High-Performance Computing

COMP 40370

Alexey Lastovetsky
(B2.06, alexey.lastovetsky@ucd.ie)

Array Libraries

Array Libraries

- Function extensions of C and Fortran 77 with array or vector libraries
  - The libraries are supposed to be optimised for each particular computer
  - Regular compilers can be used => no need in dedicated optimising compilers
- One of the most well-known and well-designed array libraries is the Basic Linear Algebra Subprograms (BLAS)
  - Provides basic array operations for numerical linear algebra
  - Available for most modern VP and SP computers

BLAS

- All BLAS routines are divided into 3 main categories:
  - Level 1 BLAS addresses scalar and vector operations
  - Level 2 BLAS addresses matrix-vector operations
  - Level 3 BLAS addresses matrix-matrix operations

- Routines of Level 1 do
  - vector reduction operations
  - vector rotation operations
  - element-wise and combined vector operations
  - data movement with vectors

Level 1 BLAS

- A vector reduction operation
  - The addition of the scaled dot product of two real vectors x and y into a scaled scalar r

$$r \leftarrow \beta r + \alpha x^T y = \beta r + \alpha \sum_{i=0}^{n-1} x_i y_i$$

- The C interface of the routine implementing the operation is

```
void BLAS_ddot(
    enum blas_conj_type conj, int n, double alpha,
    const double *x, int incx, double beta,
    const double *y, int incy, double *r );
```

Level 1 BLAS (ctd)

- Other routines doing reduction operations
  - Compute different vector norms of vector x
  - Compute the sum of the entries of vector x
  - Find the smallest or biggest component of vector x
  - Compute the sum of squares of the entries of vector x

- Routines doing rotation operations
  - Generate Givens plane rotation
  - Generate Jacobi rotation
  - Generate Householder transformation

Level 1 BLAS (ctd)

- An element-wise vector operation
  - The scaled addition of two real vectors x and y

$$w \leftarrow \alpha x + \beta y$$

- The C interface of the routine implementing the operation is

```
void BLAS_dwaxpby(
      int n, double alpha, const double *x, int incx,
      double beta, const double *y, int incy,
      double *w, int incw );
```

- Function `BLAS_cwaxpby` does the same operation but on complex vectors

Level 1 BLAS (ctd)

- Other routines doing element-wise operations
  - Scale the entries of a vector *x* by the real scalar *1/a*
  - Scale a vector *x* by *a* and a vector *y* by *b*, add these two vectors to one another and store the result in the vector *y*
  - Combine a scaled vector accumulation and a dot product
  - Apply a plane rotation to vectors *x* and *y*

Level 1 BLAS (ctd)

- An example of data movement with vectors
  - The interchange of real vectors x and y
- The C interface of the routine implementing the operation is

```
void BLAS_dswap( int n, double *x, int incx,
                 double *y, int incy );
```

- Function **`BLAS_cswap`** does the same operation but on complex vectors

Level 1 BLAS (ctd)

- Other routines doing data movement with vectors
  - Copy vector *x* into vector *y*
  - Sort the entries of real vector *x* in increasing or decreasing order and overwrite this vector *x* with the sorted vector as well as compute the corresponding permutation vector *p*
  - Scale the entries of a vector *x* by the real scalar *1/a*
  - Permute the entries of vector *x* according to permutation vector *p*

Level 2 BLAS

- Routines of Level 2
  - Compute different matrix vector products
  - Do addition of scaled matrix vector products
  - Compute multiple matrix vector products
  - Solve triangular equations
  - Perform rank one and rank two updates
  - Some operations use symmetric or triangular matrices

Level 2 BLAS (ctd)

- To store matrices, the following schemes are used
  - Conventional column-based and row-based storage
  - Packed storage for symmetric or triangular matrices
  - Band storage for band matrices
- Conventional storage
  - An *nxn* matrix *A* is stored in a one-dimensional array a
    - a_{ij} `=> a[i+j*s]` (C, column-wise storage)
    - a_{ij} `=> a[j+i*s]` (C, row-wise storage)
  - If $s=n$, rows (columns) will be contiguous in memory
  - If $s>n$, there will be a gap of $(s-n)$ memory elements between two successive rows (columns)
  - Only significant elements of symmetric/triangular matrices need be set

Packed Storage

- Packed storage
  - The relevant triangle of a symmetric/triangular matrix is packed by columns or rows in a one-dimensional array
  - The upper triangle of an *nxn* matrix *A* may be stored in a one-dimensional array `a`
    - `a`$_{ij}$ `(i≤j) => a[j+i*(2*n-i-1)/2]`(C, row-wise storage)
- Example.

