Architecture-Cognizant Divide and Conquer Algorithms *

Kang Su Gatlin and Larry Carter
University of California San Diego
Computer Science and Engineering Department
kgatlin, carter@cs.ucsd.edu

Abstract

Divide and conquer programs can achieve good performance on parallel computers and computers with deep memory hierarchies. We introduce *architecture-cognizant* divide and conquer algorithms, and explore how they can achieve even better performance.

An architecture-cognizant algorithm has functionally-equivalent variants of the divide and/or combine functions, and a *variant policy* that specifies which variant to use at each level of recursion. An optimal variant policy is chosen for each target computer via experimentation. With h levels of recursion, an exhaustive search requires $\Theta(v^h)$ experiments (where v is the number of variants). We present a method based on dynamic programming that reduces this to $\Theta(h^c)$ (where c is typically a small constant) experiments for a class of architecture-cognizant programs.

We verify our technique on two kernels (matrix multiply and 2-D Point Jacobi) using three architectures. Our technique improves performance by up to a factor of two, compared to architecture-oblivious divide and conquer implementations. Further our dynamic programming approach succeeds in selecting the optimal variant policy.

1 Introduction and Motivation

Recursive divide and conquer algorithms have traditionally been used to design algorithms that provide good asymptotic performance on the RAM model and are attractive for parallel programming [4] [9] [11] [13]. Recent work has shown that standard divide and conquer algorithms are also useful with respect to the memory hierarchy. Some of these works use an *architecture-oblivious* approach which uses a single method of dividing the problem at all levels of recursion [3][18]. A few works make use of architecture-cognizant implementations [1] [10]. Our work is a generalization of some techniques used in FFTW [8] which are architecture-cognizant, but specific to FFT.

Divide and conquer improves memory hierarchy performance by recursively reducing the size of the data, thereby reducing the working set size. This reduction may allow the data to fit in a given level of the memory hierarchy (i.e. cache).

Unfortunately other issues may inhibit the performance of these standard divide and conquer codes. For instance, with low set-associativity, cache conflict problems can occur in codes with large arrays of two or more dimensions. This problem occurs when mapping these arrays into storage locations. For example, in a direct-mapped cache of size 2^k, given an array A of size $n \times n$ where $n = 2^c$ and $c \geq k$, two neighboring elements in A (e.g. $a_{i,j}$ and $a_{i+1,j}$ with row-major storage mapping) can not reside in cache at the same time. This conflict exists despite their *logical* proximity. That is just one class of problem that may exist with standard divide and conquer implementations. Other issues that may greatly affect performance include selecting of recursion truncation point, managing memory bandwidth (especially on a shared bus), and coordinating data movement up and down the recursion tree.

1.1 Architecture-Cognizant Divide and Conquer

We will explore the use *architecture-cognizant* implementations for handling these problems. An architecture-cognizant divide and conquer implementation has a *variant policy* which specifies which functionally-equivalent variants of divide and combine mechanisms to use at recursion invocations. This choice is made to maximize performance on a given architecture.

In an architecture-cognizant implementation we have two areas where we can potentially place different variants: (1) In the divide portion of the algorithm we make decisions regarding data movement during recursion and how data should be divided. An example of the former is choosing between copying data or not. An example of the latter is choosing between dividing the space domain or dividing the time domain. (2) In the combine portion of the algorithm we make decisions regarding how data

*Updated versions of the paper are kept at http://www.cs.ucsd.edu/~kgatlin

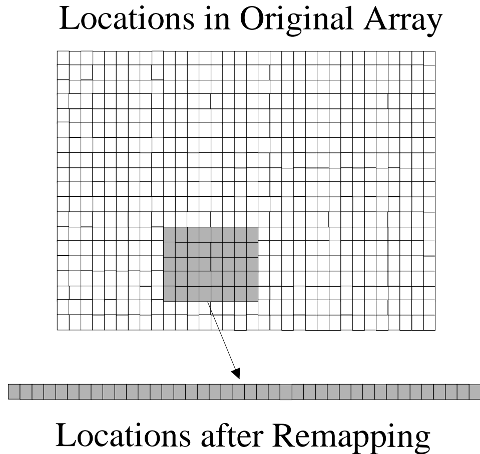

Figure 1: A remapping to move data that may have associativity problems into contiguous space.

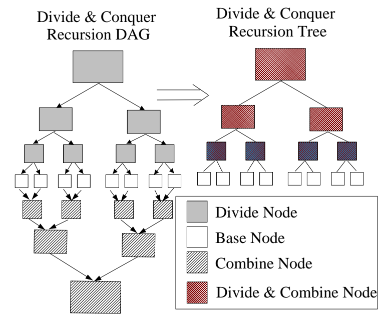

Figure 2: An example divide and conquer recursion DAG on the left and an equivalent divide and conquer tree on the right. The size of the nodes represent the working set size of the data at a given level of the recursion.

should be combined and whether this combining should take place at a given level of recursion or postponed to a higher level. The former can sometimes obtain better locality and the latter can reduce overhead due to combining small subproblems.

Unfortunately optimally determining which variants to call at each level of the recursion is not easy to do without experimentation. It doesn't appear that there is a straightforward analytical solution to the problem. In fact our experiments show that a solution that is logically close to the optimal solution may yield very poor performance. Due to these problems we search for the optimal variant policy experimentally.

An exhaustive search approach to the problem requires v^h experiments (where v is the number of variants and h is the height of the recursion tree). In section 3 we present a dynamic programming algorithm for this problem that can greatly reduce the the number of experiments. For some classes of problems, this algorithm reduces the number of experiments to a polynomial in the height of the recursion tree.

