Language-Based Optimisation of Sensor-Driven Distributed Computing Applications

Jonathan J. Davies, Alastair R. Beresford, and Alan Mycroft

Computer Laboratory, University of Cambridge,
15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK
`Firstname.Lastname@cl.cam.ac.uk`

Abstract. In many distributed computing paradigms, especially sensor networks and ubiquitous computing but also grid computing and web services, programmers commonly tie their application to a particular set of processors. This can lead to poor utilisation of resources causing increased compute time, wasted network bandwidth or poor battery life, particularly if later changes to the architecture or application render early decisions inappropriate. This paper describes a system which separates application code from the description of the resources available to execute it. Our framework and prototype compiler determines the best location to execute different parts of the distributed application. In addition, our language encourages the programmer to structure data, and the operations performed on it, as monoids and monoid homomorphisms. This approach enables the compiler to apply particular program transformations in a semantically-safe way, and therefore further increase the flexibility of the assignment of application tasks to available resources.

1 Introduction

Sensor networks [1] are composed of many low-power computer nodes; each node contains one or more sensors, a small processor and one or more methods of communication. Nodes collaborate to collect, process and distribute data from their own sensors as well as data from other nodes. The ultimate goal of a sensor network is to deliver a pertinent *summary* of the raw sensor data to a sink node or gateway. For example, raw temperature readings from sensors might be summarised into minimum, mean and maximum values. Raw data is not delivered to the gateway since either the available network bandwidth makes it infeasible or the power budget of the sensor nodes makes it undesirable.

When building sensor networks today, a programmer will typically take a sensor platform such as the Mica Mote and write code for the platform directly in a low-level language such as nesC [2]. This approach to systems building results in *early physical binding* since the programmer must decide at design time the places at which data is processed and summarised. This can lead to poor utilisation of resources, such as increased compute time, wasted network bandwidth or poor battery life, particularly if later changes to the architecture or application render early decisions inappropriate. This paper describes a system which separates application code from the description of the resources available to execute it. This allows *late physical binding* of the application since the components of the application can be optimised for execution on available resources after the components have been written or modified.

This work is not only relevant to sensor networks. Ubiquitous computing envisions an era when computers "weave themselves into the fabric of everyday life until they are indistinguishable from it" [3]. Such a system requires sensors to gather information about the real world in order to interact seamlessly with its inhabitants. In addition, our work is applicable to on-going research in Grid Computing [4] which utilises data and processing capabilities in many different physical locations and across organisational boundaries. Grid Computing enables scientists to write programs which are distributed over multiple computers and access repositories of data which are sufficiently large that moving programs onto processors near the data source is much easier than moving the data itself. We believe there is a strong analogy between the requirement to fix sensors at a particular physical location in a sensor network and the position of a petabyte data store in the Grid: both are infeasible to move.

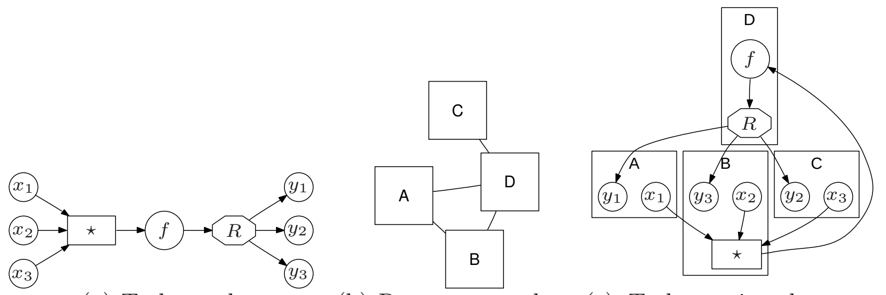
When introducing web services as components of the Semantic Web, Berners-Lee et al. describe a scenario where a patient wishes to book an appointment with a doctor [5], necessitating the sharing of calendar data. Determining what data should be transferred where is not obvious: for example, should appropriate diary times be transferred to the doctor's computer for comparison, or to the patient's? Here a qualitative measure of 'confidentiality' might replace latency or bandwidth as a metric for optimal placement.

Our approach allows application programmers in these fields to define the tasks which constitute a program separately from the topology of the network of processors and data sources. The tasks are described in such a way (Sect. 2) that permits late binding to processors by our compiler. Our language encourages programmers to identify particular kinds of program task to allow the program to be analysed and to determine whether particular program transformations can be performed safely. These transformations (Sect. 2.2) enable the program to be better-suited to execution in a particular network of processors. We observe that sensor data often forms a *monoid* and exploit this in both the language design and the optimisation framework. We then describe the syntax of the language (Sect. 3) and the implementation of the compiler (Sect. 3.3).

2 Computational Model

Applications in sensor networks usually involve executing a set of tasks to collate, process and distribute sensor-related information, where a task is a set of instructions which must be executed sequentially on a single processor. Our aim is to make tasks and the datatypes they operate over (i) simple to specify and (ii) structured so that program optimisations are possible.

