1K \times 1K$	1	540.8	0.000	540.8	-	-	-
	2	270.5	0.632	271.1	2970	1.670	2972
	4	135.6	0.927	136.5	1470	3.030	1473
	8	67.99	1.452	69.44	728.1	4.395	732.5
	16	34.17	1.855	36.03	362.0	5.906	367.9
	32	17.26	2.277	19.54	180.1	7.316	187.4

Table 24: SUM, 2 dim. array, reduction along a dimension, array size = 64×64

Processors	Decomposition	Reduction Along	iPSC/860 time (ms)			iPSC/2 time (ms)		
			t_{calc}	t_{comm}	t_{total}	t_{calc}	t_{comm}	t_{total}
4	(any decomp, X)	row	0.630	0.000	0.630	8.940	0.000	8.940
		col	0.650	2.118	2.768	8.875	6.535	15.41
	(X, any decomp)	row	0.651	2.112	2.763	9.120	6.485	15.60
		col	0.629	0.000	0.629	8.940	0.000	8.940
	(any decomp, any decomp)							
2×2		row	0.696	0.988	1.685	9.345	2.690	12.03
		col	0.695	0.989	1.684	9.270	2.650	11.92
8	(any decomp, X)	row	0.316	0.000	0.316	4.447	0.000	4.447
		col	0.337	3.236	3.573	4.465	9.837	14.30
	(X, any decomp)	row	0.338	3.248	3.587	4.605	9.767	14.37
		col	0.315	0.000	0.315	4.485	0.000	4.485
	(any decomp, any decomp)							
4×2		row	0.380	0.507	0.888	4.830	1.737	6.567
		col	0.391	1.851	2.242	4.847	5.267	10.11
2×4		row	0.394	1.857	2.251	4.935	5.250	10.18
		col	0.377	0.507	0.884	4.830	1.720	6.550
16	(any decomp, X)	row	0.160	0.000	0.160	2.220	0.000	2.220
		col	0.184	4.443	4.627	2.267	13.20	15.47
	(X, any decomp)	row	0.185	4.423	4.608	2.340	13.21	15.55
		col	0.160	0.000	0.160	2.259	0.000	2.260
	(any decomp, any decomp)							
4×4		row	0.234	0.969	1.203	2.682	3.392	6.075
		col	0.231	0.976	1.207	2.659	3.385	6.044
8×2		row	0.222	0.470	0.692	2.594	1.572	4.166
		col	0.256	2.831	3.087	2.806	8.000	10.80
2×8		row	0.259	2.834	3.093	2.846	7.931	10.78
		col	0.219	0.474	0.693	2.604	1.564	4.167
32	(any decomp, X)	row	0.081	0.000	0.081	1.126	0.000	1.126
		col	0.109	5.689	5.798	1.189	16.66	17.85
	(X, any decomp)	row	0.108	5.693	5.801	1.217	16.61	17.83
		col	0.081	0.000	0.081	1.142	0.000	1.142
	(any decomp, any decomp)							
16×2		row	0.146	0.457	0.602	1.483	1.492	2.975
		col	0.222	3.947	4.169	2.021	10.90	12.92
2×16		row	0.226	3.939	4.165	2.049	10.85	12.90
		col	0.143	0.450	0.593	1.497	1.488	2.985
8×4		row	0.155	0.915	1.070	1.567	3.090	4.657
		col	0.175	1.530	1.705	1.715	5.094	6.809
4×8		row	0.179	1.534	1.713	1.725	5.079	6.805
		col	0.152	0.915	1.068	1.562	3.082	4.644

Table 25: SUM, 2 dim. array, reduction along a dimension, array size = 256×256

Processors	Decomposition	Reduction Along	iPSC/860 time (ms)			iPSC/2 time (ms)		
			t_{calc}	t_{comm}	t_{total}	t_{calc}	t_{comm}	t_{total}
4	(any decomp, X)	row	15.87	0.000	15.87	144.6	0.000	144.6
		col	10.76	3.762	14.52	142.9	14.47	157.4
	(X, any decomp)	row	15.78	3.909	19.69	2560	53.89	2614
		col	10.68	0.000	10.68	2397	0.000	2397
	(any decomp, any decomp)							
2×2		row	15.93	1.366	17.30	146.0	5.405	151.4
		col	10.79	1.305	12.10	144.1	5.160	149.3
8	(any decomp, X)	row	7.905	0.000	7.905	71.98	0.000	71.98
		col	5.422	5.818	11.24	71.38	21.28	92.67
	(X, any decomp)	row	7.814	5.952	13.77	1275	82.22	1357
		col	5.340	0.000	5.340	1198	0.000	1198
	(any decomp, any decomp)							
4×2		row	7.952	1.055	9.007	72.77	3.782	76.55
		col	5.459	2.638	8.097	71.96	9.412	81.37
2×4		row	7.927	2.659	10.59	73.31	9.435	82.74
		col	5.434	1.088	6.522	72.32	3.777	76.09
16	(any decomp, X)	row	3.768	0.000	3.768	35.71	0.000	35.71
		col	2.736	7.936	10.67	35.23	28.31	63.54
	(X, any decomp)	row	3.745	7.961	11.71	633.0	110.9	743.9
		col	2.652	0.000	2.652	599.3	0.000	599.3
	(any decomp, any decomp)							
4×4		row	3.858	2.182	6.040	36.59	6.653	43.24
		col	2.753	2.172	4.925	36.14	6.725	42.87
8×2		row	3.851	1.000	4.851	36.33	2.876	39.21
		col	2.788	4.088	6.876	36.24	13.73	49.97
2×8		row	3.862	4.106	7.968	36.95	13.17	50.12
		col	2.734	0.982	3.716	36.28	2.912	39.20
32	(any decomp, X)	row	1.281	0.000	1.281	17.70	0.000	17.70
		col	1.349	10.10	11.45	17.61	35.03	52.64
	(X, any decomp)	row	1.367	10.16	11.53	308.1	139.3	447.4
		col	1.259	0.00	1.259	299.7	0.000	299.6
	(any decomp, any decomp)							
16×2		row	1.366	0.611	1.977	18.22	1.759	19.98
		col	1.442	5.699	7.141	18.50	18.36	36.86
2×16		row	1.468	5.676	7.144	19.09	18.31	37.40
		col	1.340	0.563	1.903	18.34	1.775	20.11
8×4		row	1.378	1.992	3.370	18.43	5.361	23.79
		col	1.384	3.447	4.831	18.43	10.01	28.44
4×8		row	1.409	3.468	4.877	18.68	9.980	28.66
		col	1.354	1.929	3.283	18.30	5.445	23.75

Table 26: SUM, 2 dim. array, reduction along a dimension, array size = $1K \times 1K$

Processors	Decomposition	Reduction Along	iPSC/860 time (ms)			iPSC/2 time (ms)		
			t_{calc}	t_{comm}	t_{total}	t_{calc}	t_{comm}	t_{total}
4	(any decomp, X)	row	293.9	0.000	293.9	2431	0.000	2431
		col	170.9	15.19	186.1	2363	54.12	2417
	(X, any decomp)	row	395.0	15.65	410.6	2560	53.89	2614
		col	170.7	0.000	170.7	2397	0.000	2397
	(any decomp, any decomp)							
2×2		row	326.6	4.122	330.7	2483	13.68	2497
		col	170.9	3.901	174.8	2379	13.66	2393
8	(any decomp, X)	row	136.4	0.000	136.4	1198	0.000	1198
		col	85.58	23.31	108.9	1174	82.85	1257
	(X, any decomp)	row	187.5	23.67	211.1	1275	82.22	1357
		col	85.32	0.000	85.32	1198	0.000	1198
	(any decomp, any decomp)							
4×2		row	144.3	2.090	146.4	1214	7.102	1221
		col	85.55	7.722	93.27	1182	27.17	1209
2×4		row	158.5	8.002	166.5	1239	27.12	1266
		col	85.51	1.980	87.49	1190	7.065	1197
16	(any decomp, X)	row	64.41	0.000	64.41	593.6	0.000	593.6
		col	42.92	31.78	74.71	586.0	111.3	697.3
	(X, any decomp)	row	66.68	32.24	98.91	633.0	110.9	744.0
		col	42.66	0.000	42.66	599.3	0.000	599.3
	(any decomp, any decomp)							
4×4		row	65.18	4.117	69.29	606.2	13.85	620.1
		col	42.84	3.860	46.70	591.2	13.72	604.9
8×2		row	64.73	1.478	66.21	598.9	4.596	603.4
		col	42.91	11.87	54.78	587.7	41.27	629.0
2×8		row	66.10	12.23	78.33	617.8	41.37	659.1
		col	42.78	1.346	44.12	595.2	4.597	599.8
32	(any decomp, X)	row	31.76	0.000	31.76	293.9	0.000	293.9
		col	21.59	40.05	61.64	289.1	139.5	428.6
	(X, any decomp)	row	32.27	40.14	72.42	308.1	139.3	447.4
		col	21.33	0.000	21.33	299.7	0.000	299.7
	(any decomp, any decomp)							
16×2		row	31.82	1.102	32.93	296.7	3.459	300.1
		col	21.62	16.07	37.69	293.9	56.05	350.0
2×16		row	32.19	16.32	48.51	305.8	56.13	361.9
		col	21.43	1.092	22.53	297.8	3.428	301.2
8×4		row	31.86	2.713	34.57	298.8	8.856	307.7
		col	21.52	5.957	27.48	294.2	20.88	315.1
4×8		row	31.91	5.995	37.90	301.7	20.93	322.6
		col	21.46	2.662	24.12	295.9	8.844	304.7