1.2 Outline of Paper

The goals (and outline) of this paper are to:

- Characterize algorithms which may perform better as architecture-cognizant implementations. (Section 2)

- Present a dynamic programming algorithm which reduces the number of experiments necessary to find an optimal architecture-cognizant variant policy. (Section 3)

- Describe implementations of architecture-cognizant matrix multiply and 2D Point Jacobi. (Section 4)

- Show experimental results from these implementations. This will show the performance boost architecture-cognizant implementations can provide over architecture-oblivious implementations. (Section 5)

2 Characteristics of Architecture-Cognizance

In determining if an algorithm is a good candidate for an architecture-cognizant implementation we must understand what characteristics these algorithms should possess. In this section we give a brief outline of characteristics with justification.

- The contiguity in memory of the working set. As mentioned earlier, low associative caches can lead to cache conflicts even if the size of the working set is smaller than the capacity of cache. However if the data for subproblems always resides in contiguous memory then copying is unnecessary. Mergesort and FFT provide a good example. Both have the same complexity of $O(n \lg n)$. Mergesort deals almost exclusively with contiguous data and our preliminary experiments show that an architecture-oblivious approach is as good as an architecture-cognizant one.[1] On the other hand FFT deals largely with large power-of-two strides that can thrash in cache and TLB. Implementations such as the NAS Parallel

[1] In these experiments, unlike the work by [10], we only consider a two-way merge. In future work we will also experiment with multi-way merges and expect our results to show an improvement with architecture cognizance.

FT Benchmark have computer-dependent tuning parameters that specify the size of a subproblem where copying should occur.

- The amount of data reuse. Copying data does not come for free, but analytically determining if copying is worthwhile requires a cost model [15]. In general this can be viewed as the tradeoff between amortizing the cost of moving data against the cost of solving subproblems. If the cost of copying is trivial, then an architecture-oblivious approach that always copies is appropriate. An example is rectilinear Steiner tree minimization [16]. This problem takes $O(3^n)$ time on input data of size n, thus copying data is almost free. By contrast, matrix multiplication requires "only" $O(n^3)$ time for data of size $O(n^2)$. As our experiments will show, the cost of copying data is not negligible and therefore an architecture-cognizant algorithm is appropriate.

- Cost of different variants. 2D Point Jacobi has an interesting tradeoff. Dividing the space domain has an advantage over dividing the time domain in that it gives better locality; however it requires additional computation to be done on the ghost cells.

- Sensitivity to recursion termination point. Continuing to divide the subproblem down to a single data point is rarely a good idea. The best termination point is often determined empirically since standard analytic methods (such as instruction counting) can prove inaccurate with respect to execution time [17]. The dynamic programming algorithm we will present for architecture-cognizant implementations will select the recursion termination point in conjunction with the divide and combine variants for a given machine. As we will see with the Point Jacobi example, the optimal recursion termination point depends on the architecture.

3 Automatically Generating an Optimal Divide and Conquer Tree

For architecture-cognizant divide and conquer algorithms the variant policy specifies which divide or combine variant to execute, or whether to invoke the base procedure, for a given size subproblem. As stated, analytically deriving the optimal variant policy is difficult, therefore we move to an experimental method that can be implemented during preprocessing.

We will limit ourselves to policies that depend only on subproblem size. Formally we define a *variant policy* to be a string of length h over the alphabet V, where V is the set of variants. The ith character in the string represents

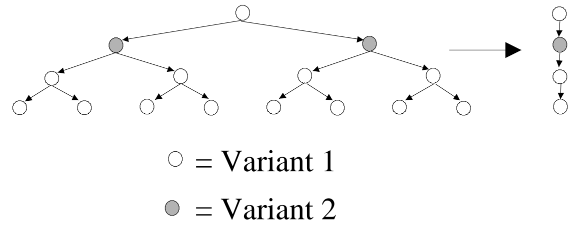

Figure 3: The left shows a tree with two variants. Note that all paths have the same ordering of variants, thus we can view the tree as a single path from the tree.

the variant executed for problems of size S where $2^{i-1} < S \leq 2^i$.

Now we define a special variant called an *isolator variant*. The isolator variant is special in that it has the property known as *isolation*. We formally define this *Isolation Property* here:

Definition 1 *Let V be the set of variants, and $p : V^+ \longmapsto \Re$ be the evaluation function (i.e. $p(a)$ gives the performance of policy a). $i \in V$ has the property of isolation if $\forall L \forall (a, b, c, d$ where $a, b \in V^*$ and $c, d \in V^{L-1})$(if $p(cia) \geq p(dia)$ then $p(cib) \geq p(dib)$).*

In other words, when an isolator variant is used at level L of recursion the optimal policy for levels deeper than L can be chosen independent of the variants chosen at levels less than L. An example of an isolator variant would be a divide phase that copies all of its input parameters to contiguous storage locations with a standard alignment (see Figure 1. Assuming that this copying takes approximately the same amount of time no matter which variant policy is used, the performance at the levels deeper than the copy should be independant of the state before the copy.

As we will see, isolator variants provide the optimal subproblem property necessary to use dynamic programming [5]. A dynamic programming algorithm is desirable since an exhaustive search would need to examine v^h variant policies of length h, where v is the number of variants.