Formally, we model the tasks representing an algorithm as a directed graph $G^{\mathrm{t}} = (E^{\mathrm{t}}, V^{\mathrm{t}})$ called the *task graph*. The set of vertices, V^{t}, are the tasks and



(a) Task graph (b) Resource topology graph (c) Tasks assigned to resources

Fig. 1. Example graphs.

the edges, E^t, indicate the direction of data flow between tasks. An edge (v_1, v_2) indicates that task v_2 receives the output of task v_1, and that v_2 cannot commence execution until the execution of v_1 is complete. An example is shown in Fig. 1(a). We define algorithms in terms of six kinds of task. These differ in terms of the type and number of inputs and outputs, and are sufficient to express any algorithm. We present them intuitively first, then explain the underpinning theory.

Source tasks are points where data is produced, drawn as circles. A source task has no inputs and one output. Although they only have one output edge, multiple values can be emitted in a sequential fashion. In other words, a source task produces a stream of values. For example, in a sensor network, a thermometer which outputs the temperature once per minute is modelled as a source task.

Sink tasks are points where data is consumed, drawn as circles. A sink task has one input and no outputs.

Processing tasks are functions which transform data of one type to another type, and are drawn as circles.

Merge tasks are functions which combine two items of data of a particular type into a single value of that type, and are drawn as rectangles. A merge task has two inputs, a single output and is commutative and associative. More than two items of data can be combined into a single value by chaining several merge tasks together in any order. For convenience, we draw a chain of merge tasks combining n items of data as a single n-ary task.
Merge tasks are particularly important in applications where a large number of input values from different sources need to be processed, such as sensor networks or grid computing. Because of the wealth of input data, it is usually necessary to be able to aggregate data into a significantly smaller amount of information to make their processing and interpretation more manageable.
Because merge nodes combine data which may have different data rates and sensor applications must be resilient to the failure of a subset of sensors,

each merge chain can be given a specified timeout after which it produces a result based on the available inputs.

Split tasks are functions which decompose a single item of data of a particular type into two values of that same type, and are drawn as rectangles. These values must be constructed such that, when fed into a merge task, the original item of data is yielded. Thus, split tasks can be thought of as the inverse of merge tasks. As with merge tasks, an item of data can be split into more than two parts by chaining several split tasks, and we draw such a chain as a single n-ary task.

Split tasks allow large items of data to be *partitioned* into smaller items so that computation can be performed in parallel. This permits a *divide-and-conquer* approach to data processing.

Replication tasks are functions which copy a value into a pair of identical values, and are drawn as octagons. A replication task thus has a single input and two outputs. As before, a chain of replication tasks can be constructed in order to generate more than two replicas of a value, and is drawn as a single n-ary replication task.

Datatypes are defined in terms of underlying sets from which values are drawn (e.g. the natural numbers) along with operations, or tasks, that may be performed on them.

Some datatypes which hold values of type T also have a binary operation $\star : T \times T \to T$ which is associative and commutative, and have an identity element $i \in T$ such that $\forall a \in T \,.\, i \star a = a$. The identity element denotes the datatype's 'empty' value. These datatypes are of particular interest because their operation $\star$ is equivalent to the definition of merge tasks above. Thus, using $\star$, several items of data of the same type can be combined into a single item of that type.

Such a datatype is modelled mathematically as a *commutative monoid* $(T, \star, i)$. Some examples of simple commutative monoids are set union $(\mathcal{P}(S), \cup, \emptyset)$, addition $(\mathbb{R}, +, 0)$ and maximisation $(\mathbb{R}, \max, -\infty)$. We refer to these datatypes as being *mergeable*. Associativity and commutativity reflect the idea of summarising data from a *set* of physically distributed sensors.

Since a split task for a particular datatype is an inverse of its merge task, it follows that mergeable datatypes necessarily support split operations. By analogy with the monoid $(\mathbb{N}, \times, 1)$, where merging is multiplication, splitting is factorisation into a pair of factors. Note that while splitting merely needs to be a right-inverse for merge, and therefore many split operations may exist for a given merge operation $\star$, we will nonetheless use the notation $\star^{-1}$.

In an application which processes data, it is not always enough to manipulate data within a single type, so functions $f : T_1 \to T_2$, where $T_1 \neq T_2$, are necessary in order to transform data into a new type. We refer to such functions as *processing* functions. An example of a processing function is list2hist which converts a multiset of temperature readings (encoded as a list) into a histogram.

A processing function f is a *monoid homomorphism* if it transforms data from one monoid $(S, \star, i_1)$ into data from another monoid $(T, \otimes, i_2)$ whilst satisfying two properties:

$$f(i_1) = i_2, \tag{1}$$
$$f(a \star b) = f(a) \otimes f(b). \tag{2}$$

An example of a monoid homomorphism is a function $f(x) = e^x$ from monoid $(\mathbb{N}, +, 0)$ to monoid $(\mathbb{R}, \times, 1)$. It is trivial to check that $f(0) = 1$ and $f(a+b) = f(a)f(b)$. A more realistic example is list2hist above.