5 Array Manipulation Functions

5.1 CSHIFT

The Fortran 90D specification for CSHIFT is given in figure 6. The algorithm for CSHIFT for a one-dimensional array is given in figures 7 and 8. The algorithm consists of two parts: one for sending messages and one for receiving messages. Within each part, there are separate parts for positive and negative shift factor. If the shift factor is positive, the array is shifted to the right and elements coming off the edge of the rightmost processor are shifted to the beginning of the array in the leftmost processor on the grid. On the other hand, if the shift factor is negative, the array is shifted to the left and elements coming off the edge of the leftmost processor are shifted to the end of the array in the rightmost processor on the grid. The shift factor can be greater than the size of the local array. As can be observed from the algorithm, a processor needs to send messages to at the most two processors. Similarly a processor needs to receive messages from at the most two processors. The algorithm shows how these messages are packed and how local arrays are shifted to make room for the incomming messages. A temporary array is used to save the elements coming of the edges. This array needs to be used for performing a circular shift if the number of processors is only 1. For 2 dimensional CSHIFT, the shift can be performed along dimension 1 or dimension 2. The shift factor is specified by a shift vector whose elements can have different values. Furthermore, the values of that vector can be positive or negative. A combination of positive and negative values is allowed as long as the same dimension is used. For 2 dimensional CSHIFT, two algorithms have been written. The first algorithm is for the first dimension and the second algorithm is for the second dimension. These algorithms are shown in figure 9 and figure 10. Note that these algorithms pack different number of array elements from a row or column, depending upon the value of the elements of the shift vector. All the packed array elements are packed in only one message for either positive or negative values of shift. Therefore, each processor needs to send and receive only one message, for positive and negative directions, instead of generating messages for each row or each column. Since these algorithms assume that the value of shift in each element of the shift vector is less than the size of the local array, the algorithms for 2d cshift cannot directly be called if the amount of shift is greater than the local array size. In that case, a front end routine has been written which invokes these algorithms a number of times if required.

```
Begin
   if (SHIFT is negative) then
temp = -SHIFT
dest1 = (lb - 1 + temp)/local_size
destPE1 = (PE on the left side of the  grid x-axis at distance dest1)
if (destPE1 is not equal to thisPE) then
    L1 = MOD(temp, loc_size) + 1
    U1 = loc_size
    length1 = U1 - L1 +1
    send a message to destPE1 of length length1 starting from L1
end if
dest2 = (ub - 1 + temp)/local_size
destPE2 = (PE on the left side of the  grid x-axis at distance dest2)
if (destPE2 is not equal to thisPE and
    destPE2 is not equal to destPE1) then
    L2 = lb
    U2 = MOD(temp -1 , loc_size) + 1
    length2 = U2 - L2 +1
    send a message to destPE2 of length length2 starting from L2
end if
   else if (SHIFT is positive) then
dest1 = (lb - 1 + SHIFT)/local_size
destPE1 = (PE on the right side of the  grid x-axis at distance dest1)
if (destPE1 is not equal to thisPE) then
    L1 = loc_size
    U1 = MOD(global_size - 1 - SHIFT, loc_size) + 1
    length1 = U1 - L1 +1
    send a message to destPE1 of length length1 starting from L1
end if
dest2 = (ub - 1 + temp)/local_size
destPE2 = (PE on the left side of the  grid x-axis at distance dest2)
if (destPE2 is not equal to thisPE and
    destPE2 is not equal to destPE1) then
    L2 = MOD(global_size  - SHIFT, loc_size) + 1
    U2 = loc_size
    length2 = U2 - L2 +1
    send a message to destPE2 of length length2 starting from L2
end if
   end if
End
```

Figure 7: Algorithm for 1D CSHIFT

```
BEGIN
   if (SHIFT is negative) then
        in = lb
        temp = -SHIFT
            if (temp is less than loc_size) then
                        Store the array(1: temp) in a temporary array
                        Perform local shift on array(1: temp)
            endif
        src1 = (lb - 1 + temp)/local_size
        srcPE1 = (PE on the rigth side of the  grid x-axis at distance src1)
        if (srcPE1 is not equal to thisPE) then
            receive a message from srcPE1 of length length1
            and store it in array starting at in
            in = in + len
        end if
        src2 = (ub - 1 + temp)/local_size
        srcPE2 = (PE on the right side of the  grid x-axis at distance src2)
        if (srcPE2 is not equal to thisPE and srcPE2 is not equal to destPE1) then
            receive a message from srcPE2 of length length2
            and store it in array starting at in
        if (srcPE1 is equal to thisPE and srcPE2 is equal to thisPE) then
            in = in - 1
            put temporary array in the local array
        end if
   else if (SHIFT is positive) then
        in = lb
            if (SHIFT is less than loc_size) then
                        Store the array(ub-SHIFT+1: ub) in a temporary array
                        Perform local shift on array(SHIFT: ub)
            endif
         src1 = (lb - 1 + SHIFT)/local_size
         srcPE1 = (PE on the right side of the  grid x-axis at distance src1)
         if (srcPE1 is not equal to thisPE) then
                  receive a message from srcPE1 of length length1
                  and store it in array starting at in
         end if
         src2 = (ub - 1 + temp)/local_size
         srcPE2 = (PE on the left side of the  grid x-axis at distance src2)
         if (srcPE2 is not equal to thisPE and srcPE2 is not equal to destPE1) then
                  in = lb
                  receive a message from srcPE2 of length length2
                  and store it in array starting at in
         if (srcPE1 is equal to thisPE and srcPE2 is equal to thisPE) then
                  put temporary array in the local array
         end if
    end if
END
```

Figure 8: Algorithm for 1D CSHIFT, for receiving messages

```
Begin
    len = 0
    do i = lb2, ub2
     if (direc(i) .LT. 0) then
           do j = 0, shift(i) -1
                len = len + 1
           send_buff(len) = array(lb1+j,i)
            end do
            do k = lb1+shift(i), ub1
                array(k-shift(i),i) = array(k,i)
            end do
         end if
    end do
    if (len .GT. 0) then
         send send_buff of length len to up node
         receive rec_buff of length len from down node
         len = 1
         do i = lb2, ub2
              if (direc(i) .LT. 0) then
                 do k = 1, shift(i)
                     array(ub1-shift(i)+k,i) = rec_buff(len)
                     len = len + 1
                 end do
              end if
          end do
     end if
     len = 0
     do i = lb2, ub2
         if (direc(i) .GT. 0) then
             do j = 1, shift(i)
                 len = len + 1
                 send_buff(len) = array(ub1-shift(i)+j,i)
             end do
             do k = ub1-shift(i), lb1, -1
                  array(k+shift(i),i) = array(k,i)
             end do
          end if
      end do
      if (len .GT. 0) then
          send send_buff of length len to down node
          receive rec_buff of length len from up node
          len = 1
          do i = lb2, ub2
              if (direc(i) .GT. 0) then
                  do k = 0, shift(i) -1
              array(lb1+k,i)=rec_buff(len)
              len = len + 1
  end do
       end if
  end do
      end if
End
```

Figure 9: Algorithm for 2D CSHIFT with DIM = 1

```
Begin
    len = 0
    do i = lb1, ub1
        if (direc(i) .LT. 0) then
            do j = 0, shift(i) -1
                len = len + 1
                send_buff(len) = array(i, lb2+j)
            end do
            do k = lb2+shift(i), ub2
                array(i, k-shift(i)) = array(i, k)
            end do
         end if
    end do
    if (len .GT. 0) then
         send send_buff of length len to left node
         receive rec_buff of length len from right node
         len = 1
         do i = lb1, ub1
             if (direc(i) .LT. 0) then
                 do k = 1, shift(i)
                     array(i, ub2-shift(i)+k) = rec_buff(len)
                     len = len + 1
                 end do
             end if
          end do
    end if
    len = 0
    do i = lb1, ub1
        if (direc(i) .GT. 0) then
            do j = 1, shift(i)
        len = len + 1
send_buff(len) = array(i, ub2-shift(i)+j)
    end do
    do k = ub2-shift(i), lb2, -1
                array(i, k+shift(i)) = array(i, k)
            end do
         end if
    end do
    if (len .GT. 0) then
        send send_buff of length len to right node
        receive rec_buff of length len from left node
len = 1
do i = lb1, ub1
    if (shift(i) .GT. 0) then
       do k = 0, shift(i) -1
           array(i,lb2+k)=rec_buff(len)
           len = len + 1
       end do
    end if
 end do
    end if
End
```

Figure 10: Algorithm for 2D CSHIFT with DIM = 2

5.2 EOSHIFT

The Fortran 90D specification for EOSHIFT is given in figure 11. The algorithm for CSHIFT for a one-dimensional array is given in figure 12. Again, the algorithm consists of two parts: one for sending messages and one for receiving messages. Then within each part, there are separate parts for positive and negative shift factor. If the shift factor is positive and the array is shifted to the right and boundary values are shifted in the beginning of the array in the leftmost processor on the grid. On the other hand, if the shift factor is negative and the array is shifted to the left and boundary values are inserted at the end of the array in the rightmost processor on the grid. The shift factor can be greater than the size of the local array. As can be noticed from figure 12, the algorithm for the sending processors is the same as that of 1 dimensional CSHIFT algorithm. Also, a processor needs to receive messages from at the most two processors. However, the processors at the extreme edges of the processor grid fill their array sections with boundary values, depending upon the direction of shift.