To construct our dynamic programming algorithm we first should note an observation we can make about isolator variants:

Definition 2 *A variant policy is an L-isolated variant policy if it has an isolator variant in the Lth position.*

Observation 1 *If P is an optimal L-isolated variant policy of length L then for all L' > L there exists an optimal L-isolated variant policy, S, of length L' with P as its initial substring.*

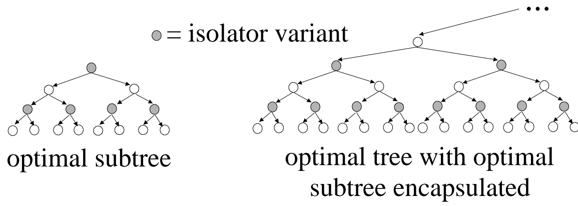

Figure 4: A visual example of Observation 1.

We present pseudocode in Figure 5 of this dynamic programming algorithm. The array $policy[x]$ is the best policy that has an isolator variant in position x. The array $speed[x]$ is the corresponding array that holds the speed of the associated policy. The values $best_speed$ and $best_policy$ hold the speed and policy for the optimal variant policy. The input to the algorithm is the height of the recursion tree and the variants.

The dynamic programming pseudocode uses three functions. $string1||string2$ returns the concatenation of two strings. *head(string, i)* returns a string that consists of the first i characters of $string$. *p(policy)* returns the performance (in work per unit time) of a variant policy that is experimentally determined by running the program on a representative input using the specified policy.

The basic idea behind this dynamic programming algorithm is that if an isolator variant is in position L one must just find the variant policy with respect to positions less than L and we can use that to build up larger variant policies (i.e. if we find we that variant policy cia is better than dia then we know that cib is better than dib).

In the pseudocode of Figure 5 note that we use a default string '*R*' to pad out the variant policy. This was done since we used a fixed problem size for the experiments. From Definition 1 we see that we can set *R* to the empty string and run smaller problem sizes. This is an alternative approach which results in having to run fewer of the largest problem size (and smaller problems on average), but doesn't affect the total number of experiments that need to be run.

The number of experiments required for the dynamic programming algorithm is $\Theta(h^2m^h)$ where m is the number of non-isolator variants. In the cases where we have one non-isolator variant we have reduced the number of experiments required to find the optimal variant policy from $\Theta(2^h)$ to $\Theta(h^2)$.

To verify our dynamic programming algorithm we conducted the experiments presented in Section 5 and verified our search algorithm with the experimental results. In our experiments we show our dynamic programming algorithm always picked the optimal variant policy.

```
proc FindVariantPolicy(h, variants)
  best_speed = SLOWEST
  best_policy = ∅
  policy[0] = ∅
  for (i = 0; i ≤ h; i++)
     R = default string of length h − i (see text)
     policy[i] = policy[i − 1]||B||R
     speed[i] = p(policy[i])
     for (j = 1; j ≤ i; j++)
        foreach strings P of length i − j − 1 w/o an isolator variant
          foreach isolator variant I
            this_policy = policy[j]||P||I||R
            this_speed = p(this_policy)
            if (this_speed > speed[i])
              speed[i] = this_speed
              policy[i] = this_policy
            end if
          end for
        end for
     end for
     if (speed[i] > best_speed)
       best_speed = best_speed[i]
       best_policy = best_policy[i]
     end if
  end for
endproc
```

Figure 5: Pseudocode for dynamic programming algorithm to select optimal variant policy. I is the isolator variant and B is the base procedure (Note: the base procedure can also be an isolator variant). This algorithm requires $\Theta(h^2m^h)$ experiments, where h is the number of levels of recursion and m the number of non-isolator variants, versus $\Theta(v^h)$, where v is the total number of variants, via an exhaustive search method.

```
for i := 1 to r
    for j := 1 to s
        for k := 1 to t
            C[i,j]+ = A[i,k] * B[k,j]
        end for
    end for
end for
```

Figure 6: The naive implementation of a matrix-matrix multiplication algorithm.

4 Description of Algorithms

In this section we describe our architecture-cognizant matrix multiply and 2D Point Jacobi algorithms and their mechanisms for configurability with respect to the architecture.

4.1 Matrix Multiply

Given two matrices A and B of size $r \times t$ and $t \times s$ respectively, Figure 6 gives what we will refer to as the *naive algorithm* for computing $C = C + AB$.

It is clear that for large matrices locality is poor. Tiling of loops for matrix multiply is one locality optimization that has been effective [12]. An alternative is a recursive divide and conquer version of matrix multiply [3], based on the following property of matrix multiply:

Assume A and B are $2^n \times 2^n$ matrices. You can represent each matrix as being divided into four quadrants. The matrix multiply now can be viewed as:

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \times \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

where:

$$\begin{aligned} C_{00} &= A_{00}B_{00} + A_{01}B_{10} \\ C_{10} &= A_{10}B_{00} + A_{11}B_{10} \\ C_{01} &= A_{00}B_{01} + A_{01}B_{11} \\ C_{11} &= A_{10}B_{01} + A_{11}B_{11} \end{aligned}$$

This is a standard recursive definition of matrix multiply and leads to a recursive divide and conquer algorithm. We experimented with this version, but it did not have enough flexibility to achieve the best performance. Instead we implemented a scheme that divides the A and B matrices into two pieces. This is done in one of two ways depending on the aspect ratio of A. If A has more rows than columns we use the following decomposition:

$$\begin{pmatrix} A_0B_0 & A_0B_1 \\ A_1B_0 & A_1B_1 \end{pmatrix} = \begin{pmatrix} A_0 \\ A_1 \end{pmatrix} \times \begin{pmatrix} B_0 & B_1 \end{pmatrix}$$

otherwise we use:

$$\begin{pmatrix} A_0B_0 + A_1B_1 \end{pmatrix} = \begin{pmatrix} A_0 & A_1 \end{pmatrix} \times \begin{pmatrix} B_0 \\ B_1 \end{pmatrix}$$

This also yields a recursive divide and conquer implementation. Implementing this architecture obliviously raises a dilemma: if we never copy data then poor locality can result. If we always copy data then excessive overhead can result.