Monoids (identifying merge tasks) and homomorphisms (enabling certain transformations—see below) are marked syntactically. Programmers are expected to identify which datatypes and functions are appropriate to treat in these ways. We believe that programmers will be able to easily identify these in everyday applications. In the worst case, when these are overlooked, this merely results in a smaller range of placement optimisations being available to the compiler.

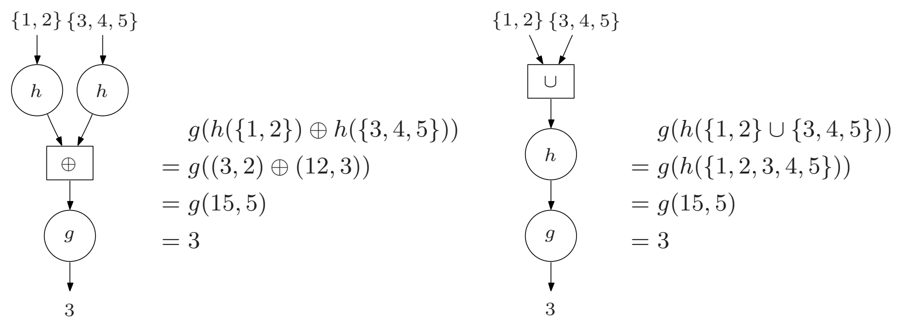
In practice, the constraints of computation mean that real implementations of datatypes are not necessarily perfect monoids. For example, addition and multiplication are only approximately associative in floating point arithmetic, thus we cannot faithfully implement the monoid $(\mathbb{R}, +, 0)$. Similarly, certain thresholding operations, e.g. $f(a) = \lfloor a \rfloor$, do not satisfy property (2) to be homomorphisms; timeouts further complicate the issue. Nevertheless, we expect programmers to identify these as monoids to reap the benefits that brings; a formal treatment would involve adding a metric space structure to monoids and adding a continuity requirement for homomorphisms and then to argue that the approximate behaviour is 'close enough' for a given application.

2.1 Example

In sensor networks, it is common to want to find the arithmetic mean of a large number of sensor readings. The centralised approach would gather and sum all the readings at the sink node and divide by the number of readings received. Partitioning the problem into smaller subsets of readings means that we can reach an answer using less energy or more quickly as several additions can be executed in parallel. However, the arithmetic means of arbitrary, distinct subsets of readings cannot be readily combined into the overall mean, because the number of readings contributing to each subset's mean is lost. A solution to this problem is to keep a running total of the number of readings in each partition. Adopting this approach, we can express the arithmetic mean of a set of numeric values by employing two processing functions—one a homomorphism, the other not.

A set of real numbers is represented by the monoid $(\mathcal{P}(\mathbb{R}), \cup, \emptyset)$. We use an intermediate monoid $(\mathbb{R} \times \mathbb{N}, \oplus, (0,0))$, where $(a_1, n_1) \oplus (a_2, n_2) \equiv (a_1 + a_2, n_1 + n_2)$, to store the numerator and denominator in the calculation of the arithmetic mean. The homomorphism to convert the set of numbers into this form is

$$h(\emptyset) = (0,0)$$
$$h(\{x\} \cup xs) = (x, 1) \oplus (h(xs)).$$



$$\begin{aligned} & g(h(\{1,2\}) \oplus h(\{3,4,5\})) \\ &= g((3,2) \oplus (12,3)) \\ &= g(15,5) \\ &= 3 \end{aligned}$$

(a) Processing before merging

$$\begin{aligned} & g(h(\{1,2\} \cup \{3,4,5\})) \\ &= g(h(\{1,2,3,4,5\})) \\ &= g(15,5) \\ &= 3 \end{aligned}$$

(b) Merging before processing

Fig. 2. Example task graphs for computing the arithmetic mean of two sets of values in a distributed fashion.

Values from this monoid can then be transformed into the desired result using a non-homomorphic function, $g(x, n) = x/n$. An example task graph for this application is depicted in Fig. 2(a). The sum and count of two sets of values are computed by h before being combined by the $\oplus$ merge task. Finally the mean is computed by g.

2.2 Program Transformation

An advantage of identifying datatypes which are monoids and processing functions which are monoid homomorphisms is that static analysis can be used to transform the program whilst maintaining semantic integrity.

For some programs, it is possible to express the graph of tasks in a variety of semantically-equivalent ways. For example, the task graph shown in Fig. 2(a) can be equivalently expressed as shown in Fig. 2(b). The former is a conversion to numerator-denominator pairs (*processing*) for both of the sets, followed by the summing function $\oplus$ (*merge*), and finally g. The latter is a union operation (*merge*) on the two sets, followed by the conversion by h to the numerator-denominator pair (*processing*), and finally g. The general form of this transformation is depicted graphically in Fig. 3. We refer to this transformation as Merge–Processing.