- *Syntax*: **EOSHIFT(ARRAY, SHIFT, BOUNDARY, DIM)**
- *Optional Arguments*: BOUNDARY, MASK
- *Description*: Performs an end-off shift on an array expression of rank one or perform an end-off shift on all the complete rank one sections along a given dimension of an array expression of rank two or greater. Elements are shifted out at one end of a section and copies of a boundary value are shifted in at the other end. Different sections may have different boundary values and may be shifted by different amounts and in different directions.
- *Arguments*:

  1. ARRAY: may be of any type. It must not be scalar.
  2. SHIFT: must be of type integer and must be scalar if ARRAY has rank one; otherwise, it must be scalar or of rank $(d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n)$ where $(d_1, d_2, ..., d_n)$ is the shape of the array.
  3. DIM (optional): must be of the same type and type parameter as ARRAY and must be scalar if ARRAY has rank one; otherwise, it must be either a scalar or of rank $n-1$ and of shape $(d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n)$. BOUNDARY may be omitted for the data types in the following table and, in this case, it is as if it were present with the scalar values shown.

```
Type of ARRAY      Value of BOUNDARY
------------------------------------
Integer            0
Real0.             0
Complex            (0.0, 0.0)
Logical            false
Character (len)    len blanks
```

  4. DIM (optional): must be scalar and of type integer with a value in the range $1 \leq \text{DIM} \leq n$, where n is the rank of ARRAY. If DIM is omitted, it is as if it were present with the value 1.

The result is of the same type and type parameter as ARRAY and has the shape of ARRAY.

Figure 11: FORTRAN 90D Specification for EOSHIFT

```
Begin
                           /* Sending side */
      Same as in the case of CSHIFT except that
      if (SHIFT is negative) then
           if the horizontal axis of sending processor is less than the
   horizontal axis of the receiving processor on the processors grid,
   then do not send any message.
                           /* Receiving side*/
           if the horizontal axis of receiving processor is greater than the
   horizontal axis of the sending processor on the processors grid,
           then do not receive any message.
           fill the array with BOUNDARY values.
      else if (SHIFT is positive) then
                           /* Sending side */
           if the horizontal axis of sending processor is greater than the
   horizontal axis of the receiving processor on the processors grid,
   then do not send any message.
                           /* Receiving side*/
           if the horizontal axis of receiving processor is less than the
   horizontal axis of the sending processor on the processors grid,
           then do not receive any message.
           fill the array with BOUNDARY values.
      end if

End
```

Figure 12: Algorithm for 1d EOSHIFT

- *Syntax*: **TRANSPOSE(MATRIX)**
- *Description*: Transpose of an array of rank two.
- *Argument*: MATRIX may be of any type but must have rank two.

The result is an array of the same type and type parameters as MATRIX and with rank two and shape (n, m) where (m, n) is the shape of MATRIX.

Figure 13: FORTRAN 90D SPECIFICATION FOR TRANSPOSE

5.3 Transpose

The Fortran 90D specification for Transpose is given in figure 13. We have implemented TRANSPOSE assuming that the matrix is distributed as (BLOCK,BLOCK) and the number of processors along the rows is a multiple of the number of processors along the columns.

Let A be the local array corresponding to the matrix whose transpose is to be determined and let A_trans be the local array corresponding to the transpose of the matrix. A *virtual grid* is defined so that the number of *virtual processors* along both dimensions is the same. The virtual dimensions of the matrix A belonging to this *virtual node* are defined on the basis of this virtual grid. The transpose of the resultant matrix for each virtual processor is called A_trans_virtual and it is used to calculate A_trans as follows:

Each node calculates, in sequence, A_trans_virtual for each virtual sub-block of the matrix A. The coordinates of the *destination virtual processor* are then determined, from which we find the *real coordinates* of the destination node. This is used to find the actual processor number of the destination. For example, the first block in node p may have virtual coordinates (i, j). This implies that the A_trans_virtual calculated for this block has to be transmitted to virtual processor (j, i), which may be located in node q. So node p does a non-blocking write that sends A_trans_virtual to node q with virtual_dest_coord(2) being used as a unique type identifier that can be recognized at the receiving end. All A_trans_virtual-s are sent off sequentially in a loop.

Table 27: TRANSPOSE: 256 × 256 matrix

# of Nodes	Processor configuration	Matrix per node	iPSC/860 *(ms)*			iPSC/2 *(ms)*		
			t_{comp}	t_{comm}	t_{total}	t_{comp}	t_{comm}	t_{total}
1	1x1	256x256	58	–	58	387	–	387
2	2x1	256x128	51	46	97	316	170	486
4	2x2	128x128	14	63	77	177	34	211
4	4x1	256x64	30	48	78	186	76	262
8	4x2	128x64	19	32	51	82	57	139
8	8x1	256x32	23	35	58	87	64	151
16	4x4	64x64	21	15	36	51	21	72
16	16x1	256x16	9	26	35	59	54	113
16	8x2	128x32	18	14	32	53	36	89
32	8x4	64x32	9	15	24	32	20	52
32	32x1	256x8	10	24	34	45	77	122
32	16x2	128x16	7	14	21	32	33	65

Table 28: TRANSPOSE: 512 × 512 matrix

# of Nodes	Processor configuration	Matrix per node	iPSC/860 *(ms)*			iPSC/2 *(ms)*		
			t_{comp}	t_{comm}	t_{total}	t_{comp}	t_{comm}	t_{total}
1	1x1	512x512	299	–	299	1605	–	1605
2	2x1	512x256	265	179	444	722	182	804
4	2x2	256x256	122	200	322	629	132	761
4	4x1	512x128	86	251	337	584	107	691
8	4x2	256x128	58	136	194	316	171	487
8	8x1	512x64	52	173	225	336	234	570
16	4x4	128x128	62	95	157	161	81	242
16	16x1	512x32	36	80	116	196	128	324
16	8x2	256x64	55	64	119	203	118	321
32	8x4	128x64	16	54	70	112	56	168
32	32x1	512x16	6	67	73	93	130	223
32	16x2	256x32	27	33	60	101	75	176

Another loop then receives all the A_trans_virtual blocks that were sent by other nodes and puts each block at the appropriate place in the transpose matrix A_trans. This overlapping of computation and communication was found to give the best performance. The algorithm used for TRANSPOSE is given in figure 14.

The timings obtained for calculating the transpose of 256×256, 512×512, 1024×1024 and 2048×2048 matrices are given in tables 27, 28, 29 and 30. The graphs of speedup versus number of processors for different array sizes are given in figures 15 and 16. The graphs of speedup versus size of matrix for different grid configurations are given in figures 17 and 18.

```
Begin
  Define a VIRTUAL GRID so that the number of virtual processors
    along both dimensions of the array is the same;
  Calculate size of matrix A belonging to a virtual node;
  /* Send off all virtual blocks */
  fac_v = \# of processors along dim 1/ \# of processors along dim 2
  For i = 0 to (fac_v - 1)
    Begin
      Calculate A_trans_virtual, the transpose of the i-th virtual sub-block;
      Calculate virtual address, virtual_coord(2), of this block;
      Calculate virtual address, virtual_dest_coord(2), of the destination of
        this block;
      Calculate real coordinates of the destination, and from that, the
        processor number of the destination;
      Send this virtual block to the destination (real) processor;
    End
  /* Receive all virtual blocks */
  For i = 0 to (fac_v - 1)
    Begin
      Calculate virtual address, virtual_coord(2), of this block;
      Calculate virtual address, virtual_source_coord(2), of the source from
        which to receive this block;
      Calculate real coordinates of the source, and from that, the
        processor number of the source;
      Receive the i-th virtual block from the source node and assign it to
        the appropriate slot in A_trans;
    End
  return(A_trans);
End
```