Our architecture-cognizant algorithm solves this problem by selectively copying data while recursing. This copying variant of the divide procedure copies the data from the arrays that would otherwise require strided access. For instance, if the arrays are stored in row major order and we use the upper matrix decomposition, then the B matrix requires copying to keep the submatrices, B_0 and B_1, stored contiguously. The A matrix may or may not be contiguous in memory at this point in the recursion tree. If it is not contiguous then it is copied; if it is contiguous then it is not copied. Arrays that aren't copied use an implicit representation of the submatrix stored within a larger matrix similar in spirit to the representation used in LAPACK [2]. Neither variant copies the C matrix: however the base procedure uses a dot product formulation that works efficiently even if C is not stored contiguously.

The question this leaves open is: when should the data be copied? Intuitively one would think that only copying at memory hierarchy boundaries would yield the best performance. The experiments we present in Section 5 only partially support this intuition.

4.2 2D Point Jacobi

We consider Poisson's Equation:

$$\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = \overrightarrow{v} \qquad (1)$$

on a set of points (x,y) $\in \Omega$ and with given boundary conditions u = $\overrightarrow{v}$ for (x,y) in a square domain.

The code for solving this equation is a finite difference smoother, specifically 2D Point Jacobi with a five point stencil which is implemented for a fixed number of time steps in a recursive divide and conquer manner. We will solve this with a divide and conquer algorithm using three variants: divide by space (without copying data), divide by space (copying data), and divide by time.

The standard iterative approach to this has an outer loop

```
proc matmul(matrix A, matrix B, matrix C, int level)
  switch (policy(level))
    case (BASE) :
      matmul_BASE(A, B, C, level − 1)
    end
    case (COPY) :
      matmulcopy(A, B, C, level − 1)
    end
    case (NOCOPY) :
      matmulnocopy(A, B, C, level − 1)
    end
  end
endproc
```

Figure 7: The pseudocode for the `matmul` interface. `policy` is a function that decides if at a given level we should call the version of matmul that copies the data or the version that doesn't move the data.

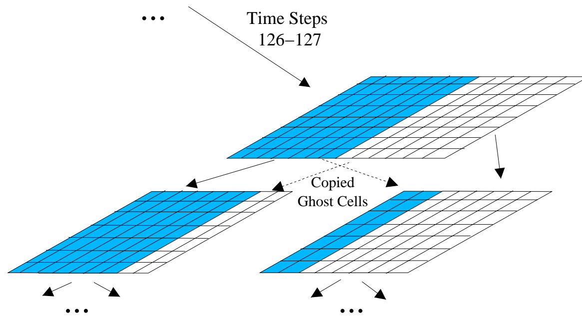

Figure 8: A visualization of the resulting SPACE divide phase when the data is copied. Ghost cells copied are commensurate with the number of time steps to be executed.

that iterates over time and inner loops that apply the five point stencil to each point in the grid. A disadvantage of this approach is that the entire array must be brought into cache once per timestep, resulting in little temporal locality at all levels of the memory hierarchy. An alternative is to partition space into grids that are small enough to fit into a given level of the memory hierarchy, and to execute multiple timesteps on one grid before moving on to the next. This version has the disadvantage that the grid must be surrounded by ghostcells at a depth equal to the number of timesteps. Since the ghostcells require extra computation we need to consider a computation datamovement tradeoff. In order to introduce ghostcells we copy the grid along with associated ghostcells (with a *SPACECOPY* variant) whenever transitioning from dividing the space domain to dividing the time domain.

The *BASE* procedure can be applied to any size grid and any number of timesteps. It uses the standard iterative approach with one modification. It writes the results from the first time step to a temporary grid which is contiguous in memory. The smoother goes back and forth between two temporary grids until the last iteration when the solution is written to the destination array.

The basic structure of the algorithm is given in Figure 9 with a visual aid in Figure 8.

5 Experimental Results

We used three machines for the experiments: (1) 300MHz Digital AlphaStation 500au (Alpha 21064a) running Digital UNIX with a 16K direct mapped L1 cache and a 1MB L2 cache (2) 266 MHz Intel Pentium II Workstation running Linux with a 16K 4-way set associative L1 cache and a 512K L2 cache. (3) A node of a 67MHz IBM SP2 (Power2) running AIX with a 128K 4-way set associative L1 cache and no L2 cache.

For each of the machines we used the vendor supplied compiler except for the Pentium II where we used gcc 2.95. All the codes are written in ANSI C.

5.1 Matrix Multiply Experiments

With the recursive divide and conquer algorithm as specified in Section 4 our goal in these experiments was to determine when should one copy data and also when to terminate recursion (ie define the *policy* function from Figure 7). We looked at a reasonably sized matrix multiply (512×512). The base code for all three architectures was written to expose instruction-level parallelism, but was not tuned for memory hierarchy considerations.