In general, we note that property (2) above implies that merging before processing will yield the same result as processing before merging *if and only if the processing function is a monoid homomorphism.* This means that it is useful for a programmer to be able to express to a compiler when a processing function is a homomorphism, so the compiler knows when the transformation can be applied and is guaranteed not to affect the semantics of the program.

In the cases where a processing function is not a homomorphism, information is lost when it is executed, and this means that performing merging before processing will not yield the same result as processing before merging. As noted earlier, functions merely approximating homomorphisms will in not in general

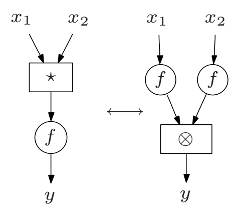


Fig. 3. Merge–Processing.

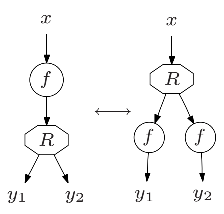


Fig. 4. Processing–Replication.

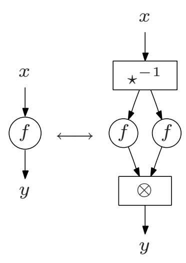


Fig. 5. Farm.

give bit-identical values, but sufficient for purpose if carefully marked as homomorphisms.

Using the Merge–Processing program transformation has several implications. Firstly, there are more tasks on the right side of the transformation than on the left. Depending on the computational complexity of f, $\star$ and $\otimes$, the overall amount of work involved may be different. Moreover, the volume of data flow may be affected by the transformation, depending on the relative sizes of the pre- and post-processing datatypes.

To exemplify these differences, consider an application which processes video data and extracts the number of people seen. If there are multiple video cameras, one distributed version of this application could involve appending all of the videos (*merging*) and then running the person-recognition algorithm on it (*processing*). Applying the transformation yields an alternative expression of the application in which the person-recognition algorithm (*processing*) is run on each individual video, and then the number of people are summed to a single value (*merging*). Since video data has a significantly higher data rate than the integer count, there is less network traffic required in the latter version of the algorithm.

2.3 Other Transformations

A second transformation, Processing–Replication, is similar to the transformation described above, but involves swapping the order of processing and replication tasks rather than processing and merge tasks. Rather than performing some processing and then replicating the result, we can replicate the input and process each replica individually. This transformation is depicted in Fig. 4. On the right, the amount of work is doubled and the volume of data flow may be affected.

The symmetry between split tasks and merge tasks gives rise to a transformation called Farm, depicted in Fig. 5. A processing task can be replaced by an array of processing tasks which each tackle a part of the input data. Although the transformation shows only two processing tasks, repeated application of the transformation can give rise to a larger number of processing tasks. This transformation facilitates the parallelisation of data processing, so is applicable to grid computing where a large problem is commonly divided into a number of

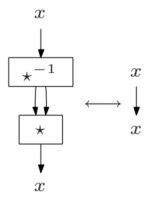


Fig. 6. Split–Merge.

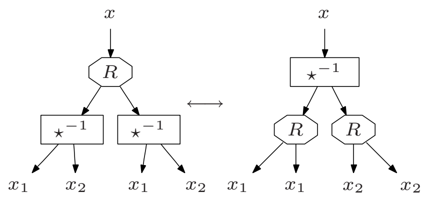


Fig. 7. Replication–Split.

smaller problems processed in parallel. This paradigm is familiar from popular distributed computing applications such as SETI@Home.

A further transformation, Split–Merge (Fig. 6), follows from the definition of a split task for a particular type as an inverse of the merge task for that type. Transformation Split–Merge is valid because $\forall x. \star(\star^{-1}(x)) = x$. The final transformation, Replication–Split, involves the exchange of replication and split tasks, shown in Fig. 7. This transformation also preserves the semantics of the program.

2.4 Task Assignment

As well as describing the program's algorithm as a task graph $G^{\mathrm{t}} = (E^{\mathrm{t}}, V^{\mathrm{t}})$, the network of processors in which it is to be executed must also be known. Program tasks can then be assigned to a processor for execution.

The network is modelled as a graph $G^{\mathrm{n}} = (E^{\mathrm{n}}, V^{\mathrm{n}})$, where now vertices V^{n} model processors and edges E^{n} model communication links between processors. An example is shown in Fig. 1(b). The processing nodes, which have local memory, are not assumed to be homogeneous in their processing power or communications capabilities.

An assignment function $A : V^{\mathrm{t}} \to V^{\mathrm{n}}$ maps tasks to processing nodes. Source and sink vertices in the task graph must be mapped to the particular nodes in the network where data is produced and consumed, respectively. Other tasks can be mapped to reachable network nodes. An example of an assignment function is shown pictorially in Fig. 1(c), where $\star$ is mapped to processor B and f and R are mapped to processor D.

