(12) UK Patent Application (19) GB (11) 2569430 (13) A

(43) Date of A Publication 19.06.2019

(21) Application No: 1816892.2

(22) Date of Filing: 17.10.2018

(30) Priority Data:
(31) 1717295 (32) 20.10.2017 (33) GB

(71) Applicant(s):
Graphcore Limited
6th Floor, 11-19 Wine Street, BRISTOL, BS1 2PH, United Kingdom

(72) Inventor(s):
Simon Christian Knowles
Daniel John Pelham Wilkinson
Richard Luke Southwell Osborne
Alan Graham Alexander
Stephen Felix
Jonathan Mangnall
David Lacey

(74) Agent and/or Address for Service:
Page White & Farrer
Bedford House, John Street, London, WC1N 2BF, United Kingdom

(51) INT CL:
G06F 9/52 (2006.01) ***G06F 9/30*** (2018.01)
G06F 15/173 (2006.01)

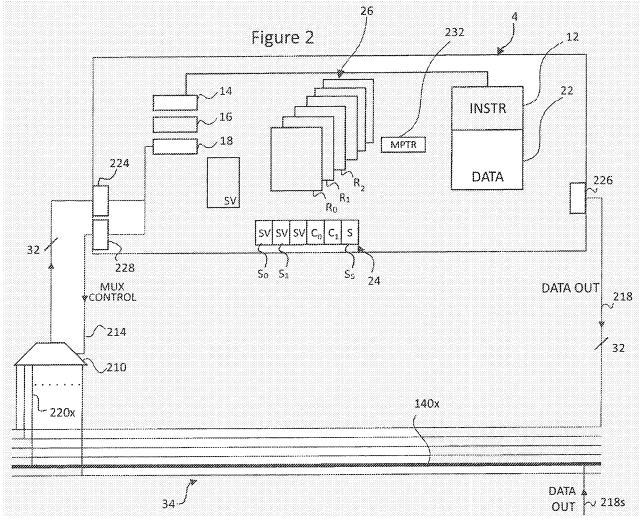
(56) Documents Cited:
US 5754789 A **US 5434861 A**
US 20140006724 A1

(58) Field of Search:
INT CL **G06F, G06N**
Other: **EPODOC, WPI, INSPEC, Patent Fulltext, XPESP, XPIEE, IP.COM, XPI3E, XPMISC, XPLNCS, XPRD, XPSPRNG, TDB**

(54) Title of the Invention: **Synchronization in a multi-tile processing array**
Abstract Title: **Data synchronisation between processing units governed by common clock**

(57) Computer comprising processing units 4 each having instruction storage 12, execution unit, data storage 22 and input 224 and output 226 interfaces comprising sets of wires; switching fabric 34 connected to each processing unit by the respective output wires and connectable to each of the processing units by the respective input wires via switching circuitry controllable by each processing unit; synchronisation module to generate a synchronisation signal to control the computer to switch between computer phase and exchange phase; wherein the processing units execute local programs according to a common clock, the local programs such that in the exchange phase at least one processing unit executes a send instruction from its local program to transmit onto its output set of connection wires at a transmit time a data packet destined for a recipient processing unit but having no destination identifier, and at a predetermined switch time the recipient processing unit controls its switching circuitry to connect its input set of wires to the fabric to receive the packet at a receive time; the transmit, switch and receive times being governed by the common clock with respect to the synchronisation signal. Also provided is a method of computing a function.

Figure 2



GB 2569430 A

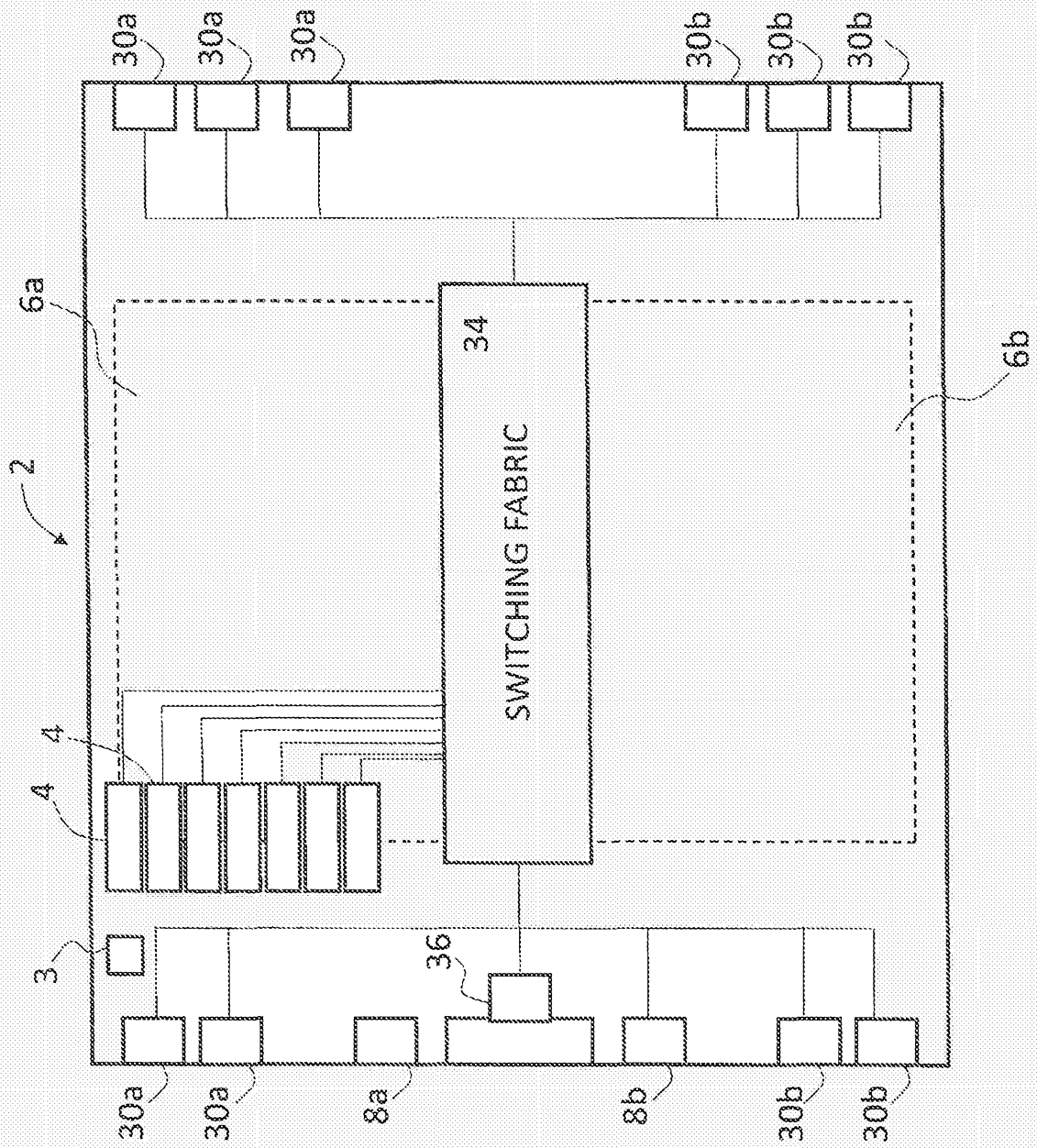

30a
30a
30a
30b
30b
30b
6a
34
SWITCHING FABRIC
6b
2
4
4
3
36
30a
30a
8a
8b
30b
30b

Figure 1

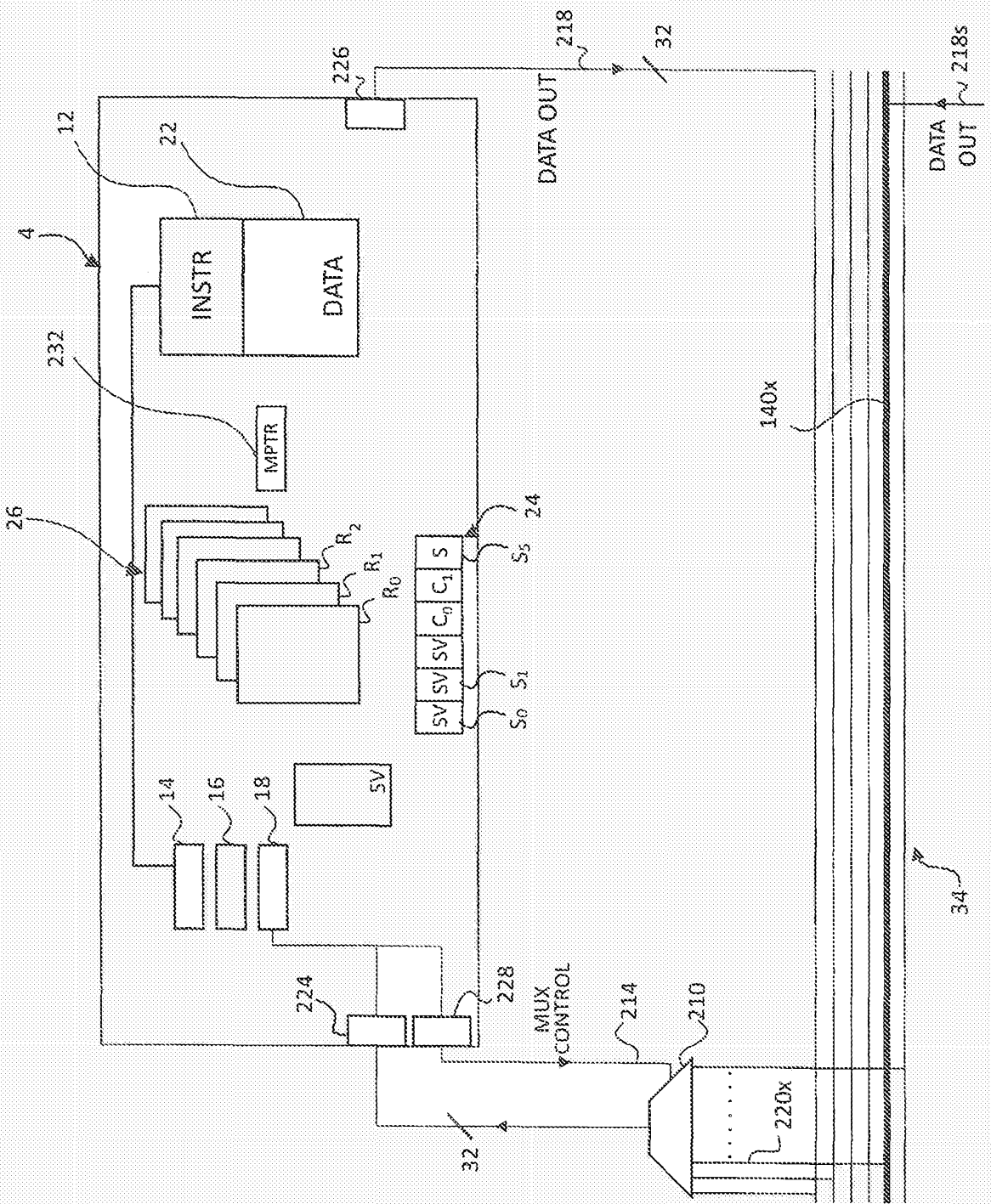

4
12
22
INSTR
DATA
232
MPTR
26
R_2
R_1
R_0
24
S_5
S_1
S_0
SV
SV
SV
C_0
C_1
S
5V
14
16
18
224
228
MUX CONTROL
214
210
220x
32
32
226
DATA OUT
218
140x
DATA OUT
218s
34

