Goals

- Develop high-performance numerical algorithms for various computer architectures and constrained architectural situations
- Demonstrate methodologies for the design of high-performance algorithms and kernels
- Extract pertinent concepts of algorithm design and performance analysis and performance of numerical and algorithmic components in restructuring compilers and other generalizations for use in restructuring compilers and other generalization environments and improvement environments

Goals
4. Architecture (HW and SW) – Be careful that the technique you are considering maps well to the HW and SW support of the machine.

3. Time – If it takes longer to figure out that you can save work than the time it takes to do the work you save then forget it.

2. Accuracy – Efficient execution may alter stability and roundoff properties compared to slower and more reliable algorithms. (This can be very difficult to check.)

1. Correctness – Efficient execution should not destroy the correctness of the algorithm. (This assumes exact arithmetic and it is not easy to test.)

Costs
that it significantly different in SW or HW architecture. Achieve the same performance percentage on another machine you have to customize for the mix. The resulting code will not machine running the best possible implementation of your code of the peak possible for a machine or more precisely for a algorithm and library designers. To get a substantial partition.

6. Portability – The classic dilemma for high-performance programs. However very useful so to do.

5. Generality – It is very easy to build a very fast program or

Costs (cont.)
and the way in which they influence algorithm design.

• Identify the crucial architectural aspects of new machine
  yields algorithm specialization.

• Identify structure in the problem and determine which
  for new architectures

Basic tasks in designing matrix computations
simple models and empirical data.

Given a set of architectural parameters, investigate performance as a function of algorithm

Advantages and algorithmically useful.

Identify a set of primitives that are architecturally

Basic tasks in designing matrix computations (cont.)
performance

- performance - exploit architectural features to yield high

- single set of primitives

- functionality - many algorithms can be written in terms of

- matrix factorizations

- ranking in complexity from element-element operations to

- matrix computations can be expressed in terms of primitives

___

Primitives
Simplification of software design for nonexpert users •
Portability via libraries

Secondary benefits:
Technology development
Identifies directions for language and restructuring

Algorithms
Provides a correct way of thinking when designing

Primary benefits:
Complexity and under many different hardware-related constraints. Consider associative operations. They can appear at many levels of multiplication. New basic operation is used to replace the simple addition and will see repeatedly as parts of much larger algorithms. Careful study of primitives can also teach basic paradigms that you

**Primitives**
with the use of those resources.

performance level by amortizing effectively overhead associated
effectively saturate the machine's resources at a high
computation in the primitive hierarchy one must use to
saturation level of the machine is the complexity of the
key considerations when designing total primitives (the
saturation level and software paradigm of the architecture are •
all details of architecture are hidden from the user •
user writes sequential code in terms of calls to primitives •
Total primitives:
convention.
example, on a distributed memory with a message-passing
the details of the architecture to be hidden effectively. For
the high-performance software paradigm may not allow all of

construct.
for starting parallel execution is a costly macrotasking
happen, for example, on machines where the only mechanism
significant portion of the application code itself. This can
be too high, perhaps even above the algorithm level to some
granularity required of a total primitive (saturation level) may

total primitives are not always possible.
primitives.

and communication details can be taken care of by the
the parallel tasks and most of the rest of the synchronization
maintain the proper sequence of calls to primitives in each of
execution is seen by the user. After that the user need only
The layout of the data and the initial starting of parallel
macrotasking-based machines or distributed memory machines.

- distributed control primitives: These can be used on
architecture from the user.

Almost total primitives: These primitives hide almost all of the
assemble macro,

range in complexity from a complete subroutine down to an
matrix vector product. Partial primitives can
multiply matrices on an SGI Origin 200 requires a I processor
architecture from the user. For example, a total primitive to
Partial primitives: these primitives hide only a small portion of the
Floating point add/multiply together in one CPU cycle

- Individual processors are also considered high performance
- Element test memory and requires multiple CPU cycles per data
- Fast memory and requires multiple CPU cycles per data that is slower - high latency and lower bandwidth compared to required by the CPU in addition to the large shared memory (registers or cache) that can supply data at the high speed
- Hierarchical memory - processors have small fast memory
- Start and stop high parallel execution is not high moderate number of processors tightly coupled (so cost of otherwise)

Shared memory - all processors can access and refer to all

Architecturally we will be interested in:
We must consider the following characteristics of the algorithms:

- Stability and accuracy – how good are the answers?
- Data locality – how well can the hierarchical memory be used?
- Larger scale for more general parallel processing
- Parallelism – simple and repetitive for vector processing and

...
referenced in the near future

If an address has just been referenced, there is a good probability that a neighboring address will be referenced in the near future.

Spatial Locality: If an address has just been referenced, there is a good probability of it being referenced again in the near future.