The decision about which nodes to use impacts on the duration of execution of the algorithm; the privacy of the originators of the data; the amount of network bandwidth consumed; and a variety of other factors. The efficacy of the assignment is described quantitatively by a *cost function* specific to each application. A cost function $C : G^{\mathrm{t}} \times G^{\mathrm{n}} \times (V^{\mathrm{t}} \to V^{\mathrm{n}}) \to \mathbb{R}$ is a function of an assignment function yielding a real number indicating the cost of the assignment. Applications will use a cost function which embodies the trade-offs they desire between relevant metrics. Finding an optimal assignment with respect to a cost function is a well-studied research area [6] and is known to be NP-complete in general; we describe our strategy in Sect. 3.3.

So that a cost function can compute the values of relevant metrics, the graphs G^{t} and G^{n} must be weighted. Nodes of the resource graph (processors) are weighted with values describing their computational characteristics, such as processor speed. Edges of the resource graph (communication links) are weighted with values characterising the links, such as maximum throughput or latency. Nodes of the task graph (tasks) are weighted with values describing their requirements, such as the number of instructions constituting them. Edges of the task graph (data flow) are weighted with values characterising the data, such as the size of the data or its level of confidentiality.

3 Language

There are various ways in which the computational model described above could be encoded in a programming language. One approach is task-oriented, in which each processing and merge task is a first-class citizen. Instead, we chose a datatype-oriented approach in which merge tasks and processing tasks are encapsulated in the definitions of the datatypes they operate on. This approach ties in well with the modelling of some datatypes as commutative monoids, with their associated binary operation. For datatypes which can be modelled in this way, it is natural to encapsulate the underlying set, binary operation and identity element in a single logical unit. Processing tasks which can process data of a particular datatype are also encapsulated within that same logical unit. The datatype-oriented approach fits in well with object-orientation (our prototype implementation uses Java).

Along with the datatype definitions, the framework must be supplied with the task graph, the resource graph and a task assignment function. In the current implementation, these are defined in a *co-ordinator* file, although logically they need not be grouped together into a single module. The description of the cost function by which to evaluate an assignment function is described in a separate file. The resource graph is not required at design time; rather, it must be provided just before compilation. Presently, the program must be recompiled whenever the topology changes.

Our compiler uses the task graph and resource graph in the co-ordinator definition in order to derive a total task assignment function mapping each task to the processor found to be most appropriate to execute it. The compiler can then distribute the merge tasks and processing tasks contained in the datatype declarations to the chosen processors.

3.1 Datatype Declarations

Each datatype is defined in its own file and has a syntax built on top of a Java class in our current implementation; this could easily be adapted for use with other languages. Metrics that are used by the cost function to evaluate a mapping are also specified in this file. A datatype is declared using the `datatype`

```
mergeable datatype PartialAv {
    private double numer;
    private int denom;

    public PartialAv() [cpu=0, out size=1] {              Identity element, (0, 0).
        this(0, 0);
    }

    public PartialAv(double numer, int denom) {           Singleton constructor
        this.numer = numer;                               for choosing a value
        this.denom = denom;                               from ℝ × ℕ.
    }

    public PartialAv merge(PartialAv a)                   Merge function, ⊕.
        [cpu=1, out size=sum]                             Cost annotation.
    {
        return new PartialAv(this.numer + a.numer,
            this.denom + a.denom);
    }
                                                          Processing function, g,
    processto Average [cpu=1, out size=1] {               returning an instance of
        return new Average(this.numer / this.denom);      the Average datatype,
    }                                                     not defined here.
}
```

Fig. 8. Datatype declaration representing the monoid $(\mathbb{R} \times \mathbb{N}, \oplus, (0,0))$.

keyword. To facilitate physical redistribution, static methods and static fields (except public static final fields) are not permitted in datatype declarations.

As described above, some datatypes are *mergeable*. Declarations of such datatypes use the `mergeable` modifier to indicate this. Figure 8 shows the datatype declaration for the monoid $(\mathbb{R} \times \mathbb{N}, \oplus, (0,0))$ used in the arithmetic mean example in Sect. 2.1, which is a typical mergeable datatype. The use of the `mergeable` modifier entails three requirements: (1) The datatype is a monoid so must have an identity element. This is implemented by requiring that mergeable datatypes support a constructor which takes no arguments. (2) It must also have a constructor to create a *singleton* instance of the datatype. In other words, a means of wrapping a single element from the underlying set of values must be provided, to allow new items of data to be instantiated. (3) The binary merge operation must be specified. This is implemented by requiring that mergeable datatypes of type α support a publicly visible method `merge` which takes an argument of type α and returns a value of type α. The merge function is specified such that the expression $a = a_1 \star a_2$ can be expressed in the fashion `a = a1.merge(a2)`.