In these experiments we used exhaustive search to try all possible variant policies and compared this with the results from our dynamic programming algorithm. Fig-

```
proc PJ(int start, int stop, grid mesh, int level)
  switch (policy(level))
    case (TIME) :
      PJtime(start, stop, grid, level − 1)
      break
    end
    case (SPACENOCOPY) :
      PJspace(start, stop, grid, level − 1)
      break
    end
    case (SPACECOPY) :
      PJspcecpy(start, stop, newgrid1, level − 1)
      break
    end
    case (BASE) :
      PJbase(start, stop, grid)
    end
  end
endproc
proc PJtime(int start, int stop, grid mesh, int level)
  PJ(start, stop/2, mesh, level)
  PJ(stop/2 + 1, stop, mesh, level)
endproc
proc PJspace(int start, int stop, grid mesh, int level)
  PJ(start, stop, bisect1(mesh), level)
  PJ(start, stop, bisect2(mesh), level)
endproc
proc PJspcecpy(int start, int stop, grid mesh, int level)
  newgrid1 = allocate(grid )
  newgrid2 = allocate(grid )
  COPY(newmesh1, bisect1(mesh))
  COPY(newmesh2, bisect2(mesh))
  PJ(start, stop, newmesh1, level)
  PJ(start, stop, newmesh2, level)
endproc
```

Figure 9: Architecture-cognizant 2D Point Jacobi pseudocode.

```
proc FindVariantPolicy2DPJ(h, variants)
  for (i = 0; i ≤ h; i++)
    for (j = 0; j ≤ i; j++)
      speed[i, j] = SLOW
      policy[i, j] = DEFAULT
    end for
  end for
  best_speed = SLOW
  best_policy = DEFAULT
  Let Ψ_{m,p} be the set of sequences
    (S*T*) of length m with p occurrences of T
  for (i = 1; i ≤ h; i++)
    for (j = 0; j ≤ i; j++)
      Let Y ∈ Ψ_{h−i,0}
      current_best = policy[i − 1, j]||I||Y
      current_speed = p(current_best)
      for (k = 0; k ≤ i; k++)
        for (l = 0; l ≤ min(k, j); l++)
          foreach isolator variant I
            Let X ∈ Ψ_{i−k−1,j−l}
            this_policy = policy[k, l]||X||I||Y
            this_speed = p(this_policy)
            if (this_speed > current_speed)
              current_best = this_policy
              current_speed = this_speed
            end if
          end for
        end for
      end for
      policy[i − 1, j] = head(current_best, i − 1)
      if (current_speed > best_speed)
        best_speed = current_speed
        best_policy = current_best
      end if
    end for
  end for
endproc
```

Figure 10: Pseudocode for dynamic programming algorithm for 2D Point Jacobi algorithm. S is the divide space variant, T is the divide time variant, and I is the spacecopy variant (an isolator variant). The restriction (SPACE variant to TIME variant transitions require a SPACECOPY variant) is represented by the sequences in Ψ. This algorithm runs in $\Theta(h^4)$ time.

ures 11 and 12 show how the choice of variant policies can affect the performance of the algorithm. Figure 11 shows the MFLOP/s when data is copied never, once, or twice during the program (and it also shows the sizes the of the submatrices that are copied into). Figure 12 shows the speed for the *best* among all variant policies that copy a given number of times. Notice that the *simple* variant policies (i.e. always copy or never copy) are never the best.

Not surprisingly, the optimal variant policy for each machine is different and the number of times the data is copied in the optimal variant policy equals the number of levels of cache. For the DEC Alpha and Intel Pentium II, copying at two levels in the recursion gives the best performance. For the Power2 and I, only one copy during the algorithm was needed for best performance.

In terms of absolute performance, this modestly tuned matrix multiply implementation performs surprisingly well compared to vendor tuned routines Our performance is slower yet comparable to ESSL on the Power2 (240 vs 222 MFLOP/s) and the vendor DGEMM on the DEC AlphaStation (140 vs 116 MFLOP/s). On the Pentium 2 our results aren't as impressive, but we suspect this is due to the notorious difficulty in tuning for the Pentium series [7] and not having access to an optimizing Intel Reference Compiler. We reiterate that architecture-cognizant algorithms do perform best with base procedures that are tuned for subproblems that fit in the smallest level of cache.

These experiments show the advantage of architecture cognizance. If data is copied at every level of the recursion the data copying consumes 10-20% of the execution time. If data is copied optimally it uses 2-7% of the execution time.[2] If it is never copied the poor locality results in a penalty that can range from 14% to nearly a factor of three.

We ran the dynamic programming algorithm on all of the machines in an effort to see if they could select the optimal variant policy. For all three architectures the dynamic programming algorithms found the optimal variant policy in far fewer experiments (115 vs 1344 per machine).

5.1.1 Isolation Property

The dynamic programming algorithm is only guaranteed to find the optimal policy if the copy variant is an isolator variant. With the data derived from these matrix multiply experiments we can use tests to see what percentage of the time the isolation property held or was violated, for the copy variant.