Figure 14: Algorithm for TRANSPOSE

Table 29: TRANSPOSE: 1024 × 1024 matrix

# of Nodes	Processor configuration	Matrix per node	iPSC/860 *(ms)*			iPSC/2 *(ms)*		
			t_{comp}	t_{comm}	t_{total}	t_{comp}	t_{comm}	t_{total}
1	1x1	1024x1024	1508	–	1508			
2	2x1	1024x512	1532	716	2248			
4	2x2	512x512	648	843	1491			
4	4x1	1024x256	722	942	1664			
8	4x2	512x256	424	350	774	1315	654	1969
8	8x1	1024x128	495	514	1009	1419	683	2102
16	4x4	256x256	243	300	543	814	225	1039
16	16x1	1024x64	139	356	495	642	522	1164
16	8x2	512x128	227	244	471	605	547	1152
32	8x4	256x128	53	223	276	340	284	624
32	32x1	1024x32	43	212	255	381	309	690
32	16x2	512x64	88	159	247	396	256	652

Table 30: TRANSPOSE: 2048 × 2048 matrix

# of Nodes	Processor configuration	Matrix per node	iPSC/860 *(ms)*			iPSC/2 *(ms)*		
			t_{comp}	t_{comm}	t_{total}	t_{comp}	t_{comm}	t_{total}
8	4x2	1024x512	1464	2090	3554			
8	8x1	2048x256	1715	2326	4041			
16	4x4	512x512	660	1566	2226			
16	16x1	2048x128	1063	1123	2186			
16	8x2	1024x256	741	977	1718			
32	8x4	512x256	343	904	1247	1602	950	2552
32	32x1	2048x64	389	719	1108	1734	812	2546
32	16x2	1024x128	386	741	1127	1620	827	2447

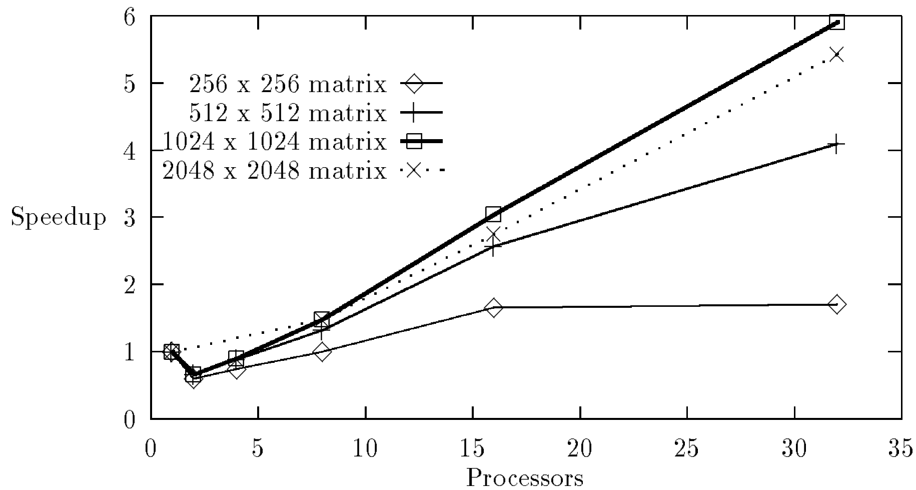

Figure 15: TRANSPOSE on iPSC/860

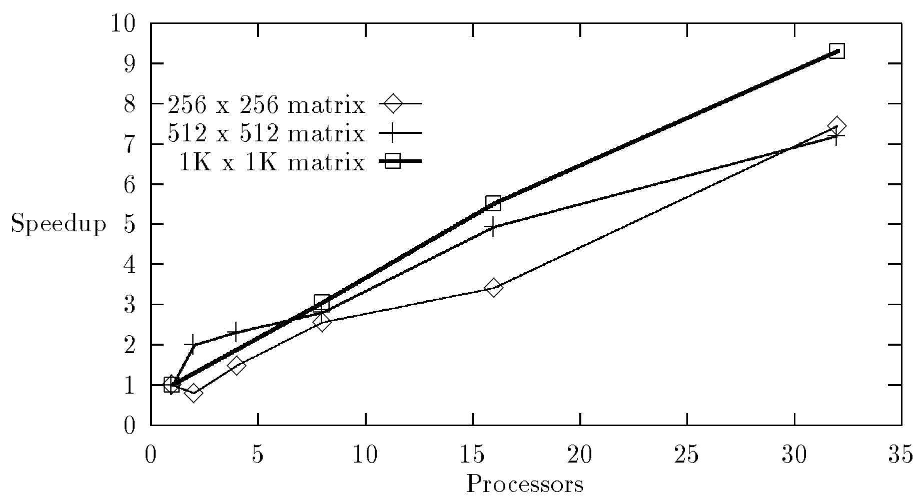

Figure 16: TRANSPOSE on iPSC/2

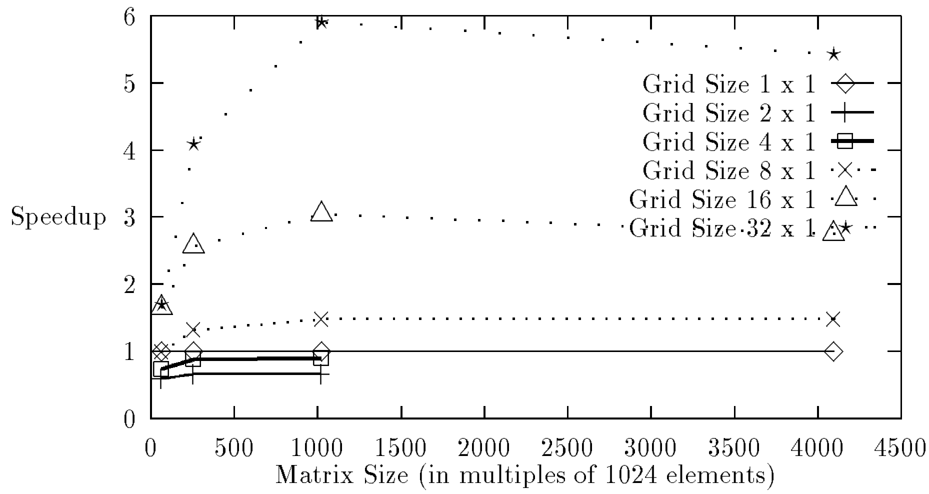

Figure 17: TRANSPOSE on iPSC/860, for different grid sizes

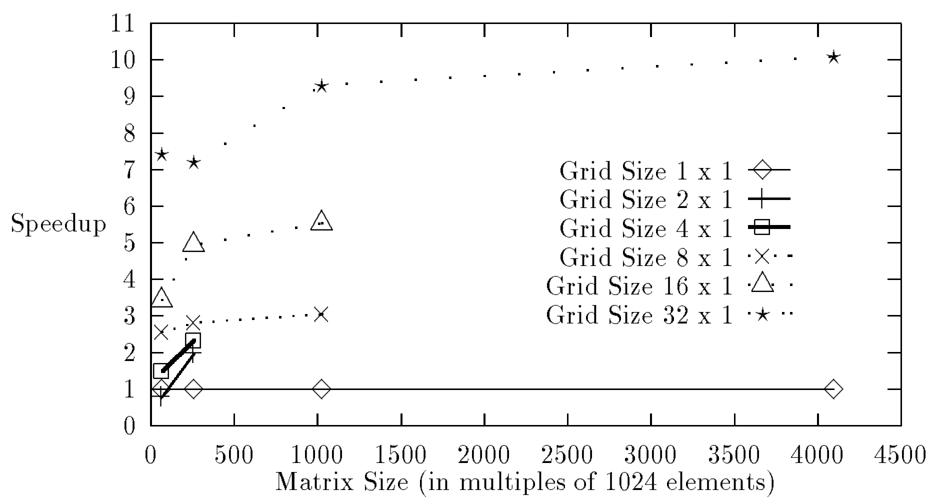

Figure 18: TRANSPOSE on iPSC/2, for different grid sizes

- *Syntax*: **MAXLOC(ARRAY, MASK), MINLOC(ARRAY, MASK)**
- *Optional Arguments*: MASK
- *Description (MAXLOC)*: Determines the location of the first element of ARRAY having the maximum value of the elements identified by MASK.
- *Description (MINLOC)*: Determines the location of the first element of ARRAY having the minimum value of the elements identified by MASK.
- *Arguments*:
  1. ARRAY: must be of type integer or real. It must not be scalar.
  2. MASK (optional): must be of type logical and must be conformable with ARRAY.

The result is of type integer. It is an array of rank one and of size equal to the rank of ARRAY. If there is more than one element with maximum (or minimum) value, it returns the location of the first element of the array in column major order, which has the maximum (or minimum) value.

Figure 19: FORTRAN 90D Specification for MAXLOC and MINLOC

- *Syntax*: **DOT_PRODUCT(VECTOR_A, VECTOR_B)**
- *Description*: Performs dot product of numeric or logical vectors.
- *Arguments*:
  1. VECTOR_A: must be of numeric type (integer, real, or complex) or of logical type. It must be array valued and of rank one.
  2. VECTOR_B: must be of the same type and size as VECTOR_A.