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ 0 & a_{11} & a_{12} \\ 0 & 0 & a_{22} \end{pmatrix} \quad => \quad a_{00} \quad a_{01} \quad a_{02} \quad a_{11} \quad a_{12} \quad a_{22}$$

Band Storage

- Band storage
  - A compact storage scheme for band matrices
- Consider a band storage scheme
  - An *mxn* band matrix *A* with *l* subdiagonals and *u* superdiagonals may be stored in a 2-dimensional array `A` with *l+u+*1 rows and *n* columns
  - Columns of matrix *A* are stored in corresponding columns of array `A`
  - Diagonals of matrix *A* are stored in rows of array `A`
    - a_{ij} => `A(u+i-j,j)` for $max(0,j-u) \le i \le min(m-1,j+l)$
- Example.

$$\begin{pmatrix} a_{00} & a_{01} & 0 & 0 & 0 \\ a_{10} & a_{11} & a_{12} & 0 & 0 \\ a_{20} & a_{21} & a_{22} & a_{23} & 0 \\ 0 & a_{31} & a_{32} & a_{33} & a_{34} \\ 0 & 0 & a_{42} & a_{43} & a_{44} \end{pmatrix} \Rightarrow \begin{matrix} * & a_{01} & a_{12} & a_{23} & a_{34} \\ a_{00} & a_{11} & a_{22} & a_{33} & a_{44} \\ a_{10} & a_{21} & a_{32} & a_{43} & * \\ a_{20} & a_{31} & a_{42} & * & * \end{matrix}$$

Level 2 BLAS (ctd)

- An example of matrix vector multiplication operation
  - The scaled addition of a real *n*-length vector *y*, and the product of a *general* real *mxn* matrix *A* and a real *n*-length vector *x*

$$y \leftarrow \alpha Ax + \beta y$$

- The C interface of the routine implementing this operation is

```
void BLAS_dgemv( enum blas_order_type order,
                 enum blas_trans_type trans, int m, int n,
                 double alpha, const double *a, int stride,
                 const double *x, int incx, double beta,
                    const double *y, int incy );
```

- Parameters

`order   =>  blas_rowmajor` or `blas_colmajor`

`trans   =>  blas_no_trans` (do not transpose *A*)

Level 2 BLAS (ctd)

- If matrix A is a general band matrix with l subdiagonals and u superdiagonals, the function

```
void BLAS_dgbmv( enum blas_order_type order,
                 enum blas_trans_type trans,
                 int m, int n, int l, int u,
                 double alpha, const double *a, int stride,
                 const double *x, int incx, double beta,
                    const double *y, int incy );
```

better uses the memory. It assumes that a *band storage scheme* is used to store matrix *A*.

Level 2 BLAS (ctd)

- Other routines of Level 2 perform the following operations

$$y \leftarrow \alpha Ax + \beta y \quad \text{where} \quad A = A^T$$

$$x \leftarrow \alpha Ax \quad \text{or} \quad x \leftarrow \alpha A^T x \quad \text{where } A \text{ is triangular}$$

$$y \leftarrow \alpha Ax + \beta Bx$$

$$x \leftarrow \beta A^T y \quad , \quad w \leftarrow \alpha Ax$$

$$x \leftarrow A^T y \quad , \quad w \leftarrow Az \quad \text{where } A \text{ is triangular}$$

  as well as many others
- For any matrix-vector operation with a specific matrix operand (triangular, symmetric, banded, etc.), there is a routine for each storage scheme that can be used to store the operand

Level 3 BLAS

- Routines of Level 3 do
  - $O(n^2)$ matrix operations
    - norms, diagonal scaling, scaled accumulation and addition
    - different storage schemes to store matrix operands are supported
  - $O(n^3)$ matrix-matrix operations
    - multiplication, solving matrix equations, symmetric rank *k* and *2k* updates
  - Data movement with matrices

Level 3 BLAS (ctd)

- An example of O(n2) matrix operation, which scales two real mxn matrices A and B and stores their sum in a matrix C, is $C \leftarrow \alpha A + \beta B$
- The C interface of the routine implementing this operation under assumption that the matrices A, B and C are of the general form, is

```
void BLAS_dge_add( enum blas_order_type order, int m, int n,
                   double alpha, const double *a, int stride_a,
                   double beta, const double *b, int stride_b,
                   double *c, int stride_c);
```