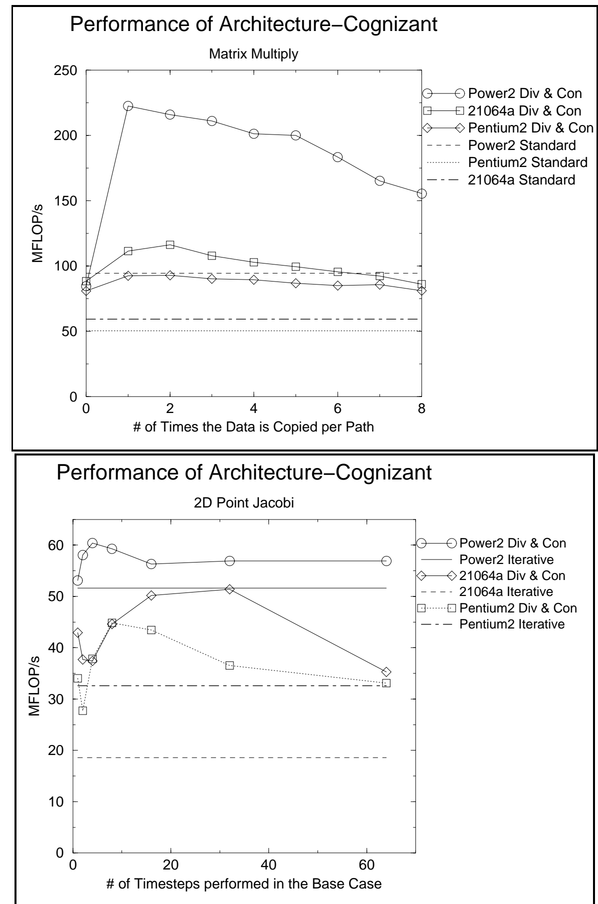

Figure 12: These charts lists the results for the best performing matrix multiply and 2D Point Jacobi policies. The matrix multiply code is dependent on the number of times data is copied along a path. The 2D Point Jacobi is dependent on the number of timesteps executed per call to the base case. For both algorithms on all machines the optimal was selected by the dynamic programming algorithm.

[2]The variance is due to the various costs of copying data on the different architectures.

Pentium II No Copy, One Copy, and Two–Copy Variant Policies for 512x512 Matrix Mu

1st Copy/2nd Copy	256X256	256X128	128X128	128X64	64X64	64X32	32X32	No Copy
512X256	81.1	78.62	76.73	88.92	82.85	**92.79**	84.49	82.74
256X256		77.89	75.41	86.84	80.36	90.04	82.14	79.75
256X128			78.87	86.26	80.86	90.38	83.1	80.95
128X128				87.87	78.88	87.76	80.99	79.48
128X64					86.93	89.71	82.44	91.04
64X64						89.38	79.62	88.41
64X32							86.75	92.48
32X32								88.04
No Copy								80.99

Power2 No Copy, One Copy, and Two–Copy Variant Policies for 512x512 Matrix Multi

1st Copy/2nd Copy	256X256	256X128	128X128	128X64	64X64	No Copy
512X256	202.52	215.88	206.15	203.83	198.72	211.14
256X256		213.02	200.8	203.06	191.68	201.23
256X128			213.12	204.69	190.07	**222.52**
128X128				205.25	184.2	213.03
128X64					198.55	216.56
64X64						206.08
No Copy						84.59

Alpha 21064a No Copy, One Copy, and Two–Copy Variant Policies for 512x512 Matrix

1st Copy/2nd Copy	256X256	256X128	128X128	128X64	64X64	64X32	32X32	No Copy
512X256	83.96	86.61	84.67	110.54	102.81	105.68	91.9	90.56
256X256		82.17	81.56	106.08	98.91	102.03	85.99	88.45
256X128			83.93	108.8	102.81	106.24	93.81	90.41
128X128				**116.21**	100.81	103.08	90.99	87.9
128X64					111.08	104.69	92.46	111.44
64X64						103.86	90.66	106.79
64X32							85.15	108.72
32X32								93.56
No Copy								88.36

Figure 11: These tables list the performance for 512x512 matrix multiply on various machines given 0, 1, and 2 copy-copy policies. We only list these three sets of policies since the best performer was always in one of these three. The numbers of the form NxK indicate the submatrix size of the resulting submatrix after the data has been copied.

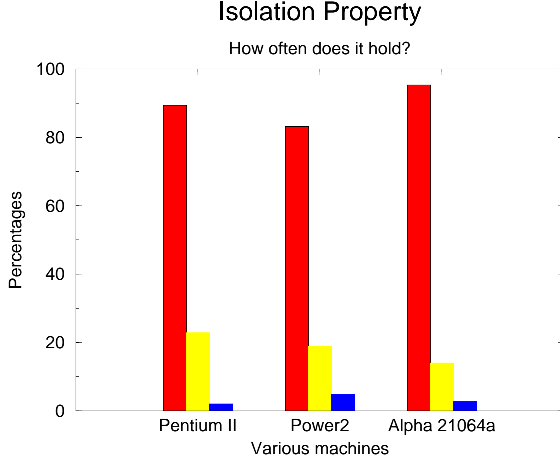

Figure 13: This chart gives information on how well the Isolation Property held for Matrix Multiply experiments done on three computers. The first bar(red) indicates how often the property held. The second bar(yellow) gives the maximum it was off when the property was violated. The last bar(blue) indicates the average it was off when the property was violated.

We determined how often the isolation property held by taking two variant policies of the form, AiK and BiK, where $\mid A \mid = \mid B \mid$ and i is an isolator variant. We then compared AiL to all other variant policies BiL (where $\mid K \mid = \mid L \mid$). If AiK outperformed BiK then AiL should outperform all other variant policies in the form BiL (or vice-versa if BiK outperformed AiK). From these tests we determined the isolation property held from 80% to 100% of the time (depending on the architecture) and when it didn't hold it was usually only modestly off. These facts are indicated in Figure 13.

5.2 2D Point Jacobi

The 2D Point Jacobi code has a more complex divide and conquer structure than the matrix multiply algorithms, but we could still apply the dynamic programming code generation to select the optimal variant policy. For our experiments we smoothed over a 512×512 mesh of doubles for 256 iterations, reporting the MFLOP/s rate.

In these experiments we used an exhaustive search method to try all possible variant policies in an effort to determine the optimal variant policy. The bottom of Figure 12 shows the effect of variant policies on performance. This graph only lists the "best" variant policy for a given number of timesteps executed in the *BASE* procedure. Even among these best performers from each class of variant policies, there is a good deal of variation (up to 61% in performance). We also compare them to an iterative 2D Point Jacobi code and the architecture-cognizant codes in general outperform them.

