Parallel Processing Letters

THEMIS: COMPONENT DEPENDENCE METADATA IN ADAPTIVE PARALLEL APPLICATIONS

PAUL H J KELLY, OLAV BECKMANN, TONY FIELD
Department of Computing, Imperial College of Science, Technology and Medicine, 180 Queen's Gate, London SW7 2BZ, United Kingdom

and

SCOTT B BADEN
Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA 92093-0114, United States of America

ABSTRACT

This paper describes Themis, a programming model and run-time library being designed to support cross-component performance optimization through explicit manipulation of the computation's iteration space at run-time.

Each component is augmented with "component dependence metadata", which characterizes the constraints on its execution order, data distribution and memory access order. We show how this supports dynamic adaptation of each component to exploit the available resources, the context in which its operands are generated, and results are used, and the evolution of the problem instance.

Using a computational fluid dynamics visualization example as motivation, we show how component dependence metadata provides a framework in which a number of interesting optimizations become possible. Examples include data placement optimization, loop fusion, tiling, memoization, checkpointing and incrementalization.

Keywords: Interprocedural optimization, run-time optimization, libraries for parallel programming

1 Introduction

In many scientific applications, the use of sophisticated data structures and elaborate, adaptive numerical methods can be highly effective in solving computational problems that would otherwise be difficult or impossible to solve. Examples include adaptive multigrid and multipole methods, and coupled multiphysics simulations. Unfortunately, the software complexity associated with these techniques means that they are seldom exploited effectively.

The crucial issue which we propose to address is the apparent conflict between the goals of improving the quality of scientific software and improving its performance. The quest for more usable, higher quality scientific software is reflected in growing interest in component-based scientific programming. Our aim is to reverse

the performance problems associated with composite programs which arise from the use of components which are developed outside the context in which they will be used.

Components are self-describing, separately-deployable units of software reuse. Explicit support for component-based programming is being developed in the scientific computing community [1]. In this paper we avoid the details of such techniques and focus on the metadata needed to support cross-component optimization.

The key new idea behind this paper is to complement data placement metadata [2,3,4,5] with a set of metadata that define *dependencies* between components. Having a powerful component dependence calculus is key to the new directions set out above which we wish to explore.

Contributions. The main contributions of this paper are as follows:

(i) We present a design for Component Dependence Metadata, a general framework for characterising the computational structure, execution order and dependence of software components.

(ii) We show how Component Dependence Metadata can be used to implement a variety of optimizations, including cross-component loop fusion, tiling, data placement optimization and automatic derivation of one- and two-sided communication plans.

(iii) We illustrate the potential for the approach with reference to a computational fluid dynamics visualization application.

(iv) We discuss the relationship between this approach and earlier work.

2 Background

Component-based programming. Recently various research groups have applied component-based software engineering to scientific computation. Examples include [1,6,7]. Component-based programming infrastructures (eg Microsoft's COM and .Net, Javabeans and the Corba Component Model) rely on dynamically-linked libraries, and indirect (virtual) method calls. Both of these techniques present barriers for performance optimization, making run-time techniques essential. An important research question is how to communicate the results of static analysis to the run-time optimizer [8].

Skeletons. The starting point for the skeleton approach is to implement recurring parallel structures of computation and communication, so that implementation and optimization techniques can be reused for a wide range of similar computational patterns [9]. It was quickly recognized that the key issue, after implementing one skeleton efficiently, is to accommodate programs consisting of several skeleton instantiations. Skeleton programming languages such as SCL [10] and P^3L [11] are actually skeleton composition languages. The task of the compiler is to implement composition (sequential, parallel, pipelined or other) efficiently. While much

research has been devoted to transforming skeleton programs (which are generally functional) to improve performance, the most successful work so far [11] has concentrated on resource management: given a pipeline of two parallel components, how should the available processors be divided between them to match their throughput?

The promise of the approach we propose lies in developing these ideas to deal with irregular data. Some prototypes have been built (for example [12]), but little progress has been made on cross-component optimization.

Compilers. From the perspective of conventional compiler techniques, cross-component optimization concerns optimising across sequences of loop nests, which may or may not be encapsulated in subroutines. Data access summary information, as used for interprocedural analysis [13] forms "metadata" describing each component. Unfortunately, with irregular data (even irregular multiblock), the actual dependence between two operations is data-dependent.

The data alignment problem for regular data has been extensively studied [14,15]. One natural approach is to exploit these powerful results in dealing with blocks, while using a run-time technique to handle sets of blocks, and thereby block-irregular applications.

Resource-, Context- and Problem-optimized Component Composition To build adaptive, high-performance scientific applications in the form of re-usable components, we need to optimize the execution of composite programs. The need and opportunity for optimization arises from:

- **Heterogeneous and Varying Resources:** We expect future high-performance computing resources to be heterogenous collections of SMP clusters, linked by fast but heterogenous networks. Furthermore, the exact configuration available is likely to vary, at least from run-to-run.

- **The Context in which Components are Used:** This consists of the data placement and time schedule with which a component's operands are produced, and its results consumed. The component may also be contending for resources with other, concurrently executing components. Optimising components for their context is complicated on systems that support multiple levels of parallelism simultaneously, each with its own characteristic level of communication granularity.

- **The Adaptive and Irregular Nature of Problem Domains:** In irregular and adaptive applications, computation and communication are focused on regions of interest which may change with time.

In the next section we describe the programming model and run-time library that support the development of resource-, context- and problem-optimized composition.

```
void main() {
  Set<Region2> Domain;

  // Build an example multiblock iteration space
  Domain.add(new Region2(0,100, 0,100));
  Domain.add(new Region2(100,200, 50,150));
  Domain.add(new Region2(200,300, 100,200));

  // Declare matching space
  Grid2 W<double>(Domain);
  Set<Region2> Domain_expanded = ... compute storage for U and V, see text
  Grid2 U<double>(Domain_expanded);
  Grid2 V<double>(Domain_expanded);

  // construct Component Composition Graph
  TaskGraph T;
  taskgraph(T) {
    parameter(Grid2<double>, U);
    parameter(Grid2<double>, V);
    parameter(Grid2<double>, W);

    jacobi2d(U, V, Domain);  // S1
    jacobi2d(V, W, Domain);  // S2
  }
  // bind Component Composition Graph to actual in/out parameters
  T.setParameters("U",U, "V",V, "W",W, NULL);

  T.execute();
}
```

Fig. 1: Sketch of a sample multiblock two-dimensional Jacobi application. The `jacobi2d` component (see Figure 2) iterates over three non-intersecting but partially-abutting rectangular regions. The `Grid2` and `Region2` types are based on KeLP's types of the same name. The `TaskGraph` is initialized (using some C++ macros for syntactic sugar) to an abstract syntax tree for the Component Composition Graph. In this example, this simply consists of two instances of the `jacobi2d` component defined in Figure 2.

3 Component Dependence Metadata in Themis

Component dependence metadata consists of two parts — characterising the constituent components, and describing how they are composed:

- Component Composition Graph. This data structure represents the large-grain, inter-component control flow graph.
- Component Dependence Summaries. These dependence metadata provide an abstract description of each component's internal iteration space, as a function of the component's parameters, together with functions mapping each iteration to the memory addresses it may use and define.

Figure 1 shows a simple example in which a Jacobi smoother is applied twice to a multiblock domain (ie a set of rectangular submeshes). The Component Composition Graph specifies the intended execution order of run-time component instances

— in this case of "`jacobi2d`". The actual dependence relationship between them can be calculated in more detail by finding the intersections between data accessed in the first component instance, and data accessed in the second. Thus we capture data dependence, and "storage" dependences, namely anti- and output-dependences arising from explicit re-use of memory[a].

3.1 Representing Component Dependence Summaries

For our current purposes (*pace* the component-based programming community), a component is a procedure which operates on aggregate data. The procedure's operands and results might simply be array subsections. More interestingly, it might operate on a "multiblock" set of array subsections [16]. Furthermore, rather than simply arrays we may have any indexed collection type [17].

To capture this variety, we generalize the notion of a multiblock array decomposition. Given a procedure `P`, we need to discuss `P`'s *properties* and `P`'s *parameters*:

- Property: `P.IterationSpace`

  This is the n-dimensional integer space in which iterations of `P`'s execution are enumerated[b].

  This is an inherent *property* of `P` representing the infinite range of possible executions which might take place.

- Parameter: `P.IterationDomain`

  This describes which actual iterations of `P` should be executed. This is represented as a set of non-intersecting `IterationRegions`. An `IterationRegion` is a polytope in `P.IterationSpace`, characterized as the intersection of a set of integer plane equations each defining a half-space.

- Parameters: `P.Operands` and `P.Results`

  These are the indexed data collections on which `P` operates.

- Property: `P.Uses`

  For each of the parameter `Operands`, this maps each point in the `IterationSpace` to the set of indices of the indexed collection which might be accessed (read) by that iteration.

  For simple array and multiblock computations, this can usually be represented as an affine function. In [5] we show how this can be extended to capture data which is accessed by many iterations (leading to a broadcast in a parallel implementation).

[a] In [4] we describe a run-time renaming scheme which can remove execution order constraints due to storage reuse — but explicit control remains important in many applications to avoid running out of space.

[b] In the case where P consists of an imperfect nest of loops, this is a simplification: a statement at an intermediate loop nesting level is represented by a set of points in the iteration space. This appears not to interfere with the effectiveness of the model.

- Property: `P.Defines`

  This is just the same as `P.Uses` but characterizes the data items (ie the elements of the `P.Results` collections) which might be *written* to by each given iteration.

Motivation. It is important to understand that it is not enough simply to characterize the set of data items which might be read/written by a component. This would be enough to find out whether invocation of two components `P` followed by `Q` are dependent. However, we need to understand the dependence relationship between corresponding iterations.

For example, to determine whether the outermost loop of `P` can be fused with the outermost (i) loop of `Q`, we need to determine whether every value needed by iteration i of `Q` is available by iteration i of `P`. We return to this important issue in Section 3.2.

```
class Region2 {
  public int i_lower, i_upper, j_lower, j_upper;

  // Constructor
  Region2(int i_l,int i_u, int j_l,int j_u) {
    i_lower = i_l; i_upper = i_i;
    j_lower = j_l; j_upper = j_u;
  }
}
class Grid2<T> {
  Set<Region2> DataArrayShapes;
  Set<Array2<T>> DataArrays;

  // Constructor
  Grid2(Set<Region2> RegionShapes) {
    foreach (i=0; i<=RegionShapes.size; ++i) {
      DataArrayShapes.add(RegionShapes[i]);
      DataArrays.add(new Array2(RegionShapes[i]));
    }
  }
}
void jacobi2d(Grid2<double> U, Grid2<double> V, Set<Region2> Domain) {

  // for each region in the set of regions
  foreach(Region2 R, Domain)
    {
      // do the standard Jacobi loop
      for (int i=R.i_lower; i < R.i_upper; ++i)
        for (int j=R.j_lower; j < R.j_upper; ++i)
          V[i][j] = (U[i-1][j]+U[i+1][j]+U[i][j-1]+U[i][j+1])*0.25;
    }
}
```

Fig. 2: Sketch of example multiblock two-dimensional Jacobi component. The component dependence metadata for `jacobi2d` is given in the text. The Jacobi loop iterates over a set of regions. The `Array2`, `Grid2` and `Region2` types are based on KeLP's types of the same name.

3.2 Example: Multiblock Jacobi

Figure 2 shows a much-simplified example to illustrate the component dependence metadata and its application. Each run-time instance of the `jacobi2d` component can be queried for the following metadata:

- Property `jacobi2d.IterationSpace` is simply the two-dimensional vector space of positive integers $[0:\infty] \times [0:\infty]$.

- Parameter `jacobi2d.IterationDomain` is a `Set` of three rectangular sections of `jacobi2d.IterationSpace`.

- For the first `jacobi2d` instance in Figure 1, parameters `jacobi2d.Operands` and `jacobi2d.Results` are `U` and `V` respectively.

  `V` is a `Set` of rectangular arrays whose bounds match the corresponding elements of `jacobi2d.IterationDomain`.

  This exact correspondence between the shape of the IterationDomain and the shape of the Result data structure occurs frequently — iteration (i, j) of the Jacobi loop assigns to location `V[i][j]`.

  The situation for `U` is somewhat more complicated, since the Jacobi loop reads a "halo" of locations (often called ghost cells) outside the range of iterations (i, j), owing to the `i-1`, `i+1` and `j-1`, `j+1` index expressions.

  To prevent these accesses from being bounds errors (and to provide boundary conditions), the storage for `U` must be somewhat larger — we need to grow each of the constituent regions by one in each direction. Although we could do this in an ad-hoc fashion, it can be handled systematically using the `Use` mappings below.

- Property `jacobi2d.Defines` consists of a single mapping, being the identity function from iteration (i, j) in `jacobi2d.IterationSpace` to location `V[i][j]` in `V`. There is one mapping because the Jacobi loop has just one assignment to `V`.

- Property `jacobi2d.Uses` consists of four mappings:

  - $f_1(i, j) = (i - 1, j)$ in `U`, owing to the memory reference `U[i-1][j]`
  - $f_2(i, j) = (i + 1, j)$ in `U`, owing to the memory reference `U[i+1][j]`
  - $f_3(i, j) = (i, j - 1)$ in `U`, owing to the memory reference `U[i][j-1]`
  - $f_4(i, j) = (i, j + 1)$ in `U`, owing to the memory reference `U[i][j+1]`

In our prior work [2,3,4,5,15], component metadata describes data placement constraints. In this framework, component dependence metadata captures the available flexibility in *execution order*.

Example. As an example of using the dependence information, consider the pair of Jacobi instances in Figure 1:

```
jacobi2d(U, V, Domain);  // S1
jacobi2d(V, W, Domain);  // S2
```

Here, we apply the Jacobi operation in statement `S1` to an initial set of Grids `U`, yielding `V`, then a second step `S2` to produce `W`. This execution order makes somewhat inefficient use of cache memory; it would be beneficial to fuse the two loops. However a simple calculation using the `Uses` mappings shows that the resulting single loop nest would fail to respect the dependences required — element `V[i][j+1]` is used by iteration (i, j) of `S2` but is generated in iteration $(i, j+1)$ of `S1`. We show how the validity of loop fusion is tested in Section 4.3.

However, it turns out that these loops can be fused. The trick [18] is to renumber `S2.IterationSpace` by shifting it by 1 in both i and j. This aligns iteration $(i+1, j+1)$ of `S1` with iteration (i, j) of `S2`. Now no dependence violation occurs.

4 Using Component Dependence Metadata

This section illustrates how component dependence metadata can be used to solve some simple cross-component optimization problems. This should explain some of the motivation behind the approach.

4.1 Deriving Data Placement Constraints

Given a data distribution D which specifies a set of subsections of an array `A` which is accessed by component `P`, we can calculate the required placement of `P`'s other operands/results as follows:

(i) Find the iteration domain corresponding to the data decomposition D. If `A` is an operand, find the set of `Uses` mappings which map iterations to uses of `A` (if `A` is a result, find the corresponding `Defines` mappings).

(ii) Invert these mappings to find the iterations which use each of the subsections described in D (assuming, of course, that the mappings are invertible).

(iii) Now, find all the data accessed by these iterations using the `Uses` and `Defines` mappings forwards.

This allows us to derive Beckmann's data placement metadata when the mappings are invertible. With replication, the mappings are not invertible [5]. Some further work is needed to show how to calculate placement constraints in this case.

Comment: Enumerated versus closed-form domains. To implement the multiblock domain decomposition of Figure 2, we simply enumerate the set of subdomains. To represent a regular domain decomposition, such as block-wise, cyclic or block-cyclic, this would be unwieldy. Instead we plan to use an extension of the `Set` collection type which uses a closed-form generator function to produce its

elements on demand. Where appropriate, this generator function can be accessed explicitly.

For example, consider the problem of finding the data placement constraints in a regular array context as discussed above. If the data decomposition D above is given as a closed form, say a blockwise decomposition, the inverse `Use` mappings can be used to yield the `IterationDomain` also in closed form.

4.2 Composing Parallel Components — Deriving A Data Communication Plan

To execute the Jacobi example in parallel, we need to partition the `IterationDomain` across the p processors. Call this p-element set of `IterationDomains` the `IterationDomainDecomposition`. Given some arbitrary partitioning, we need an efficient way to calculate the data communications involved in a specified computation (in KeLP this is called the "`MotionPlan`"). Consider our Jacobi example again; assume that the same partitioning is used to execute both `S1` and `S2`:

```
// this loop executes once on each processor
foreach (proc, ProcessorSet)
S1:  jacobi2d(U, V, IterationDomainDecomposition[proc]);

// implicit data redistribution required

// this loop executes once on each processor
foreach (proc, ProcessorSet)
S2:  jacobi2d(V, W, IterationDomainDecomposition[proc]);
```

Now each processor i looks up `IterationDomainDecomposition` to find the iterations it must execute. However, when processor i executes `S2`, it needs some values from other processors (due to the ghost cell halo). We can calculate which values are needed, and where they are stored:

(i) Use the `Uses` mappings of `S2` to find the set $uses_i$ of memory locations accessed by processor i's iterations.

(ii) Use the `Defines` mappings of `S1` to find the set $defs_j$ of memory locations written to by each processor j.

(iii) On each processor i, compute the intersection of its $uses_i$ with the $defs_j$ of each of the other participating processors. This is the set of receive operations required.

(iv) On each processor j, compute the intersection of its $defs_j$ with the $uses_i$ of each of the other participating processors. This is the set of send operations required.

An implicit assumption here is that data needed by `S2` but not produced by `S1` is already available. This happens naturally, as it must have been generated by some earlier component, say `S0` — we simply make sure this automatic data distributed operation is applied when `S0` is composed with `S1; S2`.

Data-dependent Uses mappings. Note that we assumed that each processor can calculate the `Uses` mappings of all the other processors. If it cannot, the communications must be one-sided, initiated by the processor which needs the data. In some interesting examples (such as locally-essential trees in implementations of the Barnes-Hut algorithm [19]), we can conservatively approximate the set of data needed by a processor.

4.3 Cross-Component Loop Fusion

As mentioned in Section 3.2, a key motivation is to support cross-component loop fusion and related ideas, including tiling. To check the validity of loop fusion, we need to know more than just the set of data items are accessed by the two loops — we also need to know about the order in which the elements are produced and used.

Assume that the `IterationSpace`s of the two components `S1` and `S2` are the same. To test whether a component `S1` can be fused with a component `S2`, we need to construct the dependence equation for each potential dependence (we discuss data dependences here; anti- and output-dependences are similar):

(i) Where a collection `A` appears in both `S1.Defines` and `S2.Uses`, we introduce the corresponding mappings $\phi(\vec{i})$ to model the access patterns due to each memory reference:

$$\texttt{S1.Defines[A]}.\phi(\vec{i}) \quad \text{and} \quad \texttt{S2.Uses[A]}.\phi(\vec{i})$$

(where $\vec{i}$ is a d-element vector representing a point in the d-dimensional `IterationSpace`).

(ii) Now, consider two distinct iteration space points, $\vec{i}$ and $\vec{i'}$. A dependence between iteration $\vec{i}$ of `S1` and iteration $\vec{i'}$ of `S2` occurs when the *dependence equation* is satisfied:

$$\texttt{S1.Defines[A]}.\phi(\vec{i}) = \texttt{S2.Uses[A]}.\phi(\vec{i'}).$$

(iii) To classify the dependence, we need to characterize the solutions to this dependence equation. There might be no dependence:

- There may be no integer solution at all
- The solutions may all lie outside the actual loop bounds (the `IterationDomain`)
- In an `IterationSpace` with non-unit step, the solutions may occur only at non-executed iterations

If there is a dependence, we need to find out whether there exists a solution for which $\vec{i} > \vec{i'}$ (under the lexicographic ordering).

As explained earlier, the presence of such a dependence reflects that when fused, `S2` would attempt to read a value before it has been generated by `S1`.

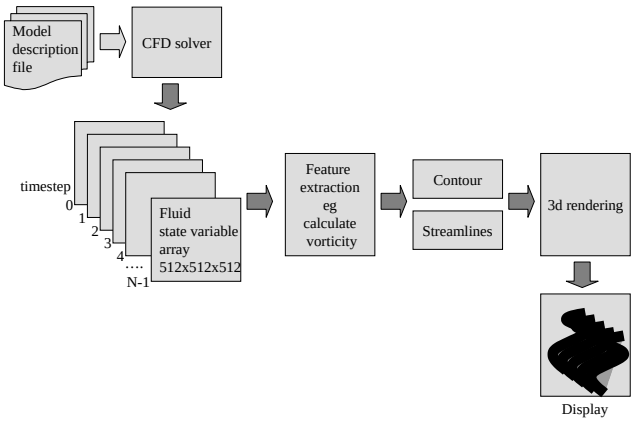

Fig. 3: Structure of the CFD visualization application.

To solve the dependence equation, our prototype implementation uses Fourier-Moztkin elimination, a standard technique [20]. Although this can, in principle, be computationally hard, the equations found in practice are almost always very simple and the time taken has been minimal.

5 Extended Example: Visualization in Computational Fluid Dynamics

To provide a testbed for these ideas, we have been developing a simple visualization tool for a three-dimensional computational fluid dynamics application.

Figure 3 shows the overall structure of the application. The prototype is a straightforward implementation using standard tools; the user interface is implemented in Tcl/tk, the visualization uses vtk [21], and the CFD application is NaSt3DGP [22]. The application essentially a simplified version of SCIRun [23]; the objective is to motivate and demonstrate generic mechanisms to support applications of this kind.

The challenge we focus on is to handle very large finite-difference meshes at each timestep, while supporting interactive exploration of the flow evolution over time. Our prototype allows the user to rotate, pan, zoom in and out to view the fluid volume, slice/select fluid subregions of interest, add specified isosurfaces (contours) and streamlines to show flow patterns and eddies, and use a slider to produce a smooth animation of the scene over a range of timesteps.

To achieve interactive responsiveness, we plan to use THEMIS to explore a number of performance enhancement techniques. For example:

- **Checkpointing/memoization** For interesting examples, the mesh representing the flow state at each timestep may be several gigabytes in size (eg $512 \times 512 \times 512$ 8-byte doubles per state variable).

  Conventionally, at each timestep the entire fluid state mesh is written to disk. Especially in a parallel system, file access can dominate execution time both for flow calculation and subsequent visualization.

Instead, we propose to let the THEMIS run-time system decide which results to store, and which to recompute on-demand. Thanks to the dependence information, THEMIS has a complete recipe for each intermediate value calculated.

This approach can be compared with periodic checkpointing of the fluid simulation. The dependence metadata gives THEMIS precise details of what data needs to be stored.

- **Scheduling and placement of malleable task graphs** When the user requests a timestep whose mesh has not been stored, we need to go back to the most recently stored fluid state, and re-run the computation from there.

  To do this quickly, we need instantaneous access to multiple processors. Unless a large parallel computer can be dedicated to the user, we need to make use of whatever resources are free at the time (see for example recent work at Imperial [24]).

  We propose to use THEMIS to decompose and schedule the computation using the (possibly-heterogenous) processors and network capacity available.

  A more sophisticated extension of this idea is to take into account the data already available on the machines in question. If a processor is used for the first time, the scheduler must account for the time to ship the code and data it needs. Subsequent uses can skip this step and perhaps also use cached intermediate results too.

- **Incrementalization** If the user is viewing only a slice of the volume, we can propagate the demand for data back through the Component Composition Graph, so that contouring is applied only to the visible region — indeed only the visible region need be extracted from the fluid simulation.

  When the user shifts the slice of the data to be rendered, we need to redo this demand propagation. The interesting challenge is to make use of whatever parts of the intermediate values we already have.

- **Fusion, tiling and pipelining** The straightforward implementation of Figure 3 would load a mesh, then apply a contouring algorithm, then apply a streamlining algorithm, then render the resulting polygons. These repeated traversals of the mesh make poor use of cache (and virtual memory). Using the loop fusion techniques described earlier, THEMIS should be able to combine multiple passes.

  This mixture of task- and data-parallelism creates a rich variety of alternative parallel implementations, including the classical rendering pipeline. `Themis` can use dependence information to implement these alternatives; we need to develop optimization algorithms (for example, see [25] to find the best one for the circumstances.

By operating at run-time, these techniques (some of which are, of course, conventional compiler technology) can be deployed adaptively, to react to actual component run-times, resource availability etc.

6 Related Work

We discussed the key published background work in Section 2. Here we briefly focus on a specific point of reference — KeLP. Component dependence metadata and the dependence calculus have been heavily influenced by Baden's use of metadata for structured irregular grids [16], which is currently being extended to unstructured meshes. KeLP's data placement metadata, the FloorPlan, defines the mapping of a block-structured irregular array onto an array of processors. KeLP further provides a *region calculus* which, given two different FloorPlans for some block-irregular array, can derive an optimized data motion plan to perform the communication for redistributing the data from one placement to the other.

Our Component Composition Graph is analogous to KeLP's MotionPlan, but rather than representing data movement, the Component Composition Graph represents a large-grain, inter-component dataflow graph.

Regarded as an extension to KeLP, Component Dependence Metadata will allow us to increase the scope for adaptive run-time scheduling, as well as off-line optimization. Further, the metadata will provide the infrastructure for automatic placement of intermediate data, currently not supported by KeLP.

Another interesting point of reference is DUDE [26]. In this C++ library, the programmer adds an explicit description of the dependence distance vectors connecting each pair of dependent components. The DUDE run-time system can then calculate what synchronization and communication is needed. Thus, in DUDE, dependence information has to be added for each component *composition*. By contrast, in THEMIS, the dependence metadata is associated with each component. The dependences between components is automatically calculated from this information.

In some sense, THEMIS can be regarded as an extension of Jade [27]. Jade is a parallel object-oriented language based on C++. Each method has an associated access descriptor which describes the objects it may read or write. Jade's run-time system automatically arranges the synchronization and communication required. In Jade, an access to an object in shared memory is potentially an access to any part of the object. In THEMIS, the dependence metadata provides more refined information about which consituents of a chared collection type might be accessed.

7 Implementation Status

The THEMIS library has not yet been implemented, but many of the ideas have been investigated in prototype form. Our "TaskGraph" library (implemented by Alistair Houghton [28]) provides a convenient syntax for the Component Composition Graph using templates, overloading and macros in C++. The library automatically derives Component Dependence Summaries for simple loop procedures, and summary metadata can be added manually for user-supplied functions.

Once the TaskGraph has been optimized, it is printed as a C program, compiled, then linked back into the running application. Considerable performance advantage is gained from run-time code generation, owing to specialization and also by avoiding function and virtual function call overheads. The library automatically exploits dependence information by fusing loops wherever possible.

THEMIS will extend this with a dependence calculus, for manipulating component dependence metadata, together with a library for manipulating the iteration domains of the components to generate optimized code. This will provide the tools with which a programmer can implement the interactive visualization application as we have described.

8 Conclusions

We have presented THEMIS, a software framework for cross-component performance optimization. The key idea is for each component to carry Component Dependence Metadata which gives an abstract and general characterization of how its iteration space accesses shared data. We present a design for Component Dependence Metadata which links the accessed data regions to the iteration space, and we demonstrate how this makes loop fusion possible.

The main challenge for future work is to provide flexible, powerful, explicit control of cross-component optimization as we have described, without introducing unmanageable complexity.

Acknowledgements

This work builds on extensive discussions with Susanna Pelagatti. We gratefully acknowledge the EPSRC's funding for Baden and Pelagatti to visit Imperial College (Ref GR/N/35571). Olav Beckmann's work was also supported by EPSRC through a PhD studentship. We are grateful to the various students who have contributed towards prototype implementation, notably Alistair Houghton, and Kulwant Bhatia and his team.

References

1. Steven Newhouse, Anthony Mayer, and John Darlington. A software architecture for hpc grid applications. In *Euro-Par 2000*, volume 1900 of *Lecture Notes in Computer Science*, 2000.
2. Olav Beckmann and Paul H. J. Kelly. Runtime interprocedural data placement optimisation for lazy parallel libraries (extended abstract). In *Proceedings of Euro-Par '97*, number 1300 in LNCS, pages 306–309. Springer Verlag, August 1997.
3. Olav Beckmann and Paul H. J. Kelly. Data distribution at run-time: Re-using execution plans. In *Proceedings of Euro-Par'98*, number 1470 in LNCS, pages 413–421, Southampton, UK, September 1998. Springer-Verlag.
4. Olav Beckmann and Paul H. J. Kelly. Efficient interprocedural data placement optimisation in a parallel library. In *LCR98: Languages, Compilers and Run-time Systems for Scalable Computers*, number 1511 in LNCS, pages 123–138. Springer-Verlag, May 1998.
5. Olav Beckmann and Paul H. J. Kelly. A linear algebra formulation for optimising replication in data parallel programs. In *LCPC99: Languages and Compilers for Parallel Computing*, number 1863 in LNCS, pages 100–116. Springer-Verlag, August 1999.
6. Omer Rana, Maozhen Li, Shields, David Walker, and David Golby. Implementing problem solving environments for computational science. In *Euro-Par 2000*, volume

1900 of *Lecture Notes in Computer Science*, 2000.

7. R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high performance scientific computing, 1999.
8. Shamik Sharma, Anurag Acharya, and Joel Saltz. Deferred Data-Flow Analysis. Technical Report TRCS98-38, University of California, Santa Barbara, December 30, 1998.
9. Murray I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman / MIT Press, 1989.
10. John Darlington, Yike Guo, Hing Wing To, and Jin Yang. Parallel skeletons for structured composition. In *PPoPP'95: Principles and Practice of Parallel Programming*, pages 19–28. ACM Press, August 1995. Published as ACM SIGPLAN Notices 30(8).
11. Susanna Pelagatti. *Structured Development of Parallel Programs*. Taylor & Francis, London, U.K., 1997.
12. Qian Wu, Anthony J. Field, and Paul H. J. Kelly. M-Tree: A parallel abstract data type for block-irregular adaptive applications. In *Proceedings of Euro-Par '97*, number 1300 in LNCS, pages 638–649. Springer Verlag, August 1997.
13. B. Creusillet and F. Irigoin. Interprocedural analyses of Fortran programs. *Parallel Computing*, 24(3–4):629–648, May 1998.
14. Siddhartha Chatterjee, John R. Gilbert, Robert Schreiber, and Shang-Hua Teng. Automatic array alignment in data-parallel programs. In *POPL'92: ACM Symposium on Principles of Programming Languages*, pages 16–28. ACM Press, 1993.
15. Olav Beckmann and Paul H. J. Kelly. A review of data placement optimisation for data-parallel component composition. In Sergei Gorlatch and Christian Lengauer, editors, *CMPP 2000:* 2^{nd} *International Workshop on Constructive Methods for Parallel Programming*, pages 3–18, July 2000. Published as Technical Report MIP–007 of the University of Passau.
16. Stephen J. Fink, Scott B. Baden, and Scott R. Kohn. Efficient run-time support for irregular block-structured applications. *Journal of Parallel and Distributed Computing*, 50(1):61–82, April 10/May 1 1998.
17. S.B. Baden, P. Colella, D. Shalit, and B. Van Straalen. Abstract kelp. In *10th SIAM Conference on Parallel Processing for Scientific Computing, Portsmouth, Virginia*, March 2001.
18. Alain Darte. On the complexity of loop fusion. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*, pages 149–157, Newport Beach, California, October 12–16, 1999. IEEE Computer Society Press.
19. J. Salmon. *Parallel Hierarchical N-body Methods*. PhD thesis, California Institute of Technology, 1990.
20. Michael Wolfe. *High-Performance Compilers for Parallel Computing*. Addison Wesley, 1995.
21. William J. Schroeder, Kenneth M. Martin, and William E. Lorensen. *The Visualization Toolkit*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, second edition, 1998.
22. Michael Griebel, Frank Koster, Michael Meyer, and Roberto Croce. NaSt3DGP – a parallel flow solver. http://wwwwissrech.iam.uni-

bonn.de/research/projects/koster/NaSt3DGP/index.htm.

23. Steven G. Parker and Christopher R. Johnson. SCIRun: A scientific programming environment for computational steering. In Sidney Karin, editor, *Proceedings of the 1995 ACM/IEEE Supercomputing Conference, December 3–8, 1995, San Diego Convention Center, San Diego, CA, USA*, pages ??–??, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995. ACM Press and IEEE Computer Society Press.
24. Mark Rossiter. Exploiting idle workstations to accelerate computation. MEng dissertation, Department of Computing, Imperial College (supervised by Paul H J Kelly), 2000.
25. David B. Skillicorn and Susanna Pelagatti. Building programs in the Network Of Tasks model. In *SAC 2000: ACM Symposium on Applied Computing*, pages 248–254. ACM Press, March 2000.
26. Suvas Vajracharya and Dirk Grunwald. Loop re-ordering and pre-fetching at runtime. In *Supercomputing '97*. ACM Press and IEEE Computer Society Press, 1997.
27. Martin C. Rinard and Monica S. Lam. The design, implementation, and evaluation of jade. *ACM Transactions on Programming Languages and Systems*, 20(3), May 1998.
28. Alastair Houghton. Run-time specialisation using a C++ meta-language. MEng dissertation, Department of Computing, Imperial College (supervised by Paul H J Kelly), 2000.