- There are other 15 routines performing this operation for different types and forms of the matrices A, B and C

Level 3 BLAS (ctd)

- An example of $O(n^3)$ matrix-matrix operation involving a real *mxn* matrix *A*, a real *nxk* matrix *B*, and a real *mxk* matrix *C* is

$$C \leftarrow \alpha AB + \beta C$$

- The C routine implementing the operation for matrices A, B and C in the general form is

```
void BLAS_dgemm( enum blas_order_type order,
                   enum blas_trans_type trans_a,
                   enum blas_trans_type trans_b,
                   int m, int n, int k, double alpha,
                   const double *a, int stride_a,
                   const double *b, int stride_c,
                   double beta, const double *c, int stride_c);
```

Level 3 BLAS (ctd)

- Data movement with matrices includes
  - Copying matrix $\boldsymbol{A}$ or its transpose with storing the result in matrix $\boldsymbol{B}$

    $$\boldsymbol{B} \leftarrow \boldsymbol{A} \quad \text{or} \quad \boldsymbol{B} \leftarrow \boldsymbol{A}^T$$

  - Transposition of a square matrix $\boldsymbol{A}$ with the result overwriting matrix $\boldsymbol{A}$

    $$\boldsymbol{A} \leftarrow \boldsymbol{A}^T$$

  - Permutation of the rows or columns of matrix $\boldsymbol{A}$ by a permutation matrix $\boldsymbol{P}$

    $$\boldsymbol{A} \leftarrow \boldsymbol{PA} \quad \text{or} \quad \boldsymbol{A} \leftarrow \boldsymbol{AP}$$

  - Different types and forms of matrix operands as well as different storage schemes are supported

Sparse BLAS

- Sparse BLAS
  - Provides routines for *unstructured sparse* matrices
  - Poorer functionality compared to Dense and Banded BLAS
    - only some basic array operations used in solving large sparse linear equations using iterative techniques
    - matrix multiply, triangular solve, sparse vector update, dot product, gather/scatter
  - Does not specify methods to store a sparse matrix
    - storage format is dependent on the algorithm, the original sparsity pattern, the format in which the data already exists, etc.
    - sparse matrix arguments are a placeholder, or *handle*, which refers to an abstract representation of a matrix, not the actual data components

Sparse BLAS (ctd)

- Several routines provided to create sparse matrices
  - The internal representation is implementation dependent
    - Sparse BLAS applications are independent of the matrix storage scheme, relying on the scheme provided by each implementation
- A typical Sparse BLAS application
  - Creates an internal sparse matrix representation and returns its handle
  - Uses the handle as a parameter in computational Sparse BLAS routines
  - Calls a cleanup routine to free resourses associated with the handle, when the matrix is no longer needed

Example

- **Example**. Consider a C program using Sparse BLAS performing the matrix-vector operation $y \leftarrow Ax$, where

$$A = \begin{pmatrix} 1.1 & 0 & 0 & 0 \\ 0 & 2.2 & 0 & 2.4 \\ 0 & 0 & 3.3 & 0 \\ 4.1 & 0 & 0 & 4.4 \end{pmatrix} \qquad x = \begin{pmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{pmatrix}$$

Example (ctd)

```
#include <blas_sparse.h>
int main() {
  const int n = 4, nonzeros = 6;
  double values[] = {1.1, 2.2, 2.4, 3.3, 4.1, 4.4};
  int index_i[] = {0, 1, 1, 2, 3, 3};
  int index_j[] = {0, 1, 3, 2, 0, 3};
  double x[] = {1.0, 1.0, 1.0, 1.0}, y[] = {0.0, 0.0, 0.0, 0.0};
  blas_sparse_matrix A;
  int k;
  double alpha = 1.0;

  A = BLAS_duscr_begin(n, n); //Create Sparse BLAS handle
  for(k=0; k < nonzeros; k++) //Insert entries one by one
    BLAS_duscr_insert_entry(A, values[k], index_i[k], index_j[k]);
  BLAS_uscr_end(A); // Complete construction of sparse matrix

  //Compute matrix-vector product y = A*x
  BLAS_dusmv(blas_no_trans, alpha, A, x, 1, y, 1);

  BLAS_usds(A); //Release matrix handle
}
```