The result is scalar.

Figure 20: FORTRAN 90D SPECIFICATION FOR DOT_PRODUCT

6 Array Location Functions

6.1 MAXLOC and MINLOC

Figure 19 gives the Fortran 90D specification for MAXLOC. The implementation of MAXLOC is similar to that of MAXVAL, except that certain additional steps need to be taken to ensure that if there is more than one element with the maximum value, then the location of the first element of the array (in column major order) having the maximum value, is returned.

Each processor determines the first element of the local array having the maximum value among the elements in the local array. On the basis of the array decomposition information, it calculates the global coordinates of this element to determine the location of the element in the global array. All processors then perform a global operation to determine the maximum element and if it is not unique, the one which appears first in the global array. The address returned by MAXLOC is the global address of the maximum element. The time taken for MAXLOC will be slightly more than for MAXVAL because of the additional local to global index conversion.

- *Syntax*: **MATMUL(MATRIX_A, MATRIX_B)**
- *Description*: Performs matrix multiplication of numeric or logical matrices.
- *Arguments*:
  1. MATRIX_A: must be of numeric type (integer, real, or complex) or of logical type. It must be array valued and of rank one or two.
  2. MATRIX_B: must be of the same type as MATRIX_A and of rank one or two. If MATRIX_A has rank one, MATRIX_B must have rank two. If MATRIX_B has rank one, MATRIX_A must have rank two. The size of the first (or only) dimension of MATRIX_B must equal the size of the last (or only) dimension of MATRIX_A.

If MATRIX_A has shape (n, m) and MATRIX_B has shape (m, k), the result has shape (n, k). If MATRIX_A has shape (m) and MATRIX_B has shape (m, k), the result has shape (k). If MATRIX_A has shape (n, m) and MATRIX_B has shape (m), the result has shape (n).

Figure 21: FORTRAN 90D SPECIFICATION FOR MATMUL

7 Vector and Matrix Multiplication Functions

7.1 DOT_PRODUCT

The Fortran 90D specification for DOT_PRODUCT is given in figure 20. We have implemented this assuming that both the vectors have identical distributions. The problem then reduces to finding the dot product of the local vectors followed by a global sum operation. If the two vectors have different distributions, it is better to convert one of the vectors into the same distribution as the other and then perform the dot product. We are in the process of implementing efficient routines to convert from one distribution to another.

Table 31: DOT PRODUCT, any decomposition

Array Size	Processors	iPSC/860 time (ms)			iPSC/2 time (ms)		
		t_{calc}	t_{comm}	t_{total}	t_{calc}	t_{comm}	t_{total}
64	1	0.044	0.000	0.044	0.220	0.000	0.220
	2	0.022	0.403	0.425	0.120	1.420	1.540
	4	0.013	0.791	0.804	0.070	2.780	2.850
	8	0.009	1.205	1.213	0.045	4.197	4.242
	16	0.006	1.657	1.663	0.031	5.650	5.681
	32	0.005	2.166	2.171	0.027	7.042	7.069
256	1	0.150	0.000	0.150	0.880	0.000	0.880
	2	0.083	0.411	0.494	0.450	1.410	1.860
	4	0.044	0.816	0.860	0.230	2.770	3.000
	8	0.022	1.214	1.236	0.120	4.180	4.300
	16	0.013	1.700	1.713	0.068	5.610	5.677
	32	0.009	2.167	2.176	0.041	7.009	7.050
1K	1	0.564	0.000	0.564	3.500	0.000	3.500
	2	0.287	0.451	0.738	1.760	1.410	3.170
	4	0.151	0.826	0.977	0.885	2.775	3.660
	8	0.083	1.227	1.310	0.447	4.182	4.630
	16	0.044	1.699	1.743	0.229	5.597	5.826
	32	0.022	2.220	2.242	0.120	6.995	7.115
4K	1	2.575	0.000	2.575	13.96	0.000	13.96
	2	1.292	0.510	1.802	6.980	1.430	8.410
	4	0.565	0.952	1.517	3.495	2.790	6.285
	8	0.287	1.264	1.551	1.752	4.210	5.962
	16	0.151	1.690	1.841	0.879	5.631	6.510
	32	0.083	2.157	2.241	0.449	6.990	7.439
16K	1	10.30	0.000	10.30	55.84	0.000	55.84
	2	5.151	0.523	5.673	27.91	1.560	29.47
	4	2.577	0.946	3.523	13.95	2.875	16.82
	8	1.291	1.337	2.628	6.980	4.252	11.23
	16	0.565	1.790	2.355	3.495	5.676	9.171
	32	0.287	2.238	2.525	1.754	7.018	8.772
64K	1	41.20	0.000	41.20	223.5	0.000	223.5
	2	20.60	0.559	21.16	111.8	1.630	113.4
	4	10.30	0.965	11.26	55.83	3.085	58.91
	8	5.150	1.365	6.515	27.91	4.380	32.29
	16	2.577	1.803	4.380	13.95	5.739	19.69
	32	1.291	2.290	3.581	6.980	7.086	14.06
256K	1	164.8	0.000	164.8	893.9	0.000	893.9
	2	82.39	0.569	82.96	447.0	1.620	448.6
	4	41.20	1.000	42.20	223.5	3.020	226.5
	8	20.60	1.392	21.99	111.7	4.467	116.2
	16	10.30	1.792	12.09	55.83	5.954	61.79
	32	5.149	2.294	7.443	27.91	7.246	35.15

7.2 MATMUL

The Fortran 90D specification for MATMUL is given in figure 21. We have implemented the communication efficient parallel matrix multiplication algorithms proposed by Fox et al [6] and Berntsen [2]. The former method requires a particular multiplicand sub-matrix to be broadcast to all the processors which are in the same row of the processor configuration, followed by multiplication and neighbor communication along the columns. In the latter method, initially matrices are redistributed such that only neighbor communication is necessary in subsequent steps. Theoretically, this algorithm reduces the asymptotic communication cost of $2(N^2/P^{1/2})\beta$ of the first algorithm to $3(N^2/P^{2/3})\beta$, where N is the matrix size, P the number of processors and β the communication cost per word. The two algorithms are given in figures 22 and 23.

If the processor configuration is not a perfect square, but one of the dimensions is a multiple of the other, the sub-matrices will not be square. In this case, virtual processors are created to get square sub-matrices. FORTRAN follows column major order to store arrays. The matrix multiplication kernel is a modified version of the conventional kernel to suit the column major ordering. Tables 32, 33, 34 and 35 show the performance of the two algorithms for possible processor configurations for several matrix sizes. The matrices are of double precision floating point numbers and the times in seconds. Both algorithms suffer when the number of processor is less due to cache misses. Super-linear speedup effects are due to cache performance. The communication cost is marginally less for the second algorithm for some of the processor configurations. The communication time varies for different configuration of processors for the same total number of processors because the buffering requirement varies for sending and receiving messages. The current implementation assumes that both matrices are block decomposed and are square matrices. The number of processors allocated along a dimension of the array must be a multiple of the other.

The graphs of speedup versus number of processors are given in figures 24 and 25. The speedup increases almost linearly with the number of processors.

```
C perform matrix multiplication C = AB
C A and B are of size NxN
C T – a temporary matrix
      subroutine MATMUL(Â, B̂, Ĉ, T̂)
C initialize matrix Ĉ
      Ĉ ← 0
      do i = 1, √N − 1
C broadcast matrix A along rows
        call bcast(Â, T̂)
        Ĉ ← Ĉ + T̂B̂
C roll matrix B̂ upwards
        call roll(B̂)
      end do
      end
```

Figure 22: Matrix multiplication Algorithm I

Table 32: MATMUL Algorithm I (double precision) on iPSC/860 (in sec.)

Processor	Matrix size					
	256 × 256		512 × 512		1024 × 1024	
Config.	t_{comp}	t_{comm}	t_{comp}	t_{comm}	t_{comp}	t_{comm}
1x1	24.0393	0.128	205.36	0.534	–	–
2x2	5.926	0.387	48.112	1.606	410.807	6.582
1x4	5.906	0.485	47.348	1.785	384.44	7.656
4x1	6.109	0.632	50.672	2.274	401.679	9.332
4x4	1.479	0.261	11.853	1.037	96.204	3.974
2x8	1.452	0.248	11.810	1.120	94.736	4.606
8x2	1.514	0.662	12.204	2.509	100.463	10.065
1x16	1.468	0.465	11.654	1.949	94.910	7.287
16x1	1.594	2.196	13.038	7.737	102.472	29.384
1x32	0.774	0.477	5.768	1.926	45.736	7.728
32x1	0.828	7.054	6.681	16.875	53.102	58.961
4x8	0.722	0.205	5.903	0.795	47.389	3.117
8x4	0.747	0.378	5.966	1.422	48.371	5.482
2x16	0.727	0.260	5.776	1.01	47.171	4.318
16x2	0.777	1.331	6.201	4.715	51.175	17.760

```
C perform matrix multiplication C = AB
C A and B are of size NxN
      subroutine MATMUL(Â, B̂, Ĉ, T̂)
C initialize matrix Ĉ
      Ĉ ← 0
C rearrange matrix Â left
      call rearrangeA(Â)
C rearrange matrix B̂ upwards
      call rearrangeB(B̂)
      do i = 1, √N − 1
C broadcast matrix A along rows
         Ĉ ← Ĉ + ÂB̂
C roll matrix Â left
         call roll_left(Â)
C roll matrix B̂ upwards
         call roll_up(B̂)
      end do
      end
```

Figure 23: Matrix multiplication Algorithm II

Table 33: MATMUL Algorithm I (double precision) on iPSC/2 (in sec.)