We now present some notation to explain the variant policies for 2D Point Jacobi:

T = *TIME* divide
C = *SPACECOPY* divide
S = *SPACENOCOPY* divide
B = *BASE* procedure.

For example a policy of the form $B^{16,7}S^2T$ signifies that the TIME variant is called first, then the SPACENOCOPY variant is called twice, and then the BASE procedure is called with 2^{16} units of space to be smoothed over for 2^7 time steps). The restrictions on the order that the variants can be called can be represented as a regular expression of the form: $\beta(S^*T^*)(C(S^*T^*))^*$ where β is a BASE procedure, $B^{x,y}$, with 2^x space units and 2^y time steps. The best performing variant policy of each machine is given here: Pentium II = $B^{14,3}S^4T^5$; Power2 = $B^{16,2}S^2T^6$; 21064a = $B^{15,5}S^3T^3$.

As with matrix multiply we used our dynamic programming algorithm for 2D Point Jacobi, given in Figure 10, and it did find the optimal variant policy for all three architectures with a greatly reduced number of experiments. One should note though that since the Point Jacobi algorithm has two non-isolator variants greater care must be taken to understand how an isolator variant may apply. For the Point Jacobi algorithm the SPACECOPY variant is the isolator variant. By copying the data we assure the data is contiguous (as we did with matrix multiply), but time is invariant of space. Thus we must view each SPACECOPY isolator variant as a function of the number of TIME variants called prior to it in the recursion tree. In other words, we must view isolation not only with respect to space, but also with time.

We give an example here, but first some notation. $C_{i,j}$ is a SPACECOPY isolator in the ith position of the variant policy with j TIME variants prior to it in the variant policy. Now in the following example the two variant policies have isolator variants that have no relationship to each other:

(1) $B^{1,2}T^2S^2C_{5,2}$
(2) $B^{2,1}TS^3C_{5,1}$

Both isolator variants are in position five of the the variant policy, but they have a different number of TIME variants prior to them. Therefore in the two-dimensional space they aren't isolating the same point.

Next we give two variants that are isolating at the same point:

(3) $B^{1,1}T^2SCTSC_{7,3}$

(4) $B^{1,1}T^3S^3C_{7,3}$

If variant policy (3) outperforms variant policy (4) we know that variant policy (3) is the policy that should be used in lieu of (4). Thus we can view the dynamic programming algorithm as one where we must find the best variant policy for all i, j over $C_{i,j}$. The number of variant policies we must examine (experiments we must run) is defined by the following multiple summation:

$$\sum_{i=0}^{h}\sum_{j=0}^{i} c_{ij}$$

where

$$c_{ij} = \sum_{k=0}^{j} k$$

From the above equation or Figure 10 we see that the number of experiments necessary is $\Theta(h^4)$, but with a very small constant ($\frac{1}{24}$).

5.2.1 Parallelism in 2D Point Jacobi

In viewing parallelism as a level of the memory hierarchy we see that our notion of architecture cognizance maps to parallel machines as well. We used a two processor 200 MHz Linux SMP Pentium Pro and a four processor 533 MHz Alpha 21164 Compaq/Digital AlphaServer 4100 SMP [6]. Both architectures have processors that share a bus to memory. Concurrent accesses to memory on the bus are sequentialized, thus a memory-bound kernel looks largely sequential even if run on multiple processors. We choose to do 2D Point Jacobi experiments on these particular machine due to their problems in getting speedup with this type of scientific code.

For our experiments we parallelized our code with Cilk 5.2 [14]. On every divide call to variant *SPACENOCOPY* or *SPACECOPY* we assign the two halves to separate threads. The spawning and scheduling of threads is handled by Cilk's low overhead runtime system.

In our experiments we discovered that serialization of memory accesses has a large effect as most variant policies showed either very little speedup or even slowdown. A few variant policies showed some moderate speed ups and these were realizable through our dynamic programming algorithm.

In Figure 14 we show the speedups attained on the Pentium Pro SMP for classes of variant policies, which are classified based upon how many time steps they execute per `BASE` case. Again we only list the best variant policy from each respective class with the overall best performing algorithm on two processors having a policy of ($B^{15,4}S^3T^4$). From this figure we can see that selecting

P	Optimal Variant Policy
1	$B^{17,0}ST^8$
2	$B^{15,0}S^2T^5CT^3$
3	$B^{15,0}T^4CS^2T^4$
4	$B^{16,0}T^5CS^2T^3$

Table 1: Table of optimal variant policies for the DEC Alphaserver SMP given a number of processors.

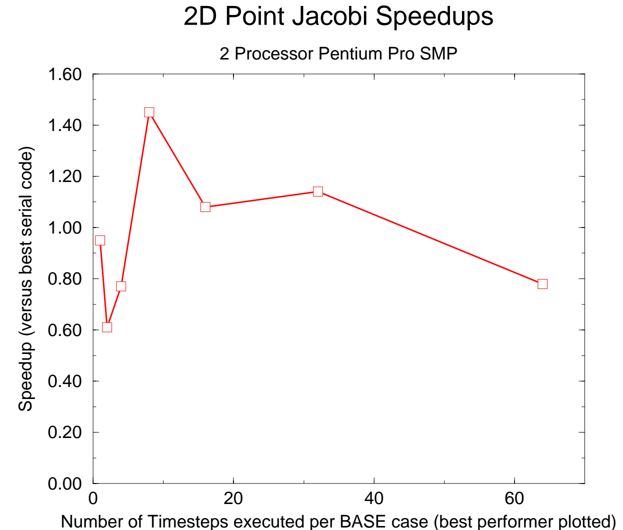

Figure 14: Shows the speedups for the two processor Pentium Pro SMP on the 512X512 2D Point Jacobi code for various numbers of time steps executed per `BASE` case call. The speedups are versus the best serial code and the best of each respective value from the x-axis is reported in the graph.

the wrong variant policy on this system can result in performance that is more than two times worse, despite there being large amounts of parallelization in the code. We again used our dynamic programming algorithm versus the exhaustive search results and the optimal variant policy was selected in far fewer experiments.