Some datatypes define an operation to split them into a pair of smaller elements. Whilst it is necessarily true that all mergeable datatypes are also splittable in theory, it may be that the algorithm for implementing splitting is significantly harder to implement than merging. For example, in the monoid $(\mathbb{N}, \times, 1)$, merging is multiplication (easy) but splitting is factorisation (hard). It is also conceivable that the converse is true for some datatypes: it may be much easier to express a split operation than a merge operation. Therefore, it is not mandatory that mergeable datatypes support a split operation. The `splittable` modifier is used on datatypes which implement a `split` operation as a publicly visible method which returns a pair of items (currently implemented as an array). Programmers of datatypes that are both mergeable and splittable need to ensure that the split operation is the inverse of the merge operation; in general it is undecidable for a compiler to check this statically, but it can be easily unit tested.

In the computational model, a processing task transforms data of one type into another type. Each datatype thus has zero or more other datatypes into which it can be processed. For each such possibility, the datatype declaration contains the code describing the processing task. These are defined in `processto` functions, which must each return an object of the target type. In our Java-based implementation, this is implemented in the 'source' datatype's declaration rather than as a constructor in the 'destination' datatype's declaration so that private members of the source datatype can be accessed. Note that the presence of a processing function in a datatype's declaration does not imply that it will necessarily be part of a task graph; it merely indicates that such a function exists. Processing functions which are monoid homomorphisms are marked with the `homomorphism` keyword, to notify the compiler that transformations appropriate to homomorphisms can be safely applied in applications using this function.

Annotations describing the values of various metrics that are employed by the cost function are required for `processto` functions, merge functions, split functions and constructors which are used as source tasks. The metrics are specified as a comma-separated list of (*key*, *value*) pairs, enclosed in square brackets, where the key is a string known to the cost function and the value is a simple arithmetic expression. Keys may include the `out` modifier to indicate that they are metrics characterising data on egress edges from the corresponding node in a task graph. Other values characterise the node itself. For processing functions, egress edge values may use the special value `in` to refer to the value of the input for the corresponding key. For merge functions, egress edge values may use the special values `sum`, `max`, `min`, `avg` to refer to the sum, minimum, maximum or average of the input values for the corresponding key.

For example, a merge task may be annotated with `cpu=50, out size=sum, out privacy=max` to indicate its CPU load; that the size of the output is the sum of the sizes of its inputs; and that the degree of sensitivity with respect to privacy is the largest such from among its inputs.

It is necessary for these annotations to be attached to the definitions of the functions, rather than the task graph, because the compiler is free to apply transformations to the task graph, and needs to know the values of the metrics on nodes and edges which it creates in the graph.

3.2 Co-ordinator Definition

The cornerstone to the programmer's description of an application is the definition of the *co-ordinator*. This file contains the ingredients which describe the application with sufficient flexibility to allow the compiler to determine a strategy for executing it. Although we implement datatype declarations using a Java-

```
coordinator CoordAv {
    taskgraph {
        source<TempSet> c0 ["/dev/ttyS0"], c1 ["/dev/ttyS0"], c2 ["/dev/ttyS0"];
        merge<TempSet> m0 [1 => inf, 2 => inf, 3 => inf];
        process<TempSet, PartialAv> p0;
        process<PartialAv, Average> p1;
        sink<Average> s0;

        c0 -> m0; c1 -> m0; c2 -> m0;
        m0 -> p0;
        p0 -> p1;
        p1 -> s0;
    }

    resourcegraph {
        sensor0: 192.168.0.100 [speed => 2];
        sensor1: 192.168.0.101 [speed => 2];
        sensor2: 192.168.0.102 [speed => 2];
        host3:   192.168.0.103 [speed => 10];

        sensor0 -- host3   [bandwidth => 5, latency => 1];
        sensor1 -- sensor0 [bandwidth => 1, latency => 1];
        sensor2 -- host3   [bandwidth => 5, latency => 1];
    }

    mapping {
        c0 -> sensor0;
        c1 -> sensor1;
        c2 -> sensor2;
        s0 -> host3;
    }
}
```

Fig. 9. Example co-ordinator definition for the temperature-averaging application.

like syntax, the co-ordinator language is largely independent of that used in the rest of the application. An example co-ordinator for the temperature-averaging application for three sensors is shown in Fig. 9.

A co-ordinator is defined using the `coordinator` keyword. It contains three kinds of definition:

Initial task graph. The `taskgraph` block is used to specify an initial task graph, giving to each task an identifier which has scope throughout the co-ordinator definition. The links between tasks, indicating the direction of data flow, are also specified.

Source tasks can optionally be given a list of arguments which are to be passed to the constructor of its datatype, if any are required. In a sensor network, the source tasks generate the application's input data, so the arguments can be used to create an instance of the source datatype appropriate to each source task.

Merge tasks specified in the task graph can be annotated with an array of timeouts. For an n-ary merge task, timeouts are specified for each number of potential inputs received from 1 to n. The timeout for m inputs indicates the longest duration of time the task should wait after having received $m-1$

input values for the next. The special value `inf` denotes an infinite duration, implying that it is not acceptable for the merge task to produce an output without having received further input values. The value 0 denotes that no further inputs are necessary. If no timeouts are specified, it is assumed that the timeout for all numbers of inputs up to and including n are infinite. (Infinite timeouts are generally undesirable in sensor networks as the failure of one sensor should not prevent the system from producing output.)