Processor Config.	Matrix size					
	256 × 256		512 × 512		1024 × 1024	
	t_{comp}	t_{comm}	t_{comp}	t_{comm}	t_{comp}	t_{comm}
1x1	394.843	0.949	–	–	–	–
2x2	97.021	1.257	789.005	5.163	–	–
1x4	95.161	1.145	776.520	4.314	–	–
4x1	98.698	3.332	791.663	13.500	–	–
4x4	23.766	0.681	194.019	2.665	1578.279	10.432
2x8	23.672	0.961	190.347	3.764	1552.078	14.995
8x2	24.077	1.968	197.396	7.546	1583.337	30.446
1x16	23.766	1.197	189.969	4.933	1523.227	18.842
16x1	24.525	5.738	199.445	19.242	1596.576	77.124
1x32	11.989	1.304	95.395	4.917	760.537	18.98
32x1	12.490	15.881	99.592	33.812	802.636	107.59
4x8	11.829	0.543	95.16	2.144	776.092	8.620
8x4	11.925	1.055	96.018	3.988	789.264	16.260
2x16	11.860	0.887	95.106	3.469	761.479	13.972
16x2	12.156	3.062	97.480	10.055	793.716	38.808

Table 34: MATMUL Algorithm II (double precision) on iPSC/2 (in sec.)

Processor Config.	Matrix size					
	256 × 256		512 × 512		1024 × 1024	
	t_{comp}	t_{comm}	t_{comp}	t_{comm}	t_{comp}	t_{comm}
1x1	395.0349	1.456	–	–	–	–
2x2	97.065	0.859	–	–	–	–
1x4	98.657	2.730	815.049	11.427	–	–
4x1	98.916	2.540	791.828	12.326	–	–
4x4	23.792	0.690	194.099	1.544	–	–
2x8	24.614	1.180	197.317	5.335	1630.089	20.944
8x2	24.146	1.070	197.832	4.038	1583.617	22.456
1x16	24.904	1.650	198.938	7.469	1590.920	57.279
16x1	24.678	1.682	199.764	6.444	1604.317	34.422
1x32	12.558	2.241	101.258	7.668	806.750	30.186
32x1	12.673	2.211	99.951	8.794	806.622	34.715
4x8	12.130	0.837	97.578	3.640	803.982	11.351
8x4	11.947	0.755	96.091	2.942	790.163	11.731
2x16	12.346	1.311	99.179	5.446	805.000	16.984
16x2	12.50	1.149	99.177	5.449	793.993	18.467

Table 35: MATMUL Algorithm II (double precision) on iPSC/860 (in sec.)

Processor	Matrix size					
	256 × 256		512 × 512		1024 × 1024	
Config.	t_{comp}	t_{comm}	t_{comp}	t_{comm}	t_{comp}	t_{comm}
1x1	23.989	0.125	204.490	0.496	–	–
2x2	5.910	0.540	48.089	2.096	409.727	6.335
1x4	5.899	0.664	47.307	2.314	384.226	10.412
4x1	6.091	0.612	50.142	2.429	397.570	10.298
4x4	1.479	0.253	11.826	1.104	95.993	3.897
2x8	1.447	0.386	11.819	1.415	94.626	5.902
8x2	1.519	0.364	12.187	1.509	100.363	5.803
1x16	1.452	0.793	11.585	2.443	94.307	8.865
16x1	1.619	0.716	13.093	2.324	102.485	9.392
1x32	0.788	0.949	5.782	2.445	46.180	8.915
32x1	0.879	0.887	5.984	3.660	53.869	8.066
4x8	0.727	0.215	5.908	0.831	47.337	3.530
8x4	0.749	0.219	5.952	0.904	48.304	3.541
2x16	0.722	0.370	5.778	1.263	47.267	5.064
16x2	0.784	0.373	6.212	1.311	51.345	4.965

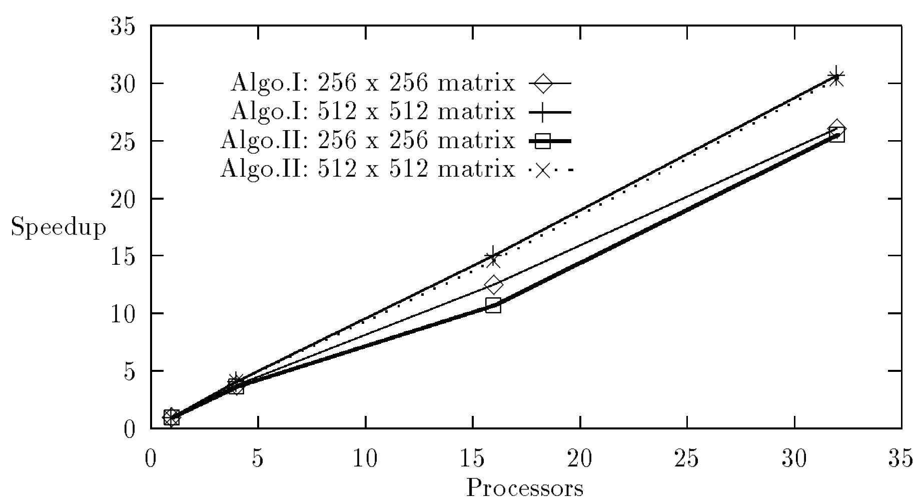

Figure 24: MATMUL on iPSC/860

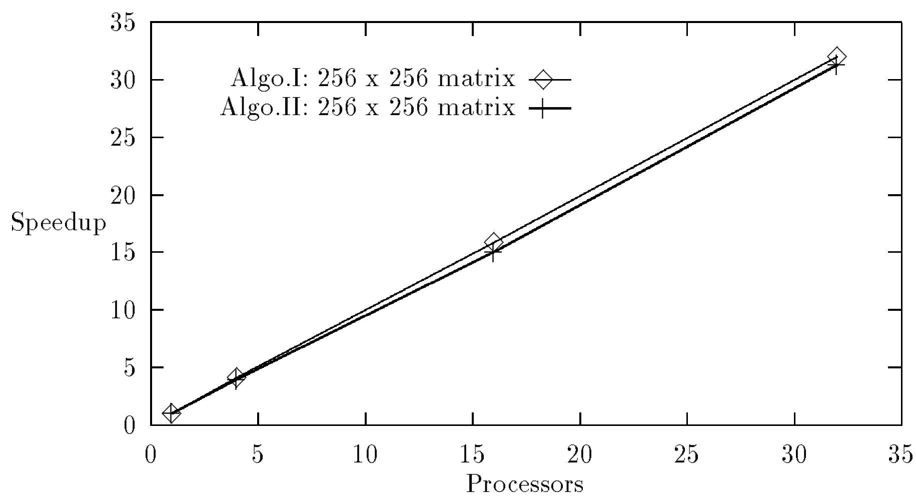

Figure 25: MATMUL on iPSC/2

References

[1] Ahmad, I., and M. Wu, *EXPRESS versus iPSC/2 Primitives: A Performance Comparison*, Technical Report, SCCS-77, Syracuse Center for Computational Science, April 1991. CRPC-TR91-147

[2] Berntsen. J, *Communication Efficient Matrix Multiplication on a Hypercube*, Parallel Computing, 1989, pp 335-342.

[3] Fox, G., The Architecture of Problems and Portable Parallel Software Systems, Technical Report SCCS-134, Syracuse Center for Computational Science. CRPC-TR91-172.

[4] Fox, G., *What have we learned from using real parallel machines to solve real problems?*, in G. Fox, editor, The Third Conference on Hypercube Concurrent Computers and Applications, Vol. 2, pp. 897-955, Jan. 1988.

[5] Fox, G., S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu, *Fortran D Language Specifications*, Technical Report COMP TR90-141, Rice University, Dec. 1990.

[6] Fox, G., S. Otto, and A. Hey, *Matrix Algorithms on a Hypercube I: Matrix multiplication*, Parallel Computing, 1987, pp. 17-31.

[7] Hiranandani, S., K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng, *An Overview of the Fortran D Programming System*, Technical Report COMP TR91-154, Rice University, March 1991.