Figure 2

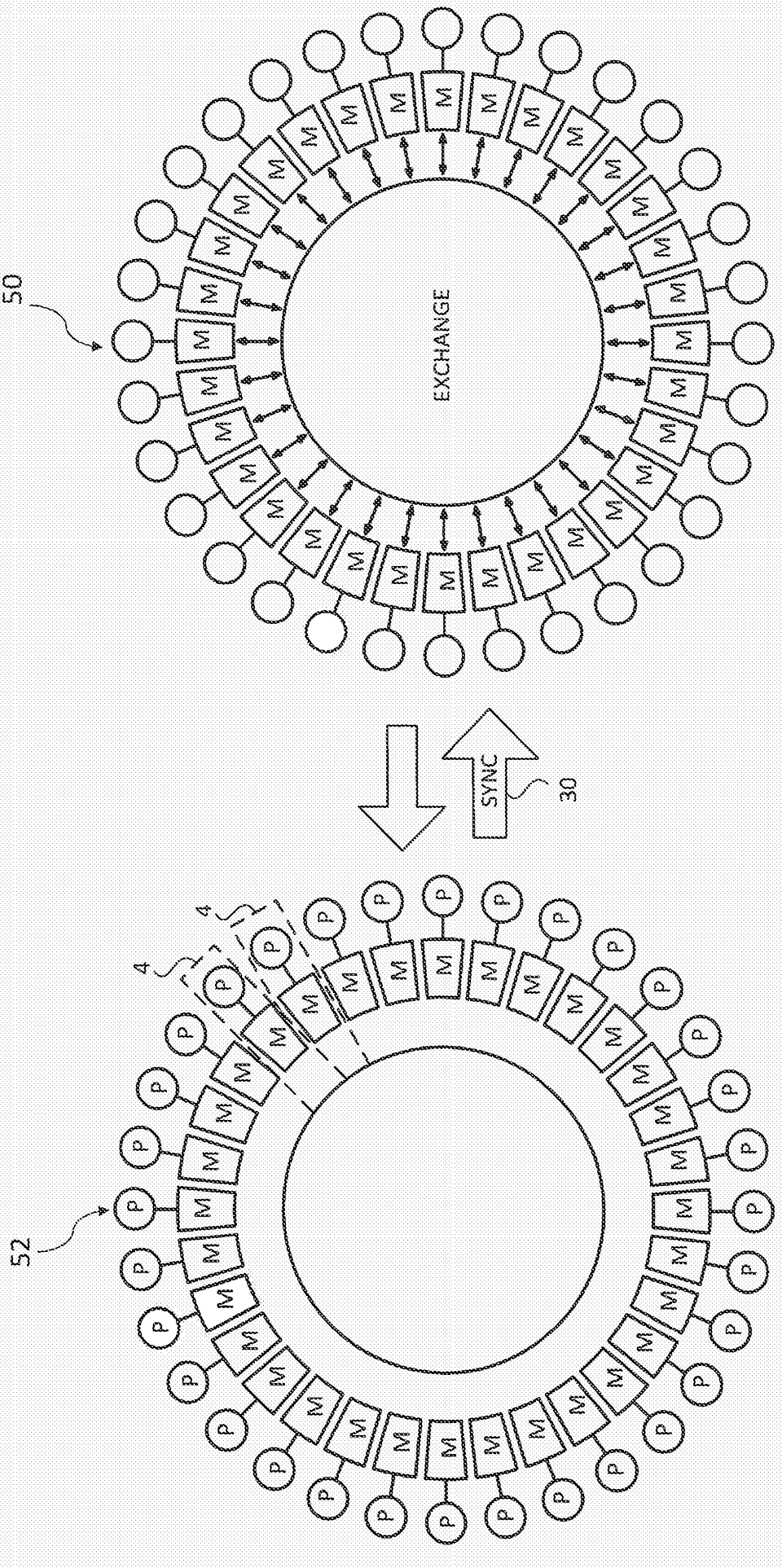


Figure 3

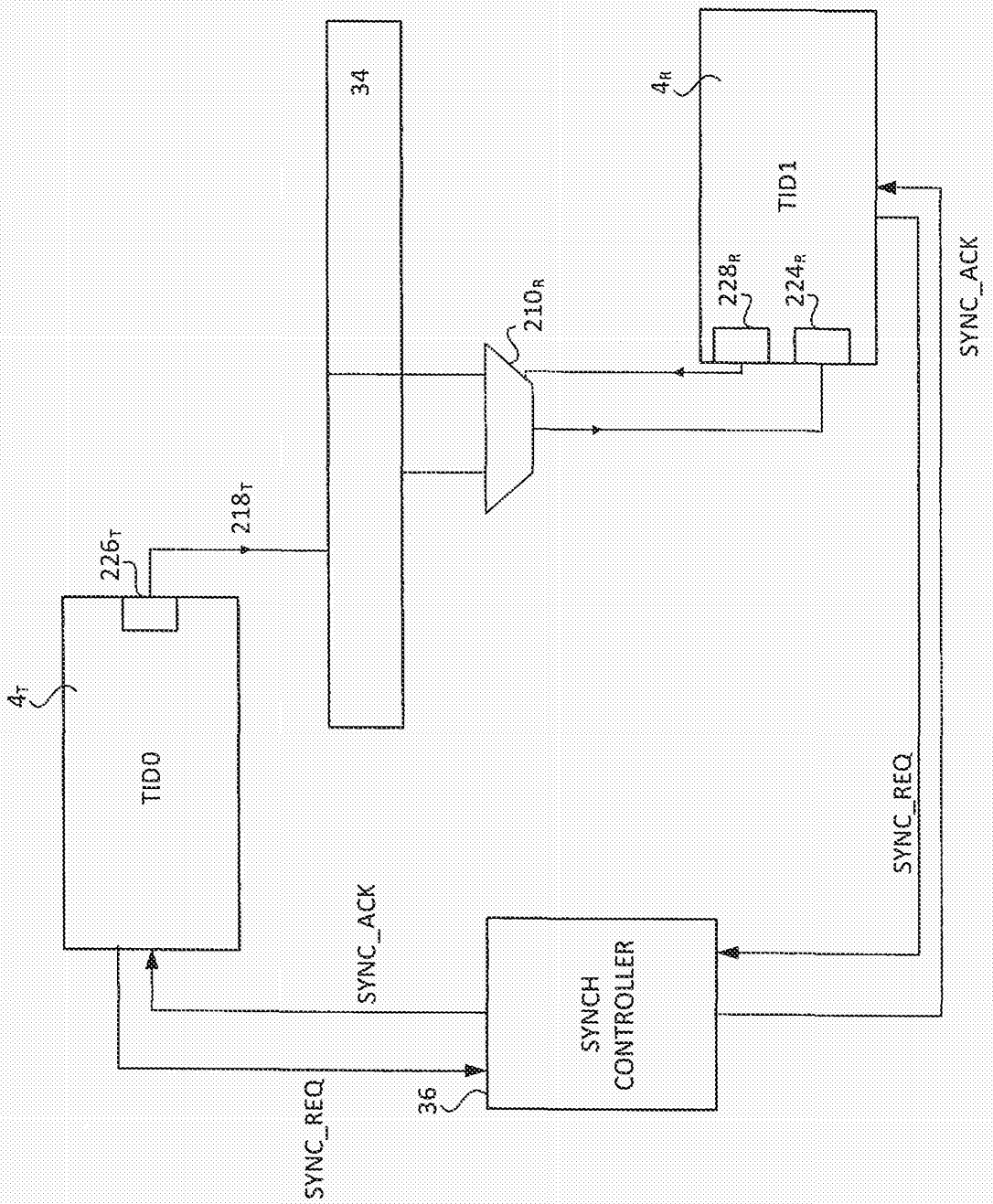

34
4_R
TID1
228_R
224_R
210_R
SYNC_ACK
226_T
218_T
4_T
TID0
SYNC_ACK
SYNC_REQ
SYNCH CONTROLLER
36
SYNC_REQ

Figure 4

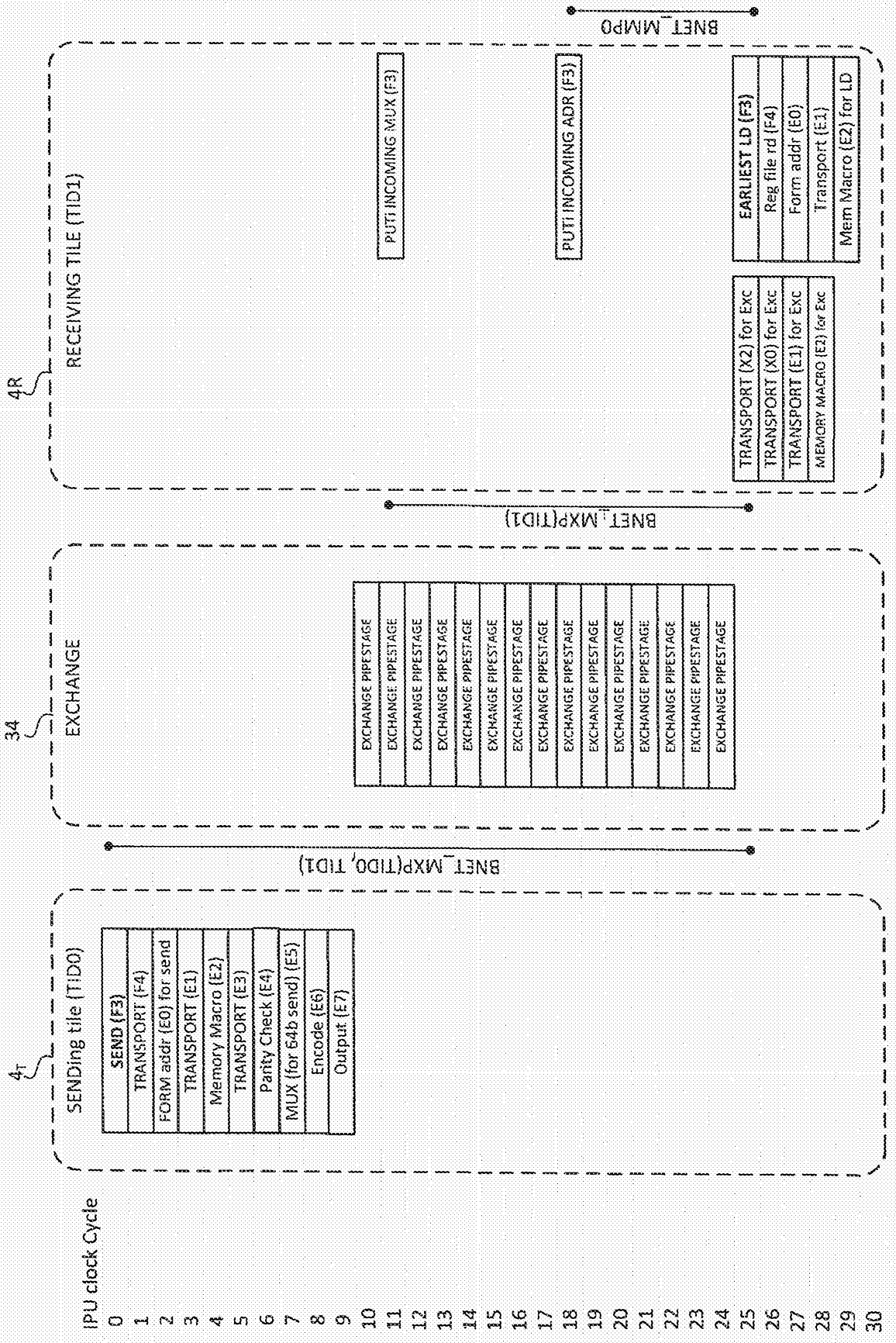

IPU clock Cycle
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
4T
SENDing tile (TID0)
SEND (F3)
TRANSPORT (F4)
FORM addr (E0) for send
TRANSPORT (E1)
Memory Macro (E2)
TRANSPORT (E3)
Parity Check (E4)
MUX (for 64b send) (E5)
Encode (E6)
Output (E7)
BNET_MXP(TID0, TID1)
34
EXCHANGE
EXCHANGE PIPESTAGE
BNET_MXP(TID1)
4R
RECEIVING TILE (TID1)
PUTI INCOMING MUX (F3)
PUTI INCOMING ADR (F3)
BNET_MMP0
EARLIEST LD (F3)
Reg file rd (F4)
Form addr (E0)
Transport (E1)
Mem Macro (E2) for LD
TRANSPORT (X2) for Exc
TRANSPORT (X0) for Exc
TRANSPORT (E1) for Exc
MEMORY MACRO (E2) for Exc

Figure 5

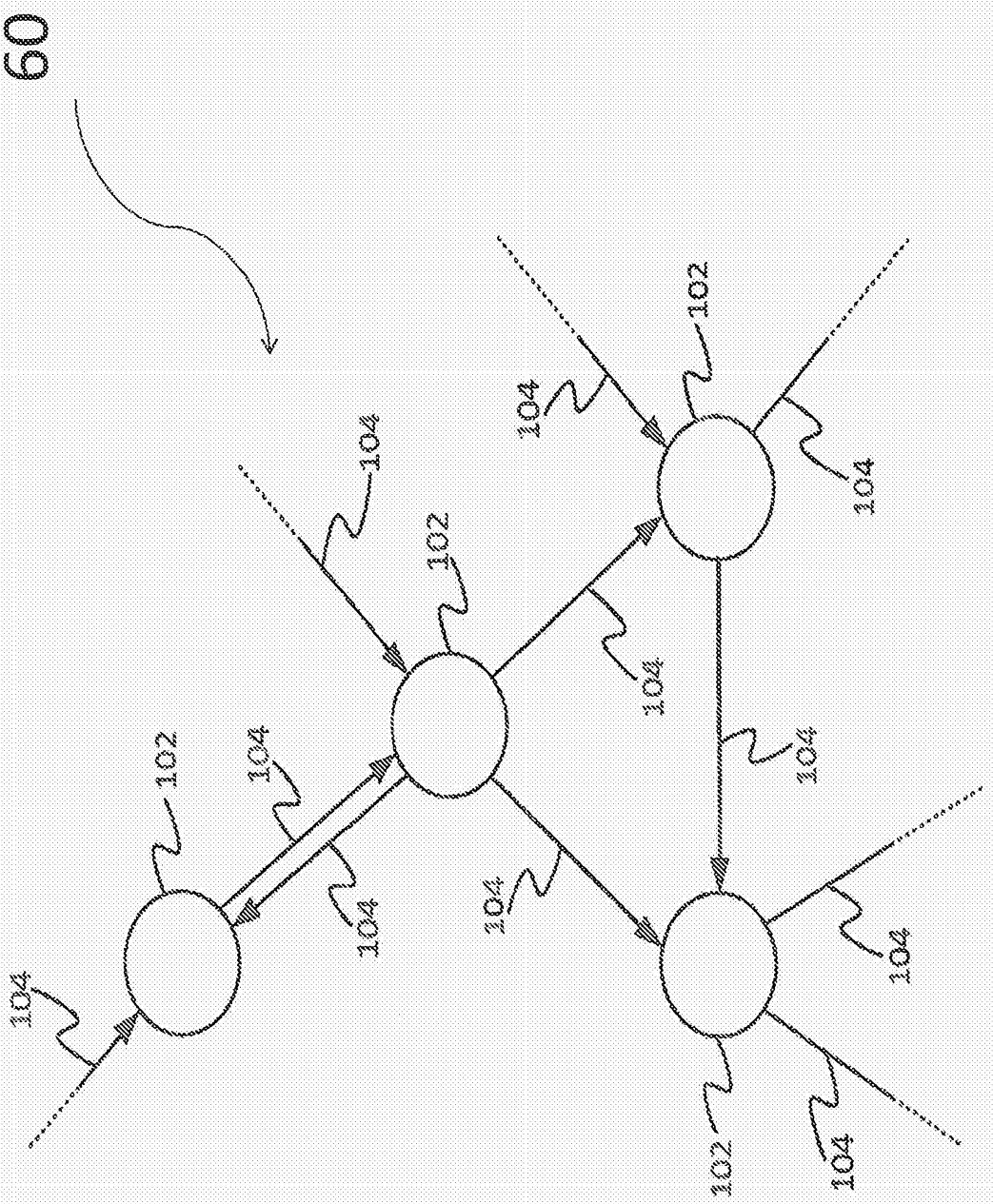


Figure 6

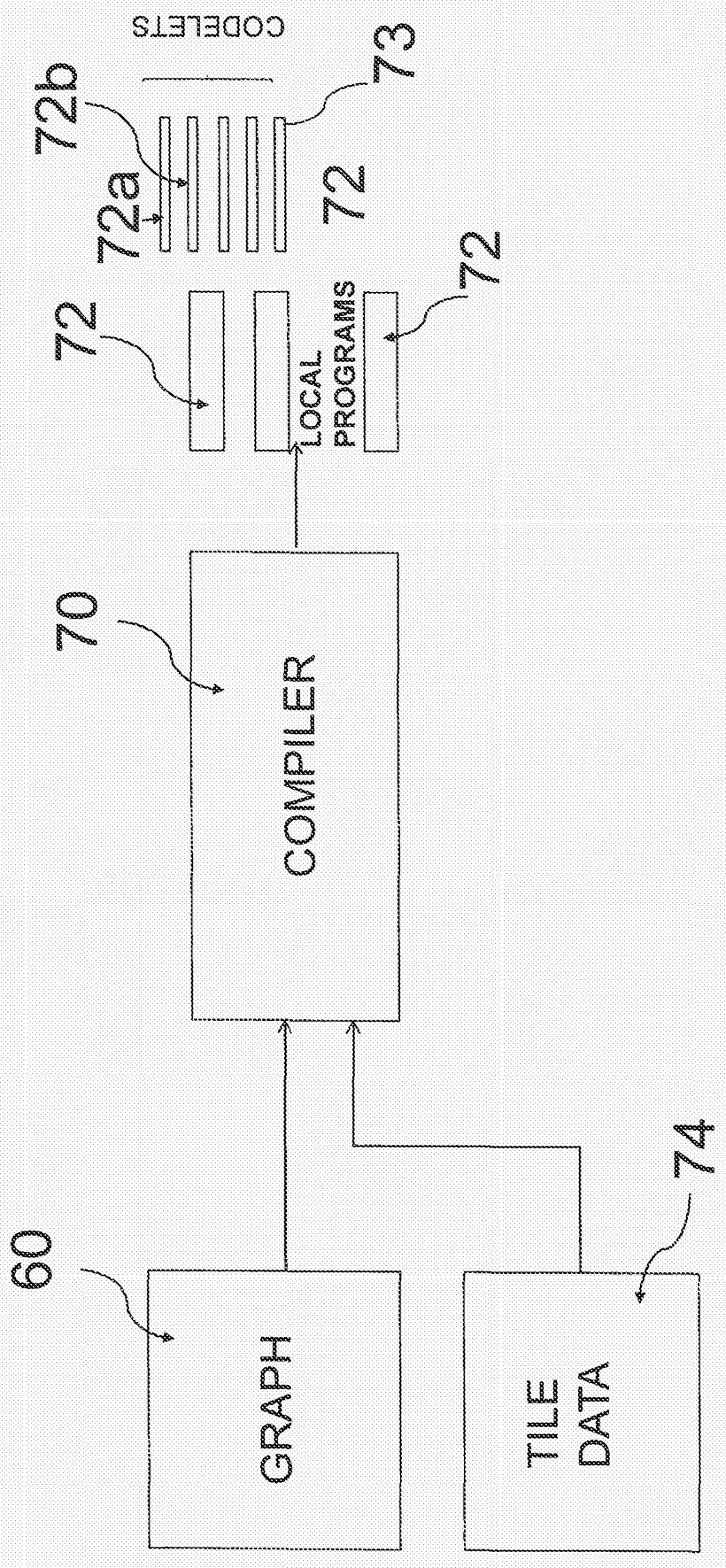
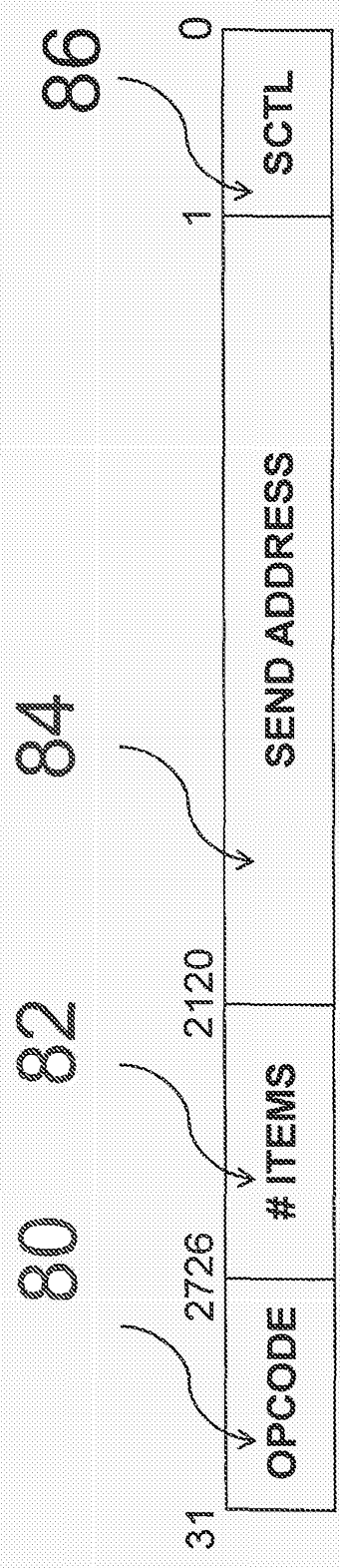
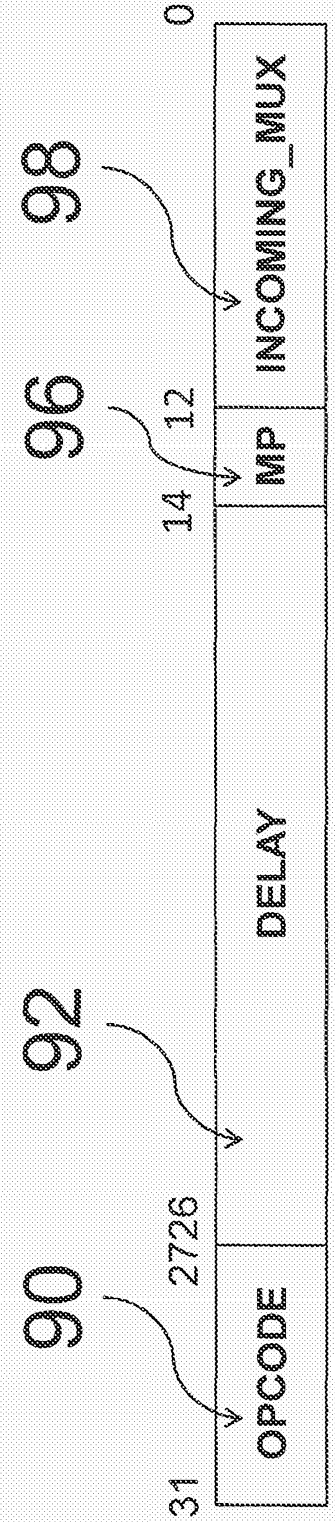

CODELETS
72b
73
72a
72
72
72
LOCAL PROGRAMS
70
COMPILER
60
GRAPH
74
TILE DATA

Figure 7



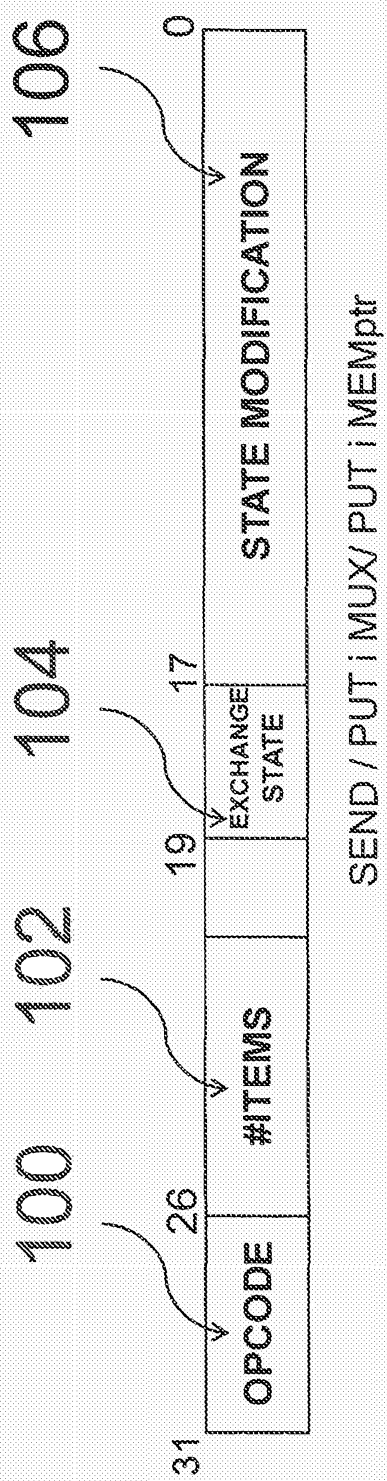
SEND

Figure 8



PUT i - MUXptr

Figure 9



SEND / PUT i MUX/ PUT i MEMptr

Figure 10

EXCH

Figure 11

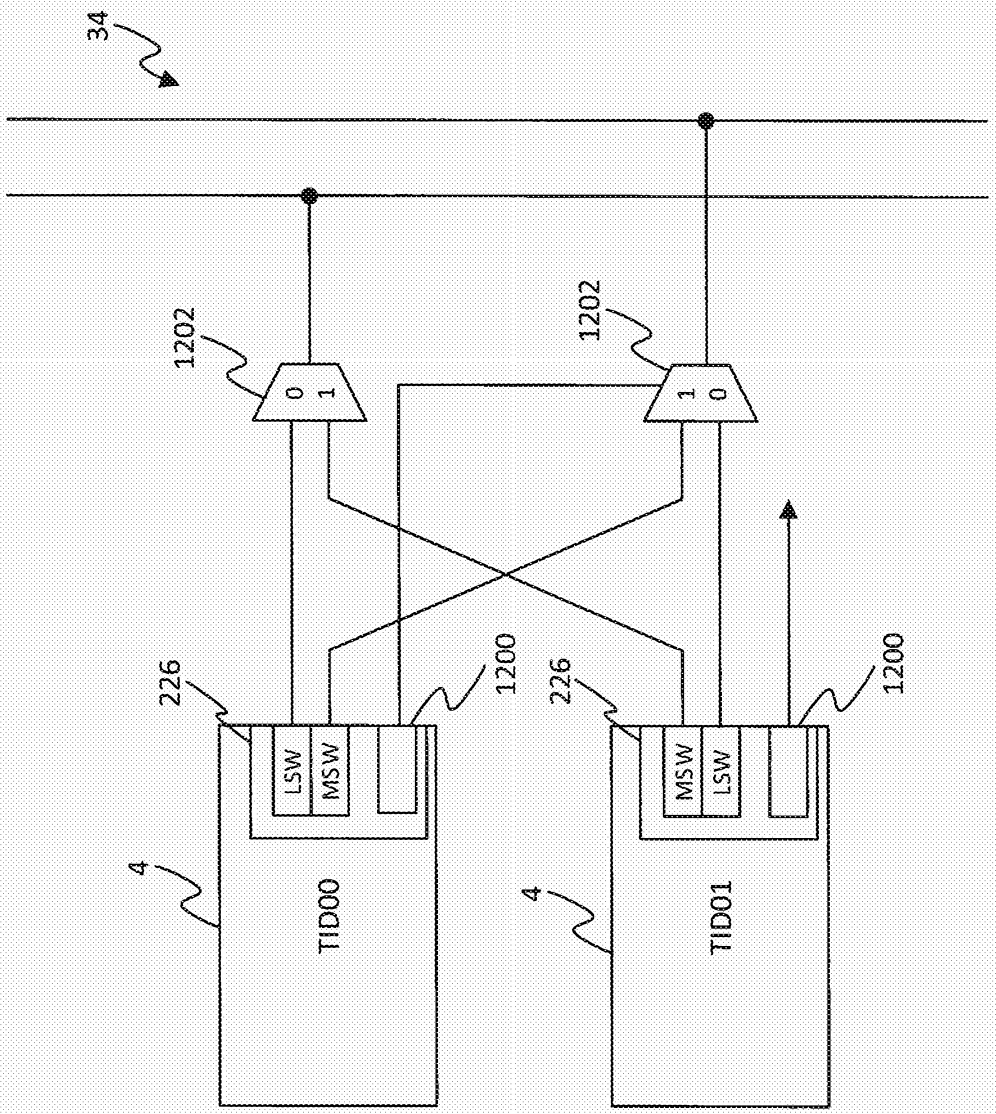

34
1202
0
1
1202
1
0
226
1200
LSW
MSW
4
TID00
226
1200
MSW
LSW
4
TID01

Figure 12

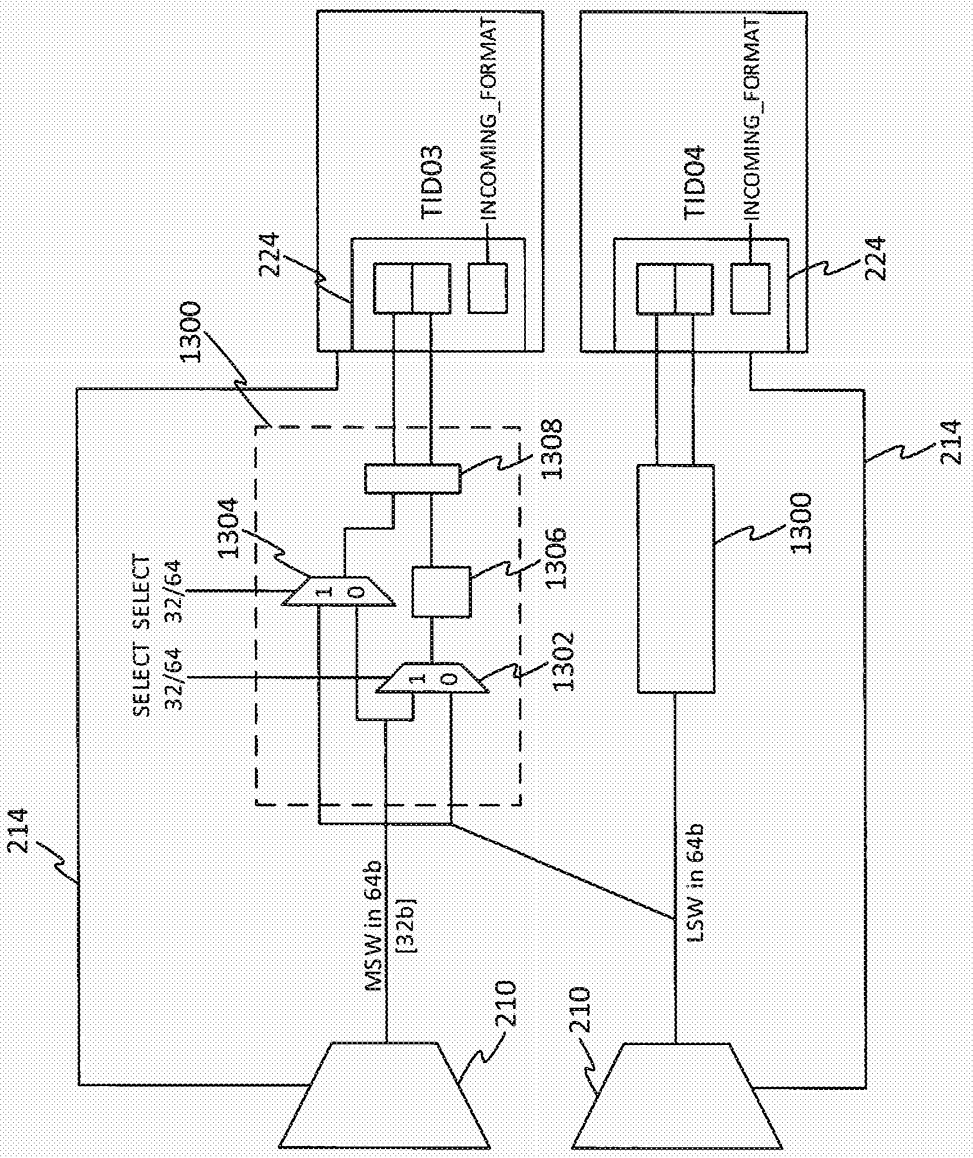

TID03
INCOMING_FORMAT
TID04
INCOMING_FORMAT
224
224
1300
1308
1304
1306
1302
1300
214
214
SELECT 32/64
SELECT 32/64
1
0
1
0
MSW in 64b
[32b]
LSW in 64b
210
210

Figure 13

Synchronization in a Multi-Tile Processing Array

Technical Field

The present disclosure relates to synchronizing the workloads of multiple different tiles in a processor comprising multiple tiles, each tile comprising a processing unit with local memory. Particularly, the disclosure relates to bulk synchronous parallel (BSP) computing protocol, whereby each of a group of tiles must complete a compute phase before any of the tiles in the group can proceed to an exchange phase.

Background

Parallelism in computing takes different forms. Program fragments may be organised to execute concurrently (where they overlap in time but may share execution resources) or in parallel where they execute on different resources possibly at the same time.

Parallelism in computing can be achieved in a number of ways, such as by means of an array of multiple interconnected processor tiles, or a multi-threaded processing unit, or indeed a multi-tile array in which each tile comprises a multi-threaded processing unit.

When parallelism is achieved by means of a processor comprising an array of multiple tiles on the same chip (or chips in the. same integrated circuit package), each tile comprises its own separate respective processing unit with local memory (including program memory and data memory). Thus separate portions of program code can be run concurrently on different tiles. The tiles are connected together via an on-chip interconnect which enables the code run on the different tiles to communicate between tiles. In some cases the processing unit on each tile may take the form of a barrel-threaded processing unit (or other multi-threaded processing unit). Each tile may have a set of contexts and an execution pipeline such that each tile can run multiple interleaved threads concurrently.

In general, there may exist dependencies between the portions of a program running on different tiles in the array. A technique is therefore required to prevent a piece of code on one tile running ahead of data upon which it is dependent being made available by another piece of code on another tile. There are a number of possible schemes for achieving this, but the

scheme of interest herein is known as "bulk synchronous parallel" (BSP). According to BSP, each tile performs a compute phase and an exchange phase in an alternating manner. During the compute phase each tile performs one or more computation tasks locally on tile, but does not communicate any results of its computations with any others of the tiles. In the exchange phase each tile is allowed to exchange one or more results of the computations from the preceding compute phase to and/or from one or more others of the tiles in the group, but does not yet begin a new compute phase until that tile has finished its exchange phase. Further, according to this form of BSP principle, a barrier synchronization is placed at the juncture transitioning from the compute phase into the exchange phase, or transitioning from the exchange phases into the compute phase, or both. That is it say, either: (a) all tiles are required to complete their respective compute phases before any in the group is allowed to proceed to the next exchange phase, or (b) all tiles in the group are required to complete their respective exchange phases before any tile in the group is allowed to proceed to the next compute phase, or (c) both. When used herein the phrase "between a compute phase and an exchange phase" encompasses all these options.

An example use of multi-threaded and/or multi-tiled parallel processing is found in machine intelligence. As will be familiar to those skilled in the art of machine intelligence, machine intelligence algorithms "are capable of producing knowledge models" and using the knowledge model to run learning and inference algorithms. A machine intelligence model incorporating the knowledge model and algorithms can be represented as a graph of multiple interconnected nodes. Each node represents a function of its inputs. Some nodes receive the inputs to the graph and some receive inputs from one or more other nodes. The output activation of some nodes form the inputs of other nodes, and the output of some nodes provide the output of the graph, and the inputs to the graph provide the inputs to some nodes. Further, the function at each node is parameterized by one or more respective parameters, e.g. weights. During a learning stage the aim is, based on a set of experiential input data, to find values for the various parameters such that the graph as a whole will generate a desired output for a range of possible inputs. Various algorithms for doing this are known in the art, such as a back propagation algorithm based on stochastic gradient descent. Over multiple iterations the parameters are gradually tuned to decrease their errors, and thus the graph converges toward a solution. In a subsequent stage, the learned model can then be used to make predictions of outputs given a specified set of inputs or to make inferences as to inputs

(causes) given a specified set of outputs, or other introspective forms of analysis can be performed on it.

The implementation of each node will involve the processing of data, and the interconnections of the graph correspond to data to be exchanged between the nodes. Typically, at least some of the processing of each node can be carried out independently of some or all others of the nodes in the graph, and therefore large graphs expose opportunities for huge parallelism.

Summary

As mentioned above, a machine intelligence model representing the knowledge model and algorithmic information about how the knowledge model is used for learning and inference can generally be represented by a graph of multiple interconnected nodes, each node having a processing requirement on data. Interconnections of the graph indicate data to be exchanged between the nodes and consequently cause dependencies between the program fragments executed at the nodes. Generally, processing at a node can be carried out independently of another node, and therefore large graphs expose huge parallelism. A highly distributed parallel machine is a suitable machine structure for computation of such machine intelligence models. This feature enables a machine to be designed to make certain time deterministic guarantees.

A factor of knowledge models which is exploited in the present disclosure is the generally static nature of the graph. That is to say that the structure of nodes and graph comprising the graph does not usually change during execution of machine intelligence algorithms. The inventors have made a machine which makes certain time deterministic guarantees to optimise computation on machine intelligence models. This allows a compiler to partition and schedule work across the nodes in a time deterministic fashion. It is this time determinism which is utilised in following described embodiments for significant optimisations in designing a computer optimised to process workloads based on knowledge models.

According to an aspect of the invention there is provided a computer comprising a plurality of processing units each having instruction storage holding a local program, an execution unit executing the local program, data storage for holding data; an input interface with a set of

input wires, and an output interface with a set of output wires; a switching fabric connected to each of the processing units by the respective set of output wires and connectable to each of the processing units by the respective input wires via switching circuitry controllable by each processing unit; a synchronisation module operable to generate a synchronisation signal to control the computer to switch between a compute phase and an exchange phase, wherein the processing units are configured to execute their local programs according to a common clock, the local programs being such that in the exchange phase at least one processing unit executes a send instruction from its local program to transmit at a transmit time a data packet onto its output set of connection wires, the data packet being destined for at least one recipient processing unit but having no destination identifier, and at a predetermined switch time the recipient processing unit executes a switch control instruction from its local program to control its switching circuitry to connect its input set of wires to the switching fabric to receive the data packet at a receive time, the transmit time and, switch time and receive time being governed by the common clock with respect to the synchronisation signal.

Another aspect of the invention provides a method of computing a function in a computer comprising: a plurality of processing units each having instruction storage holding a local program, an execution unit for executing the local program, data storage for holding data, an input interface with a set of input wires and an output interface with a set of output wires; a switching fabric connected to each of the processing units by the respective sets of output wires and connectable to each of the processing units by their respective input wires via switching circuitry controllable by each processing unit; and a synchronisation module operable to generate a synchronisation signal to control the computer to switch between a compute phase and an exchange phase, the method comprising; the processing units executing their local programs in the compute phase according to a common clock, wherein at a predetermined time in the exchange phase at least one processing unit executes a send instruction from its local program to transmit at a transmit time a data packet onto its output set of connection wires, the data packet being destined for at least one recipient processing unit but having no destination identifier, and at a predetermined switch time the recipient processing unit executing a switch control instruction from its local program to control the switching circuitry to connect its input set of wires to the switching fabric to receive the data packet at a receive time, the transmit time and switch time and being governed by the common clock with respect to the synchronisation signal.

In principle, the synchronisation signal could be generated to control the switch from a compute phase into the exchange phase, or from the exchange phase into the compute phase. For the time deterministic architecture defined herein, however, it is preferred if the synchronisation signal is generated to commence the exchange phase. In one embodiment, each processing unit indicates to the synchronisation module that its own compute phase is complete, and the synchronisation signal is generated by the synchronisation module when all processing units have indicated that their own compute phase is complete, to commence the exchange phase.

The transmit time should be predetermined to enable the time deterministic exchange to be properly completed. It can be determined by being a known number of clock cycles after the time at which the send instruction is executed, assuming that the time at which the send instruction is executed is predetermined. Alternatively, the transmit time could be a known delay, determined in some other way, from a time known from execution of the send instruction. What is important is that the transmit time is known relative to the receive time on an intended recipient processing unit.

Features of the send instruction can include that the send instruction explicitly defines the send address identifying a location in the data storage from which the data packet is to be sent. Alternatively, no send address is explicitly defined in the send instruction, and the data packets are transmitted from a send address defined in a register implicitly defined by the send instruction. The local program can include a send address update instruction for updating the send address in the implicit register.

In the embodiments described herein, the switching circuitry comprises a multiplexer having an exit set of output wires connected to its processing unit, and multiple sets of input wires connected to the switching fabric, whereby one of the multiple sets of input wires is selected as controlled by the processing unit. Each set can comprise 32 bits. When 64-bit datum are utilised, a pair of multiplexes can be connected to a processing unit and controlled together.

In the described embodiment the recipient processing unit is configured to receive the data packet and load it into the data storage at a memory location identified by a memory pointer. The memory pointer can be automatically incremented after each data packet has been loaded into the data storage. Alternatively, the local program at the recipient processing unit can

include a memory pointer update instruction which updates the memory pointer.

The send instruction may be configured to identify a number of data packets to be sent, wherein, each data packet is associated with a different transmit time, because they are sent serially from the processing unit.

One of the sets of input wires of the multiplexer can be controlled to be connected to a null input. This could be used to ignore datum otherwise arriving at that processing unit.

The recipient processing unit which is intended to receive a particular data packet could be the same processing unit that executed a send instruction at an earlier time, whereby the same processing unit is configured to send a data packet and receive that data packet at a later time. The purpose of a processing unit 'sending to itself' might be to adhere an arrangement in its memory of incoming data interleaved with data received from other processing units.
In some embodiments at least two of the processing units may cooperate in a transmitting pair wherein a first data packet is transmitted from a first processing unit of the pair via its output set of connection wires, and a second data packet is transmitted from the first processing unit of the pair via the output set of connection wires of the second processing unit of the pair to effect a double width transmission. In some embodiments at least two of the processing units may operate as a receiving pair wherein each processing unit of the pair controls its switching circuitry to connect its respective input set of wires to the switching fabric to receive respective data packets from respective tiles of a sending pair.

The multiple processing units may be configured to execute respective send instructions to transmit respective data packets, wherein at least some of the data packets are destined for no recipient processing units.

The function which is being computed may be provided in the form of a static graph comprising a plurality of interconnected nodes, each node being implemented by a codelet of the local programs. A codelet defines a vertex (node) in the graph, and can be considered as an atomic thread of execution, discussed later in the description. In the compute phase, each codelet may process data to produce a result, wherein some of the results are not required for a subsequent compute phase and are not received by any recipient processing unit. They are effectively discarded, but without the need to make any positive discard action. In the

exchange phase the data packets are transmitted between processing units via the switching fabric and the switching circuitry. Note that in the exchange phase some instructions are executed from the local program to implement the exchange phase. These instructions include the send instruction. While the compute phase is responsible for computations, note that it might be possible to include some arithmetic or logical functions during the exchange phase, provided that these functions do not include data dependency on the timing of the local program so that it remains synchronous.

The time deterministic architecture described herein is particularly useful in contexts where the graph represents a machine intelligence function.

The switching fabric can be configured such that in the exchange phase data packets are transmitted through it in a pipeline fashion via a sequence of temporary stores, each store holding a data packet for one cycle of the common clock.

According to another aspect, there is provided a computer implemented method of generating multiple programs to deliver a computerised function, each program to be executed in a processing unit of a computer comprising a plurality of processing units each having instruction storage for holding a local program, an execution unit for executing the local program and data storage for holding data, a switching fabric connected to an output interface of each processing unit and connectable to an input interface of each processing unit by switching circuitry controllable by each processing unit, and a synchronisation module operable to generate a synchronisation signal, the method comprising: generating a local program for each processing unit comprising a sequence of executable instructions; determining for each processing unit a relative time of execution of instructions of each local program whereby a local program allocated to one processing unit is scheduled to execute with a predetermined delay relative to a synchronisation signal a send instruction to transmit at least one data packet at a predetermined transmit time, relative to the synchronisation signal, destined for a recipient processing unit but having no destination identifier, and a local program allocated to the recipient processing unit is scheduled to execute at a predetermined switch time a switch control instruction to control the switching circuitry to connect its processing unit wire to the switching fabric to receive the data packet at a receive time.

In some embodiments, the processing units have a fixed positional relationship with respect to each other, and the step of determining comprises determining a fixed delay based on the positional relationship between each pair of processing units in the computer.

In some embodiments, the fixed positional relationship comprises an array of rows and columns, wherein each processing unit has an identifier which identifies its position in the array.

In some embodiments, the switching circuitry comprises a multiplexer having an output set of wires connected to its processing unit, and multiple set of input wires connectable to the switching fabric, the multiplexor located on the computer at a predetermined physical location with respect to its processing unit, and wherein the step of determining comprises determining the fixed delay for the switch control instruction to reach the multiplexer and an output data packet from the multiplexer to reach the input interface of its processing unit.

In some embodiments, the method comprises the step of providing in each program a synchronisation instruction which indicates to the synchronisation module that a compute phase at the processing unit has completed.

In some embodiments, the step of determining comprises determining for each processing unit a fixed delay between a synchronisation event on the chip and receiving back at the processing unit an acknowledgement that a synchronisation event has occurred.

In some embodiments, the step of determining comprises accessing a look-up table holding information about delays enabling the predetermined send time and predetermined switch time to be determined.

In some embodiments, the computerised function is a machine learning function.

In some embodiments, the switching circuitry comprises a multiplexer having an output set of wires connected to its processing unit, and multiple set of input wires connectable to the switching fabric, the multiplexor located on the computer at a predetermined physical location with respect to its processing unit, and wherein the step of determining comprises determining the fixed delay for the switch control instruction to reach the multiplexer and an output data packet from the multiplexer to reach the input interface of its processing unit.

In some embodiments, the step of providing in each program a synchronisation instruction which indicates to the synchronisation module that a compute phase at the processing unit has completed.

In some embodiments, the step of determining comprises determining for each processing unit a fixed delay between a synchronisation event on the chip and receiving back at the processing unit an acknowledgement that a synchronisation event has occurred.

In some embodiments, the step of determining comprises accessing a look-up table holding information about delays enabling the predetermined send time and predetermined switch time to be determined.

In some embodiments, the computerised function is a machine learning function.

According to another aspect, there is provided a compiler having a processor programmed to carry out a method of generating multiple programs to deliver a computerised function, each program to be executed in a processing unit of a computer comprising a plurality of processing units each having instruction storage for holding a local program, an execution unit for executing the local program and data storage for holding data, a switching fabric connected to an output interface of each processing unit and connectable to an input interface of each processing unit by switching circuitry controllable by each processing unit, and a synchronisation module operable to generate a synchronisation signal, the method comprising: generating a local program for each processing unit comprising a sequence of executable instructions; determining for each processing unit a relative time of execution of instructions of each local program whereby a local program allocated to one processing unit is scheduled to execute with a predetermined delay relative to a synchronisation signal a send instruction to transmit at least one data packet at a predetermined transmit time, relative to the synchronisation signal, destined for a recipient processing unit but having no destination identifier, and a local program allocated to the recipient processing unit is scheduled to execute at a predetermined switch time a switch control instruction to control the switching circuitry to connect its processing unit wire to the switching fabric to receive the data packet at a receive time; the compiler being connected to receive a fixed graph structure representing the computerised function and a table holding delays enabling the predetermined send time and predetermined switch time to be determined for each processing unit.

In some embodiments, the computerised function is a machine learning function.

In some embodiments, the fixed graph structure comprises a plurality of nodes, each node being represented by a codelet in a local program.

In some embodiments, the fixed graph structure comprises a plurality of nodes, each node being represented by a codelet in a local program.

According to another aspect, there is provided a computer program recorded on non transmissible media and comprising computer readable instructions which when executed by a processor of a compiler implement a method of generating multiple programs to deliver a computerised function, each program to be executed in a processing unit of a computer comprising a plurality of processing units each having instruction storage for holding a local program, an execution unit for executing the local program and data storage for holding data, a switching fabric connected to an output interface of each processing unit and connectable to an input interface of each processing unit by switching circuitry controllable by each processing unit, and a synchronisation module operable to generate a synchronisation signal, the method comprising: generating a local program for each processing unit comprising a sequence of executable instructions; determining for each processing unit a relative time of execution of instructions of each local program whereby a local program allocated to one processing unit is scheduled to execute with a predetermined delay relative to a synchronisation signal a send instruction to transmit at least one data packet at a predetermined transmit time, relative to the synchronisation signal, destined for a recipient processing unit but having no destination identifier, and a local program allocated to the recipient processing unit is scheduled to execute at a predetermined switch time a switch control instruction to control the switching circuitry to connect its processing unit wire to the switching fabric to receive the data packet at a receive time.

According to another aspect, there is provided a computer program comprising a sequence of instructions for execution on a processing unit having instruction storage for holding the computer program, an execution unit for executing the computer program and data storage for holding data, the computer program comprising one or more computer executable instruction which, when executed, implements: a send function which causes a data packet destined for a recipient processing unit to be transmitted on a set of connection wires connected to the processing unit, the data packet having no destination identifier but being transmitted at a predetermined transmit time; and a switch control function which causes the

processing unit to control switching circuitry to connect a set of connection wires of the processing unit to a switching fabric to receive a data packet at a predetermined receive time.

In some embodiments, the one or more instruction comprises a switch control instruction and a send instruction which defines a send address defining a location in the instruction storage from which the data packet is to be sent.

In some embodiments, the send instruction defines a number of data packets to be sent, each packet being associated with a different predetermined transmit time.

In some embodiments, the send instruction does not explicitly define a send address but implicitly defines a register in which a send address is held.

In some embodiments, the computer program comprises a further instruction for updating the send address in the implicitly defined register.

In some embodiments, the computer program comprises at least one further instruction defines a memory pointer update function which updates a memory pointer identifying a memory location in the data storage for storing the data packet which is received at the recipient processing unit.

In some embodiments, the one or more instruction is a merged instruction which merges the send function and the switch control function in a single execution cycle, whereby the processing unit is configured to operate to transmit a data packet and to control its switching circuitry to receive a different data packet from another processing unit.

In some embodiments, the at least one further instruction is a merged instruction which merges the send function and the memory pointer update function.

In some embodiments, the merged instruction is configured in a common format with an opcode portion which designates whether it merges the send function with the memory pointer update function or the switch control function.

In some embodiments, the one or more instruction is a single instruction which merges the send function, switch control function and memory pointer update function in a single execution cycle.

In some embodiments, each one or more instruction has a first bit width which matches a bit width of a fetch stage of the execution unit.

In some embodiments, each one or more instruction has a first bit width which matches a bit width of a fetch stage of the execution unit, and wherein: the instruction which merges the send function, switch control function and memory pointer update function has a second bit width which is twice the bit width of the fetch stage of the execution unit.

In some embodiments, each one or more instruction has a first bit width which matches a bit width of a fetch stage of the execution unit, and wherein: the instruction of a first bit width identifies an operand of the first bit width, the operand implementing the switch control function and memory write update function.

In some embodiments, the computer program comprises a synchronisation instruction which generates an indication when a compute phase of the processing unit has been completed.

In some embodiments, the computer program is recorded on a non-transmissible computer readable media.

In some embodiments, the computer program is in the form of a transmissible signal.

According to another aspect, there is provided a processing unit comprising instruction storage, an execution unit configured to execute a computer program and data storage for holding data, wherein the instruction storage holds a computer program comprising one or more computer executable instruction which, when executed by the execution unit, implements: a send function which causes a data packet destined for a recipient processing unit to be transmitted on a set of connection wires connected to the processing unit, the data packet having no destination identifier but being transmitted at a predetermined transmit time; and a switch control function which causes the processing unit to control switching

circuitry to connect a set of connection wires of the processing unit to a switching fabric to receive a data packet at a predetermined receive time.

According to another aspect, there is provided a computer comprising one or more die in an integrated package, the computer comprising a plurality of processing units, each processing unit having instruction storage for holding a computer program, an execution unit configured to execute the computer program and data storage for holding data, wherein the instruction storage for each processing unit holds a computer program comprising one or more computer executable instruction which, when executed, implements: a send function which causes a data packet destined for a recipient processing unit to be transmitted on a set of connection wires connected to the processing unit, the data packet having no destination identifier but being transmitted at a predetermined transmit time; and a switch control function which causes the processing unit to control switching circuitry to connect a set of connection wires of the processing unit to a switching fabric to receive a data packet at a predetermined receive time.

Brief description of the drawings

For a better understanding of the present invention and to show how the same may be carried into effect reference will now be made by way of example to the following drawings.

Figure 1 illustrates schematically the architecture of a single chip processor;
Figure 2 is a schematic diagram of a tile connected to the switching fabric;
Figure 3 is a diagram illustrating a BSP protocol;
Figure 4 is a schematic diagram showing two tiles in a time deterministic exchange;
Figure 5 is a schematic timing diagram illustrating a time deterministic exchange;
Figure 6 is one example of a machine intelligence graph;
Figure 7 is a schematic architecture illustrating operation of a compiler for generating time deterministic programs;
Figures 8 to 11 illustrate instruction formats of different instructions usable in a time deterministic architecture.
Figure 12 is a schematic diagram of two tiles operating as a transmitting pair; and
Figure 13 is a schematic diagram of two tiles operating as a receiving pair.

Detailed description of the embodiments

Figure 1 illustrates schematically the architecture of a single chip processor 2. The processor is referred to herein as an IPU (Intelligence Processing Unit) to denote its adaptivity to machine intelligence applications. In a computer, the single chip processors can be connected together as discussed later, using links on the chip, to form a computer. The present description focuses on the architecture of the single chip processor 2. The processor 2 comprises multiple processing units referred to as tiles. In one embodiment, there are 1216 tiles organised in arrays 6a, 6b which are referred to herein as "North" and "South". In the described example, each array has eight columns of 76 tiles (in fact generally there will be 80 tiles, for redundancy purposes). It will be appreciated that the concepts described herein extend to a number of different physical architectures – one example is given here to aid understanding. The chip 2 has two chip to host links 8a, 8b and 4 chip to chip links 30a, 30b arranged on the "West" edge of the chip 2. The chip 2 receives work from a host (not shown) which is connected to the chip via one of the card-to-host links in the form of input data to be processed by the chip 2. The chips can be connected together into cards by a further 6 chip-to-chip links 30a, 30b arranged along the "East" side of the chip. A host may access a computer which is architected as a single chip processor 2 as described herein or a group of multiple interconnected single chip processors 2 depending on the workload from the host application.

The chip 2 has a clock 3 which controls the timing of chip activity. The clock is connected to all of the chip's circuits and components. The chip 2 comprises a time deterministic switching fabric 34 to which all tiles and links are connected by sets of connection wires, the switching fabric being stateless, i.e. having no program visible state. Each set of connection wires is fixed end to end. The wires are pipelined. In this embodiment, a set comprises 32 data wires plus control wires, e.g. a valid bit. Each set can carry a 32-bit data packet, but note herein that the word "packet" denotes a set of bits representing a datum (sometimes referred to herein as a data item), perhaps with one or more valid bit. The "packets" do not have headers or any form of destination identifier which permits an intended recipient to be uniquely identified, nor do they have end-of-packet information. Instead, they each represent a numerical or logical value input to or output from a tile. Each tile has its own local memory (described later). The tiles do not share memory. The switching fabric constitutes a cross set of connection wires only connected to multiplexers and tiles as described later and does not hold any program visible state. The switching fabric is considered to be stateless and does not use any memory. Data exchange between tiles is conducted on a time deterministic basis as

described herein. A pipelined connection wire comprises a series of temporary stores, e.g. latches or flip flops which hold datum for a clock cycle before releasing it to the next store. Time of travel along the wire is determined by these temporary stores, each one using up a clock cycle of time in a path between any two points.

Figure 2 illustrates an example tile 4 in accordance with embodiments of the present disclosure. In the tile, multiple threads are interleaved through a single execution pipeline. The tile 4 comprises: a plurality of contexts 26 each arranged to represent the state of a different respective one of a plurality of threads; a shared instruction memory 12 common to the plurality of threads; a shared data memory 22 that is also common to the plurality of threads; a shared execution pipeline 14, 16, 18 that is again common to the plurality of threads; and a thread scheduler 24 for scheduling the plurality of threads for execution through the shared pipeline in an interleaved manner. The thread scheduler 24 is schematically represented in the diagram by sequence of time slots $S_0 \ldots S_5$, but in practice is a hardware mechanism managing program counters of the threads in relation to their time slots. The execution pipeline comprises a fetch stage 14, a decode stage 16, and an execution stage 18 comprising an execution unit (EXU) and a load/store unit (LSU). Each of the contexts 26 comprises a respective set of registers R_0, R_1... for representing the program state of the respective thread.

The fetch stage 14 is connected to fetch instructions to be executed from the instruction memory 12, under control of the thread scheduler 24. The thread scheduler 24 is configured to control the fetch stage 14 to fetch instructions from the local program for execution in each time slot as will be discussed in more detail below.

The fetch stage 14 has access to a program counter (PC) of each of the threads that is currently allocated to a time slot. For a given thread, the fetch stage 14 fetches the next instruction of that thread from the next address in the instruction memory 12 as indicated by the thread's program counter. Note that an instruction as referred to herein, means a machine code instruction, i.e. an instance of one of the fundamental instructions of the computer's instruction set, made up of an opcode and zero or more operands. Note too that the program loaded into each tile is determined by a processor or compiler to allocate work based on the graph of the machine intelligence model being supported.

The fetch stage 14 then passes the fetched instruction to the decode stage 16 to be decoded, and the decode stage 16 then passes an indication of the decoded instruction to the execution stage 18 along with the decoded addresses of any operand registers of the current context specified in the instruction, in order for the instruction to be executed.

In the present example, the thread scheduler 24 interleaves threads according to a round-robin scheme whereby, within each round of the scheme, the round is divided into a sequence of time slots S_0, S_1, S_2, S_3, each for executing a respective thread. Typically each slot is one processor cycle long and the different slots are evenly sized (though not necessarily so in all possible embodiments). This pattern then repeats, each round comprising a respective instance of each of the time slots (in embodiments in the same order each time, though again not necessarily so in all possible embodiments). Note therefore that a time slot as referred to herein means the repeating allocated place in the sequence, not a particular instance of the time slot in a given repetition of the sequence. In the illustrated embodiment, there are eight time slots, but other numbers are possible. Each time slot is associated with hardware resource, e.g. register, for managing the context of an executing thread.

One of the contexts 26, labelled SV, is reserved for a special function, to represent the state of a "supervisor" (SV) whose job it is to coordinate the execution of "worker" threads., The supervisor can be implemented as a program organised as one or more supervisor threads which may run concurrently. The supervisor thread may also be responsible for performing barrier synchronisations described later or may be responsible for exchanging data on and off the tile, as well as in and out of local memory so that is can be shared between the worker threads between computations. The thread scheduler 24 is configured so as, when the program as a whole starts, to begin by allocating the supervisor thread to all of the time slots, i.e. so the supervisor SV starts out running in all time slots S_0...S_5. However, the supervisor thread is provided with a mechanism for, at some subsequent point (either straight away or after performing one or more supervisor tasks), temporarily relinquishing each of the slots in which it is running to a respective one of the worker threads C_0, C_1 denote slots to which a worker thread has been allocated. This is achieved by the supervisor thread executing a relinquish instruction, called "RUN" by way of example herein. In embodiments this instruction takes two operands: an address of a worker thread in the instruction memory 12 and an address of some data for that thread in the data memory 22:

RUN task_addr, data_addr

Each worker thread is a codelet intended to represent a vertex in the graph and to execute atomically. That is all the data it consumers is available at launch and all the data it produces is not visible to other threads until it exits. It runs to completion (excepting error conditions).The data address may specify some data to be acted upon by the codelet. Alternatively, the relinquish instruction may take only a single operand specifying the address of the codelet, and the data address could be included in the code of the codelet; or the single operand could point to a data structure specifying the addresses of the codelet and data. Codelets may be run concurrently and independently of one another.

Either way, this relinquish instruction ("RUN") acts on the thread scheduler 24 so as to relinquish the current time slot, i.e. the time slot in which this instruction is executed, to the worker thread specified by the operand. Note that it is implicit in the relinquish instruction that it is the time slot in which this instruction is executed that is being relinquished (implicit in the context of machine code instructions means it doesn't need an operand to specify this – it is understood implicitly from the opcode itself). Thus the slot which is given away is the slot which the supervisor executes the relinquish instruction in. Or put another way, the supervisor is executing in the same space that it gives away. The supervisor says "run this codelet at this time slot", and then from that point onwards the slot is owned (temporarily) by the relevant worker thread. Note that when a supervisor uses a slot it does not use the context associated with that slot but uses its own context SV.

The supervisor thread SV performs a similar operation in each of the time slots, to give away all its slots C_0, C_1 to different respective ones of the worker threads. Once it has done so for the last slot, the supervisor pauses execution, because it has no slots in which to execute. Note that the supervisor may not give away all its slots, it may retain some for running itself.

When the supervisor thread determines it is time to run a codelet, it uses the relinquish instruction ("RUN") to allocate this codelet to the slot in which it executes the 'RUN' instruction.

Each of the worker threads in slots C_0, C_1 proceeds to perform its one or more computation tasks. At the end of its task(s), the worker thread then hands the time slot in which it is running back to the supervisor thread.

This is achieved by the worker thread executing an exit instruction ("EXIT"). In one embodiment, the EXIT instruction takes at least one operand and preferably only a single

operand, exit state (e.g. a binary value), to be used for any purpose desired by the programmer to indicate a state of the respective codelet upon ending.

EXIT exit_state

In one embodiment, the EXIT instruction acts on the scheduler 24 so that the time slot in which it is executed is returned back to the supervisor thread. The supervisor thread can then perform one or more subsequent supervisor tasks (e.g. barrier synchronization and/or movement of data in memory to facilitate the exchange of data between worker threads), and/or continue to execute another relinquish instruction to allocate a new worker thread (W4, etc.) to the slot in question. Note again therefore that the total number of threads in the instruction memory 12 may be greater than the number that barrel-threaded processing unit 10 can interleave at any one time. It is the role of the supervisor thread SV to schedule which of the worker threads W0...Wj from the instruction memory 12, at which stage in the overall program, are to be executed.

In another embodiment, the EXIT instruction does not need to define an exit state.

This instruction acts on the thread scheduler 24 so that the time slot in which it is executed is returned back to the supervisor thread. The supervisor thread can then perform one or more supervisor subsequent tasks (e.g. barrier synchronization and/or exchange of data), and/or continue to execute another relinquish instruction, and so forth.

As briefly mentioned above, data is exchanged between tiles in the chip. Each chip operates a Bulk Synchronous Parallel protocol, comprising a compute phase and an exchange phase. The protocol is illustrated for example in Figure 3. The left-hand diagram in Figure 3 represents a compute phase in which each tile 4 is in a phase where the stateful codelets execute on local memory (12, 22). Although in Figure 3 the tiles 4 are shown arranged in a circle this is for explanatory purposes only and does not reflect the actual architecture.

After the compute phase, there is a synchronisation denoted by arrow 30. To achieve this, a SYNC (synchronization) instruction is provided in the processor's instruction set. The SYNC instruction has the effect of causing the supervisor thread SV to wait until all currently

executing workers W have exited by means of an EXIT instruction. In embodiments the SYNC instruction takes a mode as an operand (in embodiments its only operand), the mode specifying whether the SYNC is to act only locally in relation to only those worker threads running locally on the same processor module 4, e.g. same tile, or whether instead it is to apply across multiple tiles or even across multiple chips.

```
SYNC mode          // mode ∈ {tile, chip, zone_1, zone_2}
```

BSP in itself is known in the art. According to BSP, each tile 4 performs a compute phase 52 and an exchange (sometimes called communication or message-passing) phase 50 in an alternating cycle. The compute phase and exchange phase are performed by the tile executing instructions. During the compute phase 52 each tile 4 performs one or more computation tasks locally on-tile, but does not communicate any results of these computations with any others of the tiles 4. In the exchange phase 50 each tile 4 is allowed to exchange (communicate) one or more results of the computations from the preceding compute phase to and/or from one or more others of the tiles in the group, but does not yet perform any new computations that have a potential dependency on a task performed on another tile 4 or upon which a task on another tile 4 might potentially have a dependency (it is not excluded that other operations such as internal control-related operations may be performed in the exchange phase). Further, according to the BSP principle, a barrier synchronization is placed at the juncture transitioning from the compute phases 52 into the exchange phase 50, or the juncture transitioning from the exchange phases 50 into the compute phase 52, or both. That is it say, either: (a) all tiles 4 are required to complete their respective compute phases 52 before any in the group is allowed to proceed to the next exchange phase 50, or (b) all tiles 4 in the group are required to complete their respective exchange phases 50 before any tile in the group is allowed to proceed to the next compute phase 52, or (c) both of these conditions is enforced. This sequence of exchange and compute phases may then repeat over multiple repetitions. In BSP terminology, each repetition of exchange phase and compute phase is referred to herein as a "superstep", consistent with usage in some prior descriptions of BSP. It is noted herein that the term "superstep" is sometimes used in the art to denote each of the exchange phase and compute phase.

The execution unit (EXU) of the execution stage 18 is configured so as, in response to the opcode of the SYNC instruction, when qualified by the on-chip (inter-tile) operand, to cause the supervisor thread in which the "SYNC chip" was executed to be paused until all the tiles 4 in the array 6 have finished running workers. This can be used to implement a barrier to the next BSP superstep, i.e. after all tiles 4 on the chip 2 have passed the barrier, the cross-tile program as a whole can progress to the next exchange phase 50.

Each tile indicates its synchronisation state to a sync module 36. Once it has been established that each tile is ready to send data, the synchronisation process 30 causes the system to enter an exchange phase which is shown on the right-hand side of Figure 3. In this exchange phase, data values move between tiles (in fact between the memories of tiles in a memory-to-memory data movement). In the exchange phase, there are no computations which might induce concurrency hazards between tile programs. In the exchange phase, each datum moves along the connection wires on which it exits a tile from a transmitting tile to one or multiple recipient tile(s). At each clock cycle, datum moves a certain distance along its path (store to store), in a pipelined fashion. When a datum is issued from a tile, it is not issued with a header identifying a recipient tile. Instead, the recipient tile knows that it will be expecting a datum from a certain transmitting tile at a certain time. Thus, the computer described herein is time deterministic. Each tile operates a program which has been allocated to it by the programmer or by a compiler exercise, where the programmer or the compiler function has knowledge of what will be transmitted by a particular tile at a certain time and what needs to be received by a recipient tile at a certain time. In order to achieve this, SEND instructions are included in the local programs executed by the processor on each tile, where the time of execution of the SEND instruction is predetermined relative to the timing of other instructions being executed on other tiles in the computer. This is described in more detail later, but firstly the mechanism by which a recipient tile can receive a datum at a predetermined time will be described. Each tile 4 is associated with its own multiplexer 210: thus, the chip has 1216 multiplexer. Each multiplexer has 1216 inputs, each input being 32-bits wide (plus optionally some control bits).Each input is connected to a respective set of connecting wires $140_{x\,in}$ the switching fabric 34. The connecting wires of the switching fabric are also connected to a data out set of connection wires 218 from each tile (a broadcast exchange bus, described later), thus there are 1216 sets of connecting wires which in this embodiment extend in a direction across the chip. For ease of illustration, a single emboldened set of wires 140_{sc} is shown connected to the data out wires 218_s, coming from a

tile not shown in Figure 2, in the south array 6b. This set of wires is labelled 140_x to indicate that it is one of a number of sets of crosswires 140_0- 140_{1215}. As can now be seen from Figure 2, it will be appreciated that when the multiplexer 210 is switched to the input labelled 220_x then that will connect to the crosswires 140_x and thus to the data out wires 218_s of the tile (not shown in Figure 2) from the south array 6b. If the multiplexer is controlled to switch to that input (220_{sc}) at a certain time, then the datum received on the data out wires which is connected to the set of connecting wire 140_x will appear at the output 230 of the multiplexer 210 at a certain time. It will arrive at the tile 4 a certain delay after that, the delay depending on the distance of the multiplexer from the tile. As the multiplexers form part of switching fabric, the delay from the tile to the multiplexer can vary depending on the location of the tile. To implement the switching, the local programs executed on the tiles include switch control instructions (PUTi) which cause a multiplexer control signal 214 to be issued to control the multiplexer associated with that tile to switch its input at a certain time ahead of the time at which a particular datum is expected to be received at the tile. In the exchange phase, multiplexers are switched and packets (data) are exchanged between tiles using the switching fabric. It is clear from this explanation that the switching fabric has no state - the movement of each datum is predetermined by the particular set of wires to which the input of each multiplexer is switched.

In the exchange phase, an all tiles to all tiles communication is enabled. The exchange phase can have multiple cycles. Each tile 4 has control of its own unique input multiplexer 210. Incoming traffic from any other tile in the chip, or from one of the connection links can be selected. Note that it is possible for a multiplexer to be set to receive a 'null' input – that is, no input from any other tile in that particular exchange phase. Selection can change cycle-by-cycle within an exchange phase; it does not have to be constant throughout. Data may be exchanged on chip, or from chip to chip or from chip to host depending on the link which is selected. The present application is concerned mainly with inter-tile communication on a chip. To perform synchronisation on the chip, a small number of pipelined signals are are provided from all of the tiles to a sync controller 36 on the chip and a pipelined sync-ack signal is broadcast from the sync controller back to all tiles. In one embodiment the pipelined signals are one-bit-wide daisy chained AND/OR signals. One mechanism by which synchronisation between tiles is achieved is the SYNC instruction mentioned above, or described in the following. Other mechanism may be utilised: what is important is that all tiles can be synchronised between a compute phase of the chip and an exchange phase of the

chip (Figure 3). The SYNC instruction triggers the following functionality to be triggered in dedicated synchronization logic on the tile 4, and in the synchronization controller 36. The synch controller 36 may be implemented in the hardware interconnect 34 or, as shown, in a separate on chip module. This functionality of both the on-tile sync logic and the synchronization controller 36 is implemented in dedicated hardware circuitry such that, once the SYNC chip is executed, the rest of the functionality proceeds without further instructions being executed to do so.

Firstly, the on-tile sync logic causes the instruction issue for the supervisor on the tile 4 in question to automatically pause (causes the fetch stage 14 and scheduler 24 to suspend issuing instructions of the supervisor). Once all the outstanding worker threads on the local tile 4 have performed an EXIT, then the sync logic automatically sends a synchronization request "sync_req" to the synchronization controller 36. The local tile 4 then continues to wait with the supervisor instruction issue paused. A similar process is also implemented on each of the other tiles 4 in the array 6 (each comprising its own instance of the sync logic). Thus at some point, once all the final workers in the current compute phase 52 have EXITed on all the tiles 4 in the array 6, the synchronization controller 36 will have received a respective synchronization request (sync_req) from all the tiles 4 in the array 6. Only then, in response to receiving the sync_req from every tile 4 in the array 6 on the same chip 2, the synchronization controller 36 sends a synchronization acknowledgement signal "sync_ack" back to the sync logic on each of the tiles 4. Up until this point, each of the tiles 4 has had its supervisor instruction issue paused waiting for the synchronization acknowledgment signal (sync_ack). Upon receiving the sync_ack signal, the sync logic in the tile 4 automatically unpauses the supervisor instruction issue for the respective supervisor thread on that tile 4. The supervisor is then free to proceed with exchanging data with other tiles 4 in via the interconnect 34 in a subsequent exchange phase 50.

Preferably the sycn_req and sync_ack signals are transmitted and received to and from the synchronization controller, respectively, via one or more dedicated sync wires connecting each tile 4 to the synchronization controller 36 in the interconnect 34.

The connection structure of the tile will now be described in more detail.

Each tile has three interfaces:

an exin interface 224 which passes data from the switching fabric 34 to the tile 4;

an exout interface 226 which passes data from the tile to the switching fabric over the broadcast exchange bus 218; and

an exmux interface 228 which passes the control mux signal 214 (mux-select) from the tile 4 to its multiplexer 210.

In order to ensure each individual tile executes SEND instructions and switch control instructions at appropriate times to transmit and receive the correct data, exchange scheduling requirements need to be met by the programmer or compiler that allocates individual programs to the individual tiles in the computer. This function is carried out by an exchange scheduler which needs to be aware of the following exchange timing (BNET) parameters. In order to understand the parameters, a simplified version of Figure 2 is shown in Figure 4. Figure 4 also shows a recipient tile as well as a transmitting tile.

I. The relative SYNC acknowledgement delay of each tile, BNET_RSAK (TID). TID is the tile identifier held in a TILE_ID register described later. This is a number of cycles always greater than or equal to 0 indicating when each tile receives the ack signal from the sync controller 36 relative to the earliest receiving tile. This can be calculated from the tile ID, noting that the tile ID indicates the particular location on the chip of that tile, and therefore reflects the physical distances. Figure 4 shows one transmitting tile 4_T, and one recipient tile 4_R. Although shown only schematically and not to scale, the tile 4_T is indicated closer to the sync controller and the tile 4_R is indicated being further away, with the consequence that the sync acknowledgement delay will be shorter to the tile 4_T than for the tile 4_R. A particular value will be associated with each tile for the sync acknowledgement delay. These values can be held for example in a delay table, or can be calculated on the fly each time based on the tile ID.

II. The exchange mux control loop delay, BNET_MXP (TID of receiving tile). This is the number of cycles between issuing an instruction (PUTi-MUXptr) that changes a tile's input mux selection and the earliest point at which the same tile could issue a (hypothetical) load instruction for exchange data stored in memory as a result of the new mux selection. Looking at Figure 4, this delay comprises the delay of the control signal getting from the exmux interface 228_R of recipients tile 4_R to its multiplexer 210_R and the length of the line from the output of the multiplexer to the data input exin interface 224.

III. The tile to tile exchange delay, BNET_TT (TID of sending tile, TID of receiving tile). This is the number of cycles between a SEND instruction being issued on one tile and the earliest point at which the receiving tile could issue a (hypothetical) load instruction pointing to the sent value in its own memory. This has been determined from the tile IDs of the sending and receiving tiles, either by accessing a table such as has already been discussed, or by calculation.. Looking again at Figure 4, this delay comprises the time taken for data to travel from transmit tile 4_T from its ex_out interface 226_T to the switching fabric 14 along its exchange bus 218_T and then via the input mux 210_R at the receiving tile 4_R to the ex_in interface 224_R of the receiving tile.

IV. The exchange traffic memory pointer update delay, BNET_MMP(). This is the number of cycles between issuing an instruction (PUTi-MEMptr) that changes a tile's exchange input traffic memory pointer and the earliest point at which that same tile could issue a (hypothetical) load instruction for exchange data stored in memory as a result of the new pointer. This is a small, fixed number of cycles. The memory pointer has not yet been discussed, but is shown in Figure 2 referenced 232. It acts as a pointer into the data memory 202 and indicates where incoming data from the ex_in interface 224 is to be stored. This is described in more detail later.

Figure 5 shows the exchange timings in more depth. On the left-hand side of Figure 4 is the IPU clock cycles running from 0-30. Action on the sending tile 4_T occurs between IPU clock cycles 0 and 9, starting with issuance of a send instruction (SEND F_3). In IPU clock cycles 10 through 24, the datum pipelines its way through the switching fabric 34.

Looking at the receiving tile 4_R in IPU clock cycle 11 a PUTi instruction is executed that changes the tile input mux selection: PUTi-MXptr (F_3). In Figure 5, this PUTi instruction is labelled as "PUTi INCOMING MUX (F3)".

In cycle 18, the memory pointer instruction is executed, PUTi-MEMptr (F_3), allowing for a load instruction in ITU clock cycle 25. In Figure 5, this PUTi instruction is labelled as "PUTi INCOMING ADR (F3)".

On the sending tile 4_t, IPU clock cycles 1, 3 and 5 are marked "Transport ( )". This is an internal tile delay between the issuance of a SEND instruction and the manifestation of the data of the SEND instruction on the exout interface F4, E1, E3 etc. denote datum from earlier SEND instructions in transport to the exout interface. IPU clock cycle 2 is allocated to forming an address EO for a SEND instruction. Note this is where EO is to be fetched from, not its destination address. In IPU clock cycle 4 a memory macro is executed to fetch E2 from memory. In IPU clock cycle 6 a parity check is performed on E4. In IPU clock cycle 7 a MUX output instruction is executed to send E5. In IPU clock cycle 8 E6 is encoded and in IPU clock cycle E7 is output.

In the exchange fabric 34, IPU clock cycles 10 through 24 are labelled "exchange pipe stage". In each cycle, a datum moves "one step" along the pipeline (between temporary stores).

Cycles 25 - 28 denote the delay on the recipient tile 4_R between receiving a datum at the exin interface (see Mem Macro (E2) for Exc), while cycles 25 – 29 denote the delay between receiving a datum at the exin interface and loading it into memory (see Mem Macro (E2)) for LD. Other functions can be carried out in that delay – see Earliest LD (F3), Reg file rd (F4), form adds (EO), Transport (E1).

In simple terms, if the processor of the receiving tile 4_R wants to act on a datum (e.g. F3) which was the output of a process on the transmitting tile 4_T, then the transmitting tile 4_T has to execute a SEND instruction [SEND (F3) at a certain time (e.g. IPU clock cycle 0 in Figure 5), and the receiving tile has to execute a switch control instruction PUTi EXCH MXptr (as in IPU clock cycle 11) by a certain time relative to the execution of the SEND instruction [SEND (F3)] on the transmitting tile. This will ensure that the data arrives at the recipient tile in time to be loaded [earliest LD (F3)] in IPU cycle 25 for use in a codelet being executed at the recipient tile.

Note that the receive process at a recipient tile does not need to involve setting the memory pointer as with instruction PUTi MEMptr. Instead, the memory pointer 232 (Figure 2) automatically increments after each datum is received at the exin interface 224. Received

data is then just loaded into the next available memory location. However, the ability to change the memory pointer enables the recipient tile to alter the memory location at which the datum is written. All of this can be determined by the compiler or programmer who writes the individual programs to the individual tiles such that they properly communicate. This results in the timing of an internal exchange (the inter exchange on chip) to be completely time deterministic. This time determinism can be used by the exchange scheduler to highly optimise exchange sequences.

Figure 6 illustrates an example application of the processor architecture disclosed herein, namely an application to machine intelligence.

As mentioned previously and as will be familiar to a person skilled in the art of machine intelligence, machine intelligence begins with a learning stage where the machine intelligence algorithm learns a knowledge model. The model may be represented as a graph 60 of interconnected nodes 102 and links104. Nodes and links may be referred to as vertices and edges. Each node 102 in the graph has one or more input edges and one or more output edges, wherein some of the input edges of some of the nodes 102 are the output edges of some others of the nodes, thereby connecting together the nodes to form the graph. Further, one or more of the input edges of one or more of the nodes 102 form the inputs to the graph as a whole, and one or more of the output edges of one or more of the nodes 102 form the outputs of the graph as a whole. Each edge 104 communicates a value commonly in the form of a tensor (n-dimensional matrix), these forming the inputs and outputs provided to and from the nodes 102 on their input and output edges respectively.

Each node 102 represents a function of its one or more inputs as received on its input edge or edges, with the result of this function being the output(s) provided on the output edge or edges. These results are sometimes referred to as activations. Each function is parameterized by one or more respective parameters (sometimes referred to as weights, though they need not necessarily be multiplicative weights). In general the functions represented by the different nodes 102 may be different forms of function and/or may be parameterized by different parameters.

Further, each of the one or more parameters of each node's function is characterized by a respective error value. Moreover, a respective error condition may be associated with the error(s) in the parameter(s) of each node 102. For a node 102 representing a function

parameterized by a single error parameter, the error condition may be a simple threshold, i.e. the error condition is satisfied if the error is within the specified threshold but not satisfied if the error is beyond the threshold. For a node 102 parameterized by more than one respective parameter, the error condition for that node 102 may be more complex. For example, the error condition may be satisfied only if each of the parameters of that node 102 falls within respective threshold. As another example, a combined metric may be defined combining the errors in the different parameters for the same node 102, and the error condition may be satisfied on condition that the value of the combined metric falls within a specified threshold, but otherwise the error condition is not satisfied if the value of the combined metric is beyond the threshold (or vice versa depending on the definition of the metric). Whatever the error condition, this gives a measure of whether the error in the parameter(s) of the node falls below a certain level or degree of acceptability.

In the learning stage the algorithm receives experience data, i.e. multiple data points representing different possible combinations of inputs to the graph. As more and more experience data is received, the algorithm gradually tunes the parameters of the various nodes 102 in the graph based on the experience data so as to try to minimize the errors in the parameters. The goal is to find values of the parameters such that, the output of the graph is as as close as possible to a desired result. As the graph as a whole tends toward such a state, the calculation is said to converge.

For instance, in a supervised approach, the input experience data takes the form of training data, i.e. inputs which correspond to known outputs. With each data point, the algorithm can tune the parameters such that the output more closely matches the known output for the given input. In the subsequent prediction stage, the graph can then be used to map an input query to an approximate predicted output (or vice versa if making an inference). Other approaches are also possible. For instance, in an unsupervised approach, there is no concept of a reference result per input datum, and instead the machine intelligence algorithm is left to identify its own structure in the output data. Or in a reinforcement approach, the algorithm tries out at least one possible output for each data point in the input experience data, and is told whether this output is positive or negative (and potentially a degree to which it is positive or negative), e.g. win or lose, or reward or punishment, or such like. Over many trials the algorithm can gradually tune the parameters of the graph to be able to predict inputs that will result in a positive outcome. The various approaches and algorithms for learning a graph will be known to a person skilled in the art of machine learning.

According to an exemplary application of the techniques disclosed herein, each worker thread is programmed to perform the computations associated with a respective individual one of the nodes 102 in a machine intelligence graph. In this case the edges 104 between nodes 102 correspond to the exchanges of data between threads, at least some of which may involve exchanges between tiles.

Figure 7 is a schematic diagram illustrating the function of a compiler 70. The compiler receives such a graph 60 and compiles the functions in the graphs into a multiplicity of codelets, which are contained into local programs labelled 72 in Figure 7. Each local program is designed to be loaded into a particular tile of the computer. Each program comprises one or more codelets 72a, 72b…plus a supervisor sub-program 73 each formed of a sequence of instructions. The compiler generates the programs such that they are linked to each other in time that is they are time deterministic. In order to do this the compiler accesses tile data 74 which includes tile identifiers which are indicative of the location of the tiles and therefore the delays which the compiler needs to understand in order to generate the local programs. The delays have already been mentioned above, and can be computed based on the tile data. Alternatively, the tile data can incorporate a data structure in which these delays are available through a lookup table.

There now follows a description of novel instructions which have been developed as part of the instruction set for the computer architecture defined herein. Figure 8 shows a SEND instruction of 32 bits. A SEND instruction indicates a data transmission from tile memory. It causes one or more data stored at a particular address in the local memory 22 of a tile to be transmitted at the exout interface of a tile. Each datum (referred to as "item" in the instruction) can be one or more words long. A SEND instruction acts on one word or multiple words to implement a send function. The SEND instruction has an opcode 80, a field 82 denoting a message count, the number of items to be sent in the form of one or more packet from the SEND address denoted in an address field 84. The field 84 defines the address in the local memory from which the items are to be sent in the form of an immediate value which is added to a base value stored in a base address register. The SEND instruction also has a send control field 86 (SCTL) which denotes the word size, selected as one of 4 and 8 bytes. The packet has no destination identifier in it: In other words, the recipient tile which is to receive the items is not uniquely identified in the instruction. The send function causes the specified number of data items from the send address to be accessed from the local memory and placed

at the ex_out interface of the tile to be transmitted at the next clock cycle. In another variation of the SEND instruction, the address from which items are to be sent could be implicit; taken from base value in the base address register and a delta value in an outgoing delta register. The delta value may be set based on information in a previous SEND instruction. In place of a unique identifier of the intended recipient tile, the compiler has arranged that the correct recipient tile will switch its local multiplexer(s) at the correct time to receive the datum (data items) as already described herein. Note that an intended recipient tile could be the transmitting tile itself in some cases.

To this end, a switch control function is provided, as described above. Figure 9 illustrates a PUT-i-MUX instruction which performs this function. An opcode field 90 defines the instruction as a PUT-i-MUX instruction. A delay period can be specified by a delay immediate value 92. This delay value can be used to replace 'no op' instructions, and is a way to optimise code compression. This instruction, when executed, defines in incoming_mux field 98 which input of the multiplexer 210 is to be set to 'listen' for items which have been sent from another tile. For the sake of compactness, this mux control function could be combined in a single instruction with a send function defined above, as shown in Figure 10. Note that there is no connection between the send function, which causes the tile to act as a transmitting tile, and the switch control function, which is a function when the tile is acting as a recipient tile, other than that they can be performed in a single execution cycle on the same tile.

Figure 10 is an example of a "merge" instruction. In this context, a "merge" instruction means an instruction that defines two or more functions which can be carried out at the same time (in one execution cycle) on one tile

Figure 10 illustrates a form of 'merge' send instruction, wherein a send function is combined with a second function which can modify the state held in registers at the tile. One function is to change the memory pointer for data received at that tile. Another function is to set the incoming MUX. The PUTi_MEMptr function enables a memory location in the local memory at which the next datum received by the tile is to be loaded to be identified. This function could be carried out by a dedicated 'receive' instruction, although its function is not to enable receipt of a datum but to modify the memory pointer. In fact, no specific instruction needs to be executed to receive data at a tile. Data arriving at the exin interface will be loaded

into the next memory location identified by the memory pointer, under the control of the exin interface. The instruction of Figure 10 has opcode field 100 and a number of items to be sent field 102. The immediate value in incoming state modification field 106 is written to an exchange configuration state register specified by field 104. In one form, the state modification field 106 may