Parallel Languages

Parallel Languages

- C and Fortran 77 do not reflect some essential features of VP and SP architectures
  - They cannot play the same role for VPs and SPs
- Optimizing compilers
  - Only for a simple and limited class of applications
- Array libraries
  - Cover a limited class of array operations
    - Other array operations can be only expressed as a combination of the locally-optimized library array operations
    - This excludes global optimization of combined array operations

Parallel Languages (ctd)

- Parallel extensions of C and Fortran 77 allows programmers
  - To explicitly express in a portable form any array operation
    - Compiler does not need to recognize code to parallelize
    - Global optimisation of operations on array is possible
  - We consider a parallel superset of Fortran 77
    - Fortran 90

Fortran 90

- Fortran 90 is a Fortran standard released in 1991
  - Widely implemented since then
- Two categories of new features
  - Modernization of Fortran according to the state-of-the-art in serial programming languages
  - Support for explicit expression of operations on arrays

Fortran 90 (ctd)

- Serial extensions include
  - Free-format source code and some other simple improvements
  - Dynamic memory allocation (automatic arrays, allocatable arrays, and pointers and associated heap storage management)
  - User-defined data types (structures)
  - Generic user-defined procedures (functions and subroutines) and operators

Fortran 90 (ctd)

- Serial extensions (ctd)
  - Recursive procedures
  - New control structures to support structured programming
  - A new program unit, MODULE, for encapsulation of data and a related set of procedures
- We focus on parallel extensions

Fortran 90 (ctd)

- Fortran 90 considers arrays first-class objects
  - Whole-array operations, assignments, and functions
  - Operations and assignments are extended in an obvious way, on an element-by-element basis
  - Intrinsic functions are array-valued for array arguments
    - operate element-wise if given an array as their argument
  - Array expressions may include scalar constants and variables, which are replicated (or expanded) to the required number of elements

Fortran 90 (ctd)

- Example:

```
REAL, DIMENSION(3,4,5) :: a, b, c, d
…
c = a + b
d = SQRT(a)
c = a + 2.0
```

WHERE Structure

- Sometimes, some elements of arrays in an array-valued expression should be treated specially
  - Division by zero in a = 1./a should be avoided

- **WHERE** statement

```
WHERE (a /= 0.) a = 1./a
```

- **WHERE** construct

```
WHERE (a /= 0.)
  a = 1./a
ELSEWHERE
  a = HUGE(a)
END WHERE
```

Fortran 90 (ctd)

- All the array elements in an array-valued expression or array assignment must be *conformable*, i.e., they must have the same *shape*
  - the same number of axes
  - the same number of elements along each axis

- Example.

```
REAL :: a(3,4,5), b(0:2,4,5), c(3,4,-1:3)
```

- Arrays a, b, and c have the same *rank* of 3, *extents* of 3,4, and 5, *shape* of {3,4,5}, *size* of 60
  - Only differ in the lower and upper dimension bounds

Array Section

- An *array section* can be used everywhere in array assignments and array-valued expressions where a whole array is allowed

- An array section may be specified with subscripts of the form of *triplet* : `lower:upper:stride`

- It designates an ordered set $i_1,\ldots,i_k$ such that
  - $i_1 = \mathtt{lower}$
  - $i_{j+1} = i_j + \mathtt{stride}$ ($j=1,\ldots,k-1$)
  - $|i_k - \mathtt{upper}| < \mathtt{stride}$

Array Section (ctd)

- Example. **`REAL`** `:: a(50,50)`

- What sections are designated by the following expressions? What are the rank and shape for each section?

  - `a(i,1:50:1)`, `a(i,1:50)`
  - `a(i,:)`
  - `a(i,1:50:3)`
  - `a(i,50:1:-1)`
  - `a(11:40,j)`
  - `a(1:10,1:10)`

Array Section (ctd)

- *Vector subscripts* may also be used to specify array sections
  - Any expression whose value is a rank 1 integer array may be used as a vector subsript

- Example.

```
REAL :: a(5,5), b(5)
INTEGER :: index(5)
index = (/5,4,3,2,1/)
b = a(index,1)
```

Array Section (ctd)

- Whole arrays and array sections of the same shape can be mixed in expressions and assignments

- Note, that unlike a whole array, an array section may not occupy contiguous storage locations