[8] Hiranandani, S., K. Kennedy, and C. Tseng, *Compiler Support for Machine-Independent Parallel Programming in Fortran D*, Technical Report COMP TR91-149, Rice University, March 1991.

[9] Hiranandani, S., K. Kennedy, and C. Tseng, *Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines*, Proc. Supercomputing '91, pp. 86-100.

[10] Parasoft Corp., *Express Fortran User's Guide*, 1990.

[11] Parasoft Corp., *Express Fortran Reference Guide*, 1990.

[12] Wu, M., and G. Fox, *Compiling Fortran 90 Programs for Distributed Memory MIMD Parallel Computers*, Technical Report SCCS-88, Syracuse Center for Computational Science, April 1991. CRPC-TR91-126.

APPENDIX

General Syntax for Intrinsic Functions

```
name_dim_type_S(or V)_M(A, sizeof_A, A_INFO, other arguments, result, proclist)


name = function name
dim = 1 for 1 dimensional array
    = 2 for 2 dimensional array
type = I for integer
       R for real
       D for double
       L for logical
       C for complex
S - if the result is scalar, or V - if the result is vector
M - if mask is specified
S (or V) and M are optional
sizeof_A = lower and upper bound in each dimension. For example, for a two-
           dimensional array A(a1:a2,a3:a4), the arguments passed will be
           a1, a2, a3, a4.
proclist = list of participating processors (one-dimensional array). The first
           element of this list gives the number of participating processors.
           The list of participating processors follows this number. If all
           processors  are participating, it is not necessary to give the
           list. In this case, the first element of proclist must be set to
           -1, to indicate that all processors are participating.
other arguments - specified separately for each function below
```

Note that if the result is scalar, we have written a function. If the result is a vector, we have written a procedure.

Specific Syntax for Intrinsic Functions

(1) ALL_1_L(ARRAY, a1, a2, ARRAY_INFO, PROCLIST)

ALL operation reduced to a scalar from 1 dimensional integer array.

(2) ALL_2_L_S(ARRAY, a1, a2, a3, a4, ARRAY_INFO, PROCLIST)

ALL operation reduced to a scalar (result) from 2 dimensional integer array.

(3) ALL_2_L_V(ARRAY, a1, a2, a3, a4, ARRAY_INFO, DIM, RESULT, r1, r2, RESULT_INFO, PROCLIST)

ALL operation reduced to a vector (result) from 2 dimensional integer array, along dimension DIM. (DIM=2 for reduction along rows, DIM=1 for reduction along columns)

(4) ANY_1_L(ARRAY, a1, a2, ARRAY_INFO, PROCLIST)

ANY operation reduced to a scalar from 1 dimensional integer array.

(5) ANY_2_L_S(ARRAY, a1, a2, a3, a4, ARRAY_INFO, PROCLIST)

ANY operation reduced to a scalar (result) from 2 dimensional integer array.

(6) ANY_2_L_V(ARRAY, a1, a2, a3, a4, ARRAY_INFO, DIM, RESULT, r1, r2, RESULT_INFO, PROCLIST)

ANY operation reduced to a vector (result) from 2 dimensional integer array, along dimension DIM. (DIM=2 for reduction along rows, DIM=1 for reduction along columns)

(7) COUNT_1_L(ARRAY, a1, a2, ARRAY_INFO, PROCLIST)

COUNT operation reduced to a scalar from 1 dimensional integer array.

(8) COUNT_2_L_S(ARRAY, a1, a2, a3, a4, ARRAY_INFO, PROCLIST)

COUNT operation reduced to a scalar (result) from 2 dimensional integer array.

(9) COUNT_2_L_V(ARRAY, a1, a2, a3, a4, ARRAY_INFO, DIM, RESULT, r1, r2, RESULT_INFO, PROCLIST)

COUNT operation reduced to a vector (result) from 2 dimensional integer array, along dimension DIM. (DIM=2 for reduction along rows, DIM=1 for reduction along columns)

(10) CSHIFT_1_I(ARRAY, a1, a2, ARRAY_INFO, SHIFT, RESULT, r1, r2, RESULT_INFO, PROCLIST)

CSHIFT circular operation along dimension 1, with factor SHIFT. The argument SHIFT which is a scalar may be positive or negative.

(11) CSHIFT_2_I_S(ARRAY, a1, a2, a3, a4, ARRAY_INFO, SHIFT, DIM, RESULT, r1, r2, r3, r4, RESULT_INFO, PROCLIST)

CSHIFT operation on a 2D array along dimension DIM with shift specified as a scalar.

(12) CSHIFT_2_I_V(ARRAY, a1, a2, a3, a4, ARRAY_INFO, SHIFT, s1, s2, SHIFT_INFO, DIM, RESULT, r1, r2, r3, r4, RESULT_INFO, PROCLIST)

CSHIFT operation on a 2D array along dimension DIM with shift specified as a vector.

(13) DOT_PRODUCT_1_I(A, a1, a2, A_INFO, B, b1, b2, B_INFO, PROCLIST)

DOT PRODUCT operation reduced to a scalar (result) from two 1 dimensional integer arrays.

(14) EOSHIFT_1_I(ARRAY, a1, a2, ARRAY_INFO, SHIFT, RESULT, r1, r2, RESULT_INFO, PROCLIST)

EOSHIFT circular operation along dimension 1, with factor SHIFT. The argument shift which is a scalar may be positive or negative. Boundary is automatically assumed to be 0 if the array type is integer.

(15) EOSHIFTB_1_I(ARRAY, a1, a2, ARRAY_INFO, SHIFT, BOUNDARY, RESULT, r1, r2, RESULT_INFO, PROCLIST)

EOSHIFT circular operation along dimension 1, with factor SHIFT. The argument SHIFT, which is a scalar, may be positive or negative. In this case a scalar BOUNDARY must be provided.

(16) EOSHIFT_2_I_S(ARRAY, a1, a2, a3, a4, ARRAY_INFO, SHIFT, DIM, RESULT, r1, r2, r3, r4, RESULT_INFO, PROCLIST)

EOSHIFT operation on a 2D array along dimension DIM with shift specifed as a scalar and boundary assumed to be 0.

(17) EOSHIFT_2_I_V(ARRAY, a1, a2, a3, a4, ARRAY_INFO, SHIFT, s1, s2, SHIFT_INFO, DIM, RESULT, r1, r2, r3, r4, RESULT_INFO, PROCLIST)

EOSHIFT operation on a 2D array along dimension DIM with shift specifed as a vector and boundary assumed to be 0.

(18) EOSHIFT_B_S_2_I_S(ARRAY, a1, a2, a3, a4, ARRAY_INFO, SHIFT, BOUNDARY, DIM, RESULT, r1, r2, r3, r4, RESULT_INFO, PROCLIST)

EOSHIFT operation on a 2D array along dimension DIM with shift specifed as a scalar and boundary specified by scalar BOUNDARY.

(19) EOSHIFT_B_S_2_I_V(ARRAY, a1, a2, a3, a4, ARRAY_INFO, SHIFT, s1, s2, SHIFT_INFO, BOUNDARY, DIM, RESULT, r1, r2, r3, r4, RESULT_INFO, PROCLIST)

EOSHIFT operation on a 2D array along dimension DIM with shift specifed as a vector and boundary specified by scalar BOUNDARY.

(20) EOSHIFT_B_V_2_I_S(ARRAY, a1, a2, a3, a4, ARRAY_INFO, SHIFT, BOUNDARY, b1, b2, BOUND_INFO, dDIM, RESULT, r1, r2, r3, r4, RESULT_INFO, PROCLIST)

EOSHIFT operation on a 2D array along dimension DIM with shift specifed as a scalar and boundary specified by vector BOUNDARY.

(21) EOSHIFT_B_V_2_I_V(ARRAY, a1, a2, a3, a4, ARRAY_INFO, SHIFT, s1, s2, SHIFT_INFO, BOUNDARY, b1, b2, BOUND_INFO, dDIM, RESULT, r1, r2, r3, r4, RESULT_INFO, PROCLIST)

EOSHIFT operation on a 2D array along dimension DIM with shift specifed as a vector and boundary specified by vector BOUNDARY.

(22) MATMUL_2_I(A, a1, a2, a3, a4, A_INFO, B, b1, b2, b3, b4, B_INFO, RESULT, r1, r2, r3, r4, RESULT_INFO, PROCLIST)

Multiplication of matrices A and B (2 dimensional arrays).

(23) MAXLOC_1_I(ARRAY, a1, a2, ARRAY_INFO, RESULT, r1, r2, RESULT_INFO, PROCLIST)

MAXLOC operation for 1 dimensional integer array without mask.

(24) MAXLOC_1_I_M(ARRAY, a1, a2, ARRAY_INFO, MASK, m1, m2, MASK_INFO, RESULT, r1, r2, RESULT_INFO, PROCLIST)

MAXLOC operation for 1 dimensional integer array with mask.

(25) MAXLOC_2_I(ARRAY, a1, a2, a3, a4, ARRAY_INFO, RESULT, r1, r2, RESULT_INFO, PROCLIST)

MAXLOC operation for 2 dimensional integer array without mask.