Temporal Locality: Two main properties can be exploited:

Locality
(I)A + (f)S = (f)S

\[ M = f \]

\[ N = I \]

Temporal locality on \( A \) and spatial locality on \( S \)

Assume address mapping of arrays is consecutive elements to consecutive addresses.
Temporal locality on $\forall$ and spatial locality on $(f)S$

Note time is defined by the index sweep and therefore, we can

\[
(I)\forall + (f)S = (f)S
\]

Also note that temporal locality does not always imply summation operations.

\[
N, I = 1 \text{ DO}
\]

\[
W, I = f \text{ DO}
\]
The BLAS

- BLAS0: the most recent BLAS updates, matrix-matrix operations
- BLAS1: the original BLAS, vector-vector operations
- BLAS2: the extended BLAS, matrix-vector operations
- BLAS3: the extended BLAS, matrix-matrix operations

Presently, there are three levels of Basic Linear Algebra Subroutines (BLAS) hierarchy.

Historically, linear algebra has adapted to new architectures via the BLAS.
classifications can change considerably. They become more subtle and the traditional preferences and tradeoffs between processors and more complex memory systems. As the number of processors increases with more parallel processors, the hierarchy makes sense for vector and parallel processors with a matrix operation sense. BLAS includes solving triangular systems. Parallel levels include more than just simple vector and parallel primitives. The hierarchy is based on the number of operations in the
processors, e.g., triangular systems.

difficulty with the traditional hierarchy when considering parallel
will discuss a more global overview and the primitives that cause
the implications of the architectures we are considering. Then we
and motivation of each level as it has been historically defined, and
In our consideration of the BLAS we will review the basic concepts
• Poor data locality

• 1-D parallelism

• (u)O operations, (u)O data

• Vector - vector operations

• Lawson et al., 1999 and Linpack '79

BLAS:
Complexity

- Proposal rules out matrix - matrix primitives due to
- Poor data locality
- 2-D parallelism

- For reuse e.g., k-adic operations or rank-1 updates
- Improves register management via accumulation or holding
- $O(n^2)$ operations and data
- Matrix - vector operations
- Standardization Doniga et al.,
- Fang and Jordan, 77, Calahan, 78, on Cray 1
- "BLAS?"
3-D parallelism

- memory due to data locality
- allows efficient management of vector registers, cache, and
  
  \( (u^T \cdot O \cdot u)^2 \) (matrix operations),
  
  \( (u^T \cdot O \cdot u)^3 \) (matrix operations)

- LAPACK

  - early 87 Donagarta et al. - standardization effort
  
  - many papers in latter part of 86 and 87
  
  - (1984) processor, Crustason and IBM on IBM 3090 (begin in
    block Householder, libraries: CSR, on multi-vector
    - 85 Calahan - block LU Cray 2, Bischof and Van Loan

- BLAS3:
\[ z = \frac{\varepsilon u/z}{\varepsilon u/z} = \frac{x_{\text{min}}}{y} \]

parallelism

Ideal speedup or average ideal speedup, \(V = \frac{\log n + z}{\varepsilon u/z} = V/\Theta = \omega_{\text{min}}\)

maximum number of processors

maximum number of data transfers

minimum number of operations

where \(A \in C \in \varepsilon u \times z u\)

Example of each level of the BLAS hierarchy can be seen by considering the operation \(AB + C \rightarrow C\).
one cycle.

Note that for the speedup we have assumed an ideal parallel maximum number of processors.

\[
\frac{2^{\log n + 2}}{2} = \frac{\mu_{\max}}{\mu_{\max}} = (\mu_{\max})^c \cdot \bullet
\]
\[ L^{q} + L^{\varphi} \rightarrow L^{\varphi} \]

row triad

\[ \frac{\partial}{\partial} u + c \rightarrow c \]

column triad

\[ q_{L} v + \Lambda \rightarrow \Lambda \]

dot product

(Vectors, Greek letters are scalars, upper case denotes matrices, any two \( u \) are set equal to one lower case denotes a)

BLAST: any two \( u \) are set equal to one
\( q \mathbf{A} + c \rightarrow c \)

\( \mathbf{A} = \text{matrix-vector product} \)

\( \mathbf{L} = \text{I rank-1 update} \)

\( \mathbf{B} \mathbf{L} + \mathbf{L} \mathbf{c} \rightarrow \mathbf{L} \mathbf{c} \)

\( \mathbf{B} = \text{I vector-matrix product} \)

BLAS2: any one \( u \) set equal to one
These two are simple generalizations of the BLAS rank 1 update. Let \( \gamma \) be a matrix-vector product primitive.

\[
\begin{align*}
\text{for } i &= 1, 2 & \gamma &= u \in \mathbb{R}^n \\
\text{for } i &= 3 & \gamma &= u \in \mathbb{R}^n
\end{align*}
\]

Sizes of the \( \mathbb{R}^n \) two basic examples of interest are differentiated by the relative BLAS3: all \( n \geq 1 \).
CC + A*B

Blas-3 data cube

row triad on N1 axis

matrix x vector in N1, N2 plane

vector x matrix in N2, N3 plane

rank-1 update in N1, N3 plane

C + A*B

origin = (1, 1, 1)