Array Constants

- Fortran 90 introduces *array constants*, or *array constructors*
  - The simplest form is just a list of elements enclosed in (/ and /)
  - May contain lists of scalars, lists of arrays, and implied-DO loops
- Examples.

```
(/ 0, i=1,50 /)
(/ (3.14*i, i=4,100,3) /)
(/ ( (/ 5,4,3,2,1 /), i=1,5 ) /)
```

Array Constants (ctd)

- The array constructors can only produce 1-dimensional arrays
  - Function RESHAPE can be used to construct arrays of higher rank

```
REAL :: a(500,500)
a = RESHAPE( (/ (0., i=1,250000) /), (/
500,500 /) )
```

Assumed-Shape and Automatic Arrays

- Consider the user-defined procedure operating on arrays

```
SUBROUTINE swap(a,b)
REAL, DIMENSION(:,:) :: a, b
REAL, DIMENSION(SIZE(a,1), SIZE(a,2)) :: temp
temp = a
a = b
b = temp
END SUBROUTINE swap
```

Assumed-Shape and Automatic Arrays (ctd)

- Formal array arguments `a` and `b` are of assumed shape
  - Only the type and rank are specified
  - The actual shape is taken from that of the actual array arguments

- The local array `temp` is an example of the automatic array
  - Its size is set at runtime
  - It stops existing as soon as control leaves the procedure

Intrinsic Array Functions

- Intrinsic array functions include
  - Extension of such intrinsic functions as `SQRT`, `SIN`, etc. to array arguments
  - Specific array intrinsic functions
- Specific array intrinsic functions do the following
  - Compute the scalar product of two vectors (`DOT_PRODUCT`) and the matrix product of two matrices (`MATMUL`)

Specific Intrinsic Array Functions

- Perform diverse reduction operations on an array
  - logical multiplication (`ALL`) and addition (`ANY`)
  - counting the number of true elements in the array
  - arithmetical multiplication (`PRODUCT`) and addition (`SUM`) of its elements
  - finding the smallest (`MINVAL`) or the largest (`MAXVAL`) element

Specific Intrinsic Array Functions (ctd)

- Return diverse attributes of an array
  - its shape (`SHAPE`)
  - the lower dimension bounds of the array (`LBOUND`)
  - the upper dimension bounds (`UBOUND`)
  - the number of elements (`SIZE`)
  - the allocation status of the array (`ALLOCATED`)

Specific Intrinsic Array Functions (ctd)

- Construct arrays by means of
  - merging two arrays under mask (`MERGE`)
  - packing an array into a vector (`PACK`)
  - replication of an array by adding a dimension (`SPREAD`)
  - unpacking a vector (a rank 1 array) into an array under mask (`UNPACK`)

Specific Intrinsic Array Functions (ctd)

- Reshape arrays (`RESHAPE`)
- Move array elements performing
  - the circular shift (`CSHIFT`)
  - the end-off shift (`EOSHIFT`)
  - the transpose of a rank 2 array (`TRANSPOSE`)
- Locate the first maximum (`MAXLOC`) or minimum (`MINLOC`) element in an array

Memory Hierarchy

- Parallel programming systems for VPs and SPs take into account their modern memory structure
  - Optimal memory management is often more efficient than optimal usage of IEUs
  - Approaches to optimal memory management appear surprisingly similar to optimisation of parallel facilities
- Simple two-level memory model
  - Small and fast register memory
  - Large and relatively slow main memory

Memory Hierarchy (ctd)

- A simple modern memory hierarchy
  - Register memory
  - Cache memory
  - Main memory
  - Disk memory
- Cache memory
  - A buffer memory between main memory and registers
  - Holds copies of some data from the main memory

Memory Hierarchy (ctd)

- Execution of instruction reading a data item from the main memory into a register
  - Check if a copy of the data item is already in the cache
  - If so, the data item will be actually transferred into the register from the cache
  - If not, the data item will be transferred into the register from the main memory, and a copy of the item will appear in the cache

Cache

- Cache
  - Partitioned into cache lines
    - Cache line is a minimum unit of data transfer between the cache and the main memory
    - Scalars may be transferred only as a part of a cache line
  - Much smaller than the main memory
    - The same cache line may reflect different data blocks from the main memory

Cache (ctd)