Resource graph. The `resourcegraph` block defines the processors in the network (four are used in Fig. 9) and the connections between them. An identifier and the hostname is specified for each, along with the values of metrics that are employed by the cost function. For each communication link, values of metrics that are employed by the cost function are specified.

Base mapping. The `mapping` block specifies an assignment function from tasks to processors. The assignment function derived by a compiler is only permitted to be a superset of the mapping specified here. This is particularly relevant in sensor networks, where a source task must execute on a particular processor because it has the sensor to be sampled, and where a sink task must execute on a particular processor because it needs to know the result of the processing. However, this feature can also be used by the programmer to lock other code to a particular processor if desired.

3.3 Current Implementation

In our current implementation we use Polyglot [7] to translate the co-ordinator definition to instantiate appropriate instances of datatype declarations and generate standard Java source code. The Java source code is then compiled using a conventional Java compiler, producing a JAR file for each processor.

During compilation, the compiler analyses the task graph and the resource graph to determine the best locations to execute the tasks. As part of this process, the annotations associated with processors and communication links in the co-ordinator file are combined with the cost function to determine the relative suitability of any particular mapping of tasks to processors. As noted earlier this is a well-studied area, and we have not developed a state-of-the-art assignment algorithm, but have used a simple technique [8] to explore both task assignment and program transformation simultaneously. In our solution, we initially assign all unmapped tasks to reachable nodes with the largest aggregate connectivity. We then use a method of steepest descent to iteratively search for improvements in two phases: firstly, we determine all possible immediate program transformations; secondly, for every program transformation we consider moving each task in turn to alternative processors in the resource graph. Finally we select the program transformation and task movement combination with the lowest cost as the starting point for our next search iteration. We terminate our search when no further improvements can be found.

The current implementation automatically generates Java RMI code to enable inter-task communication. For sensor networks, RMI is not ideal, since this approach requires a central RMI registry to be present, but this could easily be replaced by an alternative communications paradigm.

4 Related Work

In Web Services, the Business Process Execution Language (BPEL) [9] is used to describe high-level 'business protocols': stateful workflows between web services. This standard aims to separate the deployment information (where the services are executed) from the description of the protocol. Thus, an application specified in BPEL supports late binding to physical resources, as any resource supporting a particular interface could be employed to execute a task.

The Grid Computing paradigm tends to identify networks of computers as either *compute grids* or *data grids*. Compute grids involve participating computers running an execution environment into which jobs are sent by a co-ordinator, to allow an application to benefit from parallelisation. Data grids are common in the scientific community where a large corpus of data is made available to collaborators across the globe. In data grids, the question of where data integration and processing is done is paramount. So as not to incur large volumes of network traffic, processing is moved close to the data. The OGSA-DAI framework [10] achieves this using a scripting language whose programs are sent over the grid and executed close to the data. The idea of moving processing close to a data source is a particular case of the general principle of optimising the arrangement of tasks envisioned in this paper; a fixed sensor node can be thought of in the same way as a large, immovable corpus of data.

Ennals et al. describe an approach to programming network processors using a domain-specific language PacLang which permits the description of applications in an architecturally-neutral fashion [11]. An Architecture Mapping Script describes which core should execute which application task, in a similar fashion to a co-ordinator file. Furthermore, a set of transformations can be applied to enable programs to be partitioned into different arrangements of tasks, to allow the program to be better-suited to execution on a particular architecture. This work exploited linear types as the basis for the transformations; we have adopted the approach of modelling datatypes as monoids to similar effect, although these approaches are not mutually exclusive.

Program transformations are also exploited to aid task assignment in distributed query processing. It is the job of a query optimiser [12] to choose the best strategy for executing a distributed database query; this may involve rewriting the query.

Kremer et al. have implemented a compiler framework to allow an application which would otherwise run solely on a mobile device to be off-loaded onto a server [13]. Similar work has been undertaken by Li et al. at the function-call level [14] and by Ou et al. at the Java bytecode level [15]. These are specific instances of the kind of application considered in this paper, but they do not consider whether applications can be transformed to permit parallelised execution.

In sensor networks, J-Orchestra [16] is a system which automatically partitions applications into tasks, and allows developers to manually assign tasks amongst machines. The Titan framework [17] has been designed to permit dynamic reconfiguration of which processors execute which tasks for body-area sensor networks.

Some of the theoretical underpinnings of our work were greatly inspired by Afshar's use of monoids and monoid homomorphisms in parallel data-processing applications [18].

5 Conclusion

We have created a language which can be used to write applications for distributed systems. The separation of the task definitions from a notion of where in the system they are to be executed allows a compiler to derive a mapping of tasks to processors. This mapping can be improved by performing various program transformations that change the task graph. Programmers are encouraged to express datatypes as monoids and functions as homomorphisms to enable safe use of a wider range of task placement transformations.