In Figure 15 we give speedups on the DEC AlphaServer. Note the the listed speedup is based on the best variant policy for each given number of processors versus the best for a single processor. The optimal variant policies used are given in Table 1. Note that each of the multiprocessor optimal variant policies only do one time step over the grid for L1 cache locality, but they all copy data for locality at a higher level of the memory hierarchy.

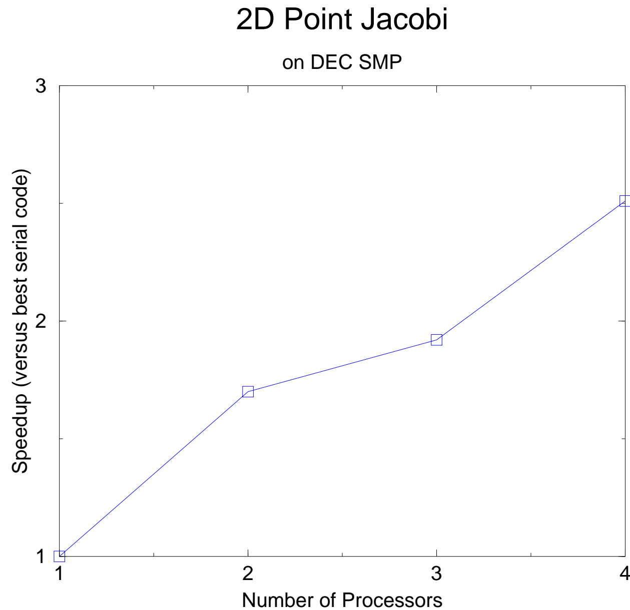

Figure 15: Shows the speedups for the four processor DEC Alphaserver SMP on the 512X512 2D Point Jacobi code. The data is representative of the best variant policy for each number of processors.

6 Conclusions, General Applicability and Future Work

We've shown that there exist a class of divide and conquer problems which architecture-cognizant algorithms perform better than architecture-oblivious implementations. Furthermore we've shown that a dynamic programming technique can be used to find the optimal variant policy in far fewer experiments.

These decision policies in the divide and conquer model we expect to be applicable to a wider range of algorithms. The current work on this is preliminary, but we are optimistic. Future work includes the property of isolation in conjunction with mergesorts that use multi-way merging [10]. We will also examine FFT and MultiGrid codes.

A long term goal of this work is to build a complete preprocessor/compiler/runtime system that is based upon the architecture-cognizant divide and conquer model of programming.

References

[1] B. Alpern, L. Carter, and J. Ferrante, *Space-limited procedures: A methodology for portable high -performance*, in International Working Conference on Massively Parallel Programming Models, 1995.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK: A portable linear algebra library for high-performance computers*, in Technical Report CS-90-105, University of Tennesee, Knoxville, 1990.

[3] R. Blumofe, M. Frigo, and C. Joerg, *An analysis of DAG-consistent distributed shared-memory algorithms*, in 8th Symposium on Parallel Algorithms and Architectures, june 1996.

[4] M. Cole, *Algorithmic skeletons: A structured approach to the management of parallel computation*, PhD thesis, University of Edinburgh, Department of Computer Science, 1988.

[5] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, MIT Press, 1994.

[6] Digital, *AlphaServer 4100 Technical Summary*, http://www.digital.com/info/alphaserver/technical.html, 1998.

[7] M. Frigo, *Portable High-Performance Programs*, PhD thesis, Massachussets Institute of Technology, June 1999.

[8] M. Frigo and S. Johnson, *The Fastest Fourier Transform in the west*, MIT-LCS-TR-728, (1997).

[9] S. Gorlatch and C. Lengauer, *Parallelization of divide-and-conquer in the bird-meertens formalism*, Formal Aspects of Computing, 3 (1995).

[10] A. LaMarca and R. Ladner, *The influence of caches on the performance of sorting*, in Symposium on Discrete Algorithms, January 1997.

[11] G. Mou and P. Hudak, *An algebraic model for divide-and-conquer algorithms and its parallelism*, Journal of Supercomputing, 2 (1988), pp. 257–278.

[12] S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufman, 1997.

[13] R. Rugina and M. Rinard, *Automatic parallelization of divide and conquer algorithms*, in Principles and Practice of Parallel Programming, June 1999.

[14] Supercomputing Technologies Group, *Cilk 5.2 Reference Manual*, http://supertech.lcs.mit.edu/cilk/release/cilk5.2.html, 1998.

[15] O. TEMAN, E. D. GRANSTON, AND W. JALBY, *To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts*, in Proc. of SuperComputing '93, Nov. 1993.

[16] C. THOMBORSON, B. ALPERN, AND L. CARTER, *Rectiliniear Steiner tree minimization on a workstation*, Computational Support for Discrete Mathematics: DIMACS Series in Discrete Mathematics and Theoretical Computer Science, (1994).

[17] M. THOTTETHODI, S. CHATTERJEE, AND A. LEBECK, *Tuning Strassen's matrix multiplication for memory efficiency*, in SuperComputing, November 1998.

[18] S. TOLEDO, *Locality of reference in lu decomposition with partial pivoting*, SIAM Journal of Matrix Anal. Appl., 18 (1997), pp. 1065–1081.