- Types of cache memory
  - Direct mapped
    - each block of the main memory has only one place it can appear in the cache
  - Fully associative
    - a block can be placed anywhere in the cache
  - Set associative
    - a block can be placed in a restricted set of places
    - a set is a group of two or more cache lines
    - n-way associative cache

Cache (ctd)

- Cache miss is the situation when a data item being referenced is not in the cache
- Minimization of cache misses is able to significantly accelerate execution of the program
- Programs intensively using basic operations on arrays are obviously suitable for that type of optimization

Loop Tiling

- The main specific optimization minimizing the number of cache misses is loop tiling
- Consider the loop nest

```
for(i=0; i<m; i++)   /* loop 1 */
  for(j=0; j<n; j++) /* loop 2 */
       if(i==0)
   b[j]=a[i][j];
      else
   b[j]+=a[i][j];
```

- `b[j]` are repeatedly used by successive iterations of loop 1

Loop Tiling (ctd)

- If n is large enough, the data items may be flushed from the cache by the moment of their repeated use
- To minimize the flushing of repeatedly used data items, the number of iterations of loop 2 may be decreased
- To keep the total number of iterations of this loop nest unchanged, an additional controlling loop is introduced

Loop Tiling (ctd)

- The transformed loop nest is:

```
for(k=0; k<n; k+=T) //additional controlling loop 0
   for(i=0; i<m; i++) // loop 1
     for(j=k; j<min(k+T,n); j++) // loop 2
       if(i==0)
         b[j]=a[i][j];
       else
         b[j]+=a[i][j];
```

- This transformation is called tiling
- T is the tile size

Loop Tiling (ctd)

In general, the loop tiling is applied to loop nests of the

```
for(i1=...)                /* loop 1 */
      for(i2=...)              /* loop 2 */
        ...
          for(in=...) {    /* loop n */
            ...
            e[i2]…[in]
            ...
          }
```

- The goal is to minimize the number of cache misses for reference `e[i2]…[in]`, which is repeatedly used by successive iterations of loop 1

Loop Tiling and Optimising Compilers

- The recognition of the loop nests, which can be tiled is the most difficult problem to be solved by optimising C and Fortran 77 compilers
  - Based on the analysis of data dependencies in loop nests

- Theorem. The loop tiling is legally applicable (to the above loop nest) iff the loops from loop 2 to loop n are fully interchangeable
  - To prove the interchangability an analysis of data dependence between different iterations of the loop nest is needed

Loop Tiling and Array Libraries

- Level 3 BLAS is specified to support block algorithms of matrix-matrix operations

- Partitioning matrices into blocks and performing the computation on the blocks maximizes the reuse of data held in the upper levels of memory hierarchy

Loop Tiling and Parallel Languages

- Compilers for parallel languages do not need to recognize loops suitable for tiling
- They can translate explicit operations on arrays into loop nests with the best possible temporal locality

Virtual memory

- Instructions address virtual memory rather than the real physical memory

- The virtual memory is partitioned into pages of a fixed size
  - Each page is stored on a disk until it is needed
  - When the page is needed, it copied to main memory, with the virtual addresses mapping into real addresses
  - This copying is known as paging or swapping

Virtual memory (ctd)

- Programs processing large enough arrays do not fit into main memory
  - The swapping takes place each time when required data are not in the main memory
  - The swapping is a very expensive operation
  - Minimization of the number of swappings can significantly accelerate the programs
- The problem is similar to minimization of cache misses and can be, therefore, approached similarly

Vector and Superscalar Processors: Summary

- VPs and SPs provide instruction-level parallelism, which is best exploited by applications with intensive operations on arrays

- Such applications can be written in a serial programming language and complied by dedicated optimizing compilers performing some specific loop optimizations
  - Modular, portable, and reliable programming are supported
  - Efficiency and portable efficiency are also supported but only for a limited class of programs

Vector and Superscalar Processors: Summary (ctd)

- Array libraries allow the programmers to avoid the use of dedicated compilers
  - The programmers express operations on arrays directly using calls to carefully implemented subroutines
  - Modular, portable, and reliable programming are supported
  - Limited efficiency and portable efficiency
    - Excludes global optimization of combined array operations

Vector and Superscalar Processors: Summary (ctd)

- Parallel languages combine advantages of the first and second approaches
  - Operations on arrays can be explicitly expressed
    - No need in sophisticated algorithms to recognize parallelizable loops
  - Global optimisation of combined array operations is possible
  - They support general-purpose programming (unlike existing array libraries)