(26) MAXLOC_2_I_M(ARRAY, a1, a2, a3, a4, ARRAY_INFO, MASK, m1, m2, m3, m4, MASK_INFO, RESULT, r1, r2, RESULT_INFO, PROCLIST)

MAXLOC operation for 2 dimensional integer array with mask.

(27) MAXVAL_1_I(ARRAY, a1, a2, ARRAY_INFO, PROCLIST)

MAXVAL operation reduced to a scalar from 1 dimensional integer array without mask.

(28) MAXVAL_1_I_M(ARRAY, a1, a2, ARRAY_INFO, MASK, m1, m2, MASK_INFO, PROCLIST)

MAXVAL operation reduced to a scalar from 1 dimensional integer array with mask.

(29) MAXVAL_2_I_S(ARRAY, a1, a2, a3, a4, ARRAY_INFO, PROCLIST)

MAXVAL operation reduced to a scalar (result) from 2 dimensional integer array without mask.

(30) MAXVAL_2_I_S_M(ARRAY, a1, a2, a3, a4, ARRAY_INFO, MASK, m1, m2, m3, m4, MASK_INFO, PROCLIST)

MAXVAL operation reduced to a scalar (result) from 2 dimensional integer array with mask.

(31) MAXVAL_2_I_V(ARRAY, a1, a2, a3, a4, ARRAY_INFO, DIM, RESULT, r1, r2, RESULT_INFO, PROCLIST)

MAXVAL operation reduced to a vector (result) from 2 dimensional integer array without mask, along dimension DIM. (DIM=2 for reduction along rows, DIM=1 for reduction along columns)

(32) MAXVAL_2_I_V_M(ARRAY, a1, a2, a3, a4, ARRAY_INFO, DIM, MASK, m1, m2, m3, m4, MASK_INFO, RESULT, r1, r2, RESULT_INFO, PROCLIST)

MAXVAL operation reduced to a vector (result) from 2 dimensional integer array with mask, along dimension DIM. (DIM=2 for reduction along rows, DIM=1 for reduction along columns)

(33) MINLOC_1_I(ARRAY, a1, a2, ARRAY_INFO, RESULT, r1, r2, RESULT_INFO, PROCLIST)

MINLOC operation for 1 dimensional integer array without mask.

(34) MINLOC_1_I_M(ARRAY, a1, a2, ARRAY_INFO, MASK, m1, m2, MASK_INFO, RESULT, r1, r2, RESULT_INFO, PROCLIST)

MINLOC operation for 1 dimensional integer array with mask.

(35) MINLOC_2_I(ARRAY, a1, a2, a3, a4, ARRAY_INFO, RESULT, r1, r2, RESULT_INFO, PROCLIST)

MINLOC operation for 2 dimensional integer array without mask.

(36) MINLOC_2_I_M(ARRAY, a1, a2, a3, a4, ARRAY_INFO, MASK, m1, m2, m3, m4, MASK_INFO, RESULT, r1, r2, RESULT_INFO, PROCLIST)

MINLOC operation for 2 dimensional integer array with mask.

(37) MINVAL_1_I(ARRAY, a1, a2, ARRAY_INFO, PROCLIST)

MINVAL operation reduced to a scalar from 1 dimensional integer array without mask.

(38) MINVAL_1_I_M(ARRAY, a1, a2, ARRAY_INFO, MASK, m1, m2, MASK_INFO, PROCLIST)

MINVAL operation reduced to a scalar from 1 dimensional integer array with mask.

(39) MINVAL_2_I_S(ARRAY, a1, a2, a3, a4, ARRAY_INFO, PROCLIST)

MINVAL operation reduced to a scalar (result) from 2 dimensional integer array without mask.

(40) MINVAL_2_I_S_M(ARRAY, a1, a2, a3, a4, ARRAY_INFO, MASK, m1, m2, m3, m4, MASK_INFO, PROCLIST)

MINVAL operation reduced to a scalar (result) from 2 dimensional integer array with mask.

(41) MINVAL_2_I_V(ARRAY, a1, a2, a3, a4, ARRAY_INFO, DIM, RESULT, r1, r2, RESULT_INFO, PROCLIST)

MINVAL operation reduced to a vector (result) from 2 dimensional integer array without mask, along dimension DIM. (DIM=2 for reduction along rows, DIM=1 for reduction along columns)

(42) MINVAL_2_I_V_M(ARRAY, a1, a2, a3, a4, ARRAY_INFO, DIM, MASK, m1, m2, m3, m4, MASK_INFO, RESULT, r1, r2, RESULT_INFO, PROCLIST)

MINVAL operation reduced to a vector (result) from 2 dimensional integer array with mask, along dimension DIM. (DIM=2 for reduction along rows, DIM=1 for reduction along columns)

(43) PRODUCT_1_I(ARRAY, a1, a2, ARRAY_INFO, PROCLIST)

PRODUCT operation reduced to a scalar from 1 dimensional integer array without mask.

(44) PRODUCT_1_I_M(ARRAY, a1, a2, ARRAY_INFO, MASK, m1, m2, MASK_INFO, PROCLIST)

PRODUCT operation reduced to a scalar from 1 dimensional integer array with mask.

(45) PRODUCT_2_I_S(ARRAY, a1, a2, a3, a4, ARRAY_INFO, PROCLIST)

PRODUCT operation reduced to a scalar (result) from 2 dimensional integer array without mask.

(46) PRODUCT_2_I_S_M(ARRAY, a1, a2, a3, a4, ARRAY_INFO, MASK, m1, m2, m3,

m4, MASK_INFO, PROCLIST)

PRODUCT operation reduced to a scalar (result) from 2 dimensional integer array with mask.

(47) PRODUCT_2_I_V(ARRAY, a1, a2, a3, a4, ARRAY_INFO, DIM, RESULT, r1, r2, RESULT_INFO, PROCLIST)

PRODUCT operation reduced to a vector (result) from 2 dimensional integer array without mask, along dimension DIM. (DIM=2 for reduction along rows, DIM=1 for reduction along columns)

(48) PRODUCT_2_I_V_M(ARRAY, a1, a2, a3, a4, ARRAY_INFO, DIM, MASK, m1, m2, m3, m4, MASK_INFO, RESULT, r1, r2, RESULT_INFO, PROCLIST)

PRODUCT operation reduced to a vector (result) from 2 dimensional integer array with mask, along dimension DIM. (DIM=2 for reduction along rows, DIM=1 for reduction along columns)

(49) SPREAD_1_I(src, src_lbo1, src_ubo1, src_info, dim, ncopies, dest, dest_lbo1, dest_ubo1, dest_lbo2, dest_ubo2, dest_info)

Creates a 2D array from a 1D array by adding a dimension.

(50) SPREAD_2_I(src, src_lbo1, src_ubo1, src_lbo2, src_ubo2, src_info, dim, ncopies, dest, dest_lbo1, dest_ubo1, dest_lbo2, dest_ubo2, dest_lbo3, dest_ubo3, dest_info)

Creates a 3D array from a 2D array by adding a dimension.

(51) SUM_1_I(ARRAY, a1, a2, ARRAY_INFO, PROCLIST)

SUM operation reduced to a scalar from 1 dimensional integer array without mask.

(52) SUM_1_I_M(ARRAY, a1, a2, ARRAY_INFO, MASK, m1, m2, MASK_INFO, PROCLIST)

SUM operation reduced to a scalar from 1 dimensional integer array with mask.

(53) SUM_2_I_S(ARRAY, a1, a2, a3, a4, ARRAY_INFO, PROCLIST)

SUM operation reduced to a scalar (result) from 2 dimensional integer array without mask.

(54) SUM_2_I_S_M(ARRAY, a1, a2, a3, a4, ARRAY_INFO, MASK, m1, m2, m3, m4, MASK_INFO, PROCLIST)

SUM operation reduced to a scalar (result) from 2 dimensional integer array with mask.

(55) SUM_2_I_V(ARRAY, a1, a2, a3, a4, ARRAY_INFO, DIM, RESULT, r1, r2, RESULT_INFO, PROCLIST)

SUM operation reduced to a vector (result) from 2 dimensional integer array without mask, along dimension DIM. (DIM=2 for reduction along rows, DIM=1 for reduction along columns)

(56) SUM_2_I_V_M(ARRAY, a1, a2, a3, a4, ARRAY_INFO, DIM, MASK, m1, m2, m3, m4, MASK_INFO, RESULT, r1, r2, RESULT_INFO, PROCLIST)

SUM operation reduced to a vector (result) from 2 dimensional integer array with mask, along dimension DIM. (DIM=2 for reduction along rows, DIM=1 for reduction along columns)

(57) TRANSPOSE_2_I(ARRAY, a1, a2, a3, a4, ARRAY_INFO, RESULT, r1, r2, r3, r4, RESULT_INFO, PROCLIST)

TRANSPOSE operation on a two dimensional array.