We have approached this work from the direction of sensor networks, but we believe that our ideas are more globally applicable. It is already evident that some concepts are readily applicable to Ubiquitous Computing, Grid Computing and Web Services. We hope that this work is a stepping-stone towards a full calculus with primitives that encompass all of these paradigms.

We have assumed a static network topology. In practice, many sensor networks contain mobile nodes, meaning that an initially optimal assignment of tasks may quickly become sub-optimal. Similarly, in practice, nodes and communication links will fail. A simple adaptation of our work would be to collect nodes into logical groups from which nodes can leave and join, but where the characteristics of each group remain largely constant. In addition, we have assumed a single, omniscient co-ordinator, which is impractical, and therefore we plan to investigate a distributed approach to co-ordination.

The examples used in this paper have been kept simple to ease comprehension. We believe that it is more generally applicable to larger applications; the application which inspired this work was the automatic generation of road maps based on sensor data collected from road vehicles [19]. This application was originally implemented in a non-distributed fashion in Java; we realised that in order to make it distributed a lot of boiler-plate code would be required which could be generated automatically. Furthermore, we realised that it was inappropriate to make task placement decisions at design-time. We are in the process of re-implementing this application using our framework.

Acknowledgments. The authors gratefully acknowledge the support and vision of Andy Hopper and comments and suggestions from Andrew Rice and the anonymous reviewers. We are also grateful for the financial support provided by EPSRC.

References

1. Akyildiz, I.F., Su, W., Sankarasubramaniam, Y., Cayirci, E.: Wireless sensor networks: a survey. Computer Networks **38** (2002) 393–422
2. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesC language: A holistic approach to networked embedded systems. In: Proceedings of Programming Language Design and Implementation (PLDI). (2003) 1–11
3. Weiser, M.: The computer for the 21st century. Scientific American **365**(3) (1991) 94–104
4. Foster, I., Kesselman, C., Nick, J.M., Tuecke, S.: Grid services for distributed system integration. IEEE Computer **35**(6) (2002) 37–46
5. Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web. Scientific American **284**(5) (2001) 28–37
6. Kwok, Y.K., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Computing Surveys **31**(4) (1999) 406–471
7. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An extensible compiler framework for Java. In: Proceedings of the 12th International Conference on Compiler Construction. Volume 2622 of LNCS. (2003) 138–152
8. Davies, J.J., Beresford, A.R.: Scalable, inter-vehicular applications. In: On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops (Part II). Volume 4806 of LNCS. (2007) 876–885
9. Business Process Execution Language for Web Services: Version 1.1. `http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/` (2003)
10. Antonioletti, M., Atkinson, M., Baxter, R., Borley, A., Chue Hong, N.P., Collins, B., Hardman, N., Hume, A.C., Knox, A., Jackson, M., Krause, A., Laws, S., Magowan, J., Paton, N.W., Pearson, D., Sugden, T., Watson, P., Westhead, M.: The design and implementation of grid database services in OGSA-DAI. Concurrency and Computation: Practice and Experience **17** (2005) 357–376
11. Ennals, R., Sharp, R., Mycroft, A.: Task partitioning for multi-core network processors. In: Proceedings of the 14th International Conference on Compiler Construction. Volume 3443 of LNCS. (2005) 76–90
12. Ioannidis, Y.E.: Query optimization. ACM Computing Surveys **28**(1) (1996) 121–123
13. Kremer, U., Hicks, J., Rehg, J.H.: A compilation framework for power and energy management on mobile computers. Technical Report DCS-TR-446, Rutgers University (2001)
14. Li, Z., Wang, C., Xu, R.: Computation offloading to save energy on handheld devices: A partition scheme. In: CASES 2001, ACM Press (2001) 238–246
15. Ou, S., Yang, K., Liotta, A.: An adaptive multi-constraint partitioning algorithm for offloading in pervasive systems. In: PerCom 2006. (2006) 116–125
16. Liogkas, N., MacIntyre, B., Mynatt, E.D., Smaragdakis, Y., Tilevich, E., Voida, S.: Automatic partitioning for prototyping ubiquitous computing applications. IEEE Pervasive Computing **3**(3) (2004) 40–47
17. Lombriser, C., Roggen, D., Stäger, M., Tröster, G.: Titan: A tiny task network for dynamically reconfigurable heterogeneous sensor networks. In: Kommunikation in Verteilten Systemen (KiVS), Springer (2007) 127–138
18. Afshar, M.: An Open Parallel Architecture for Data-intensive Applications. PhD thesis. Technical Report UCAM-CL-TR-459, University of Cambridge (1999)
19. Davies, J.J., Beresford, A.R., Hopper, A.: Scalable, distributed, real-time map generation. IEEE Pervasive Computing **5**(4) (2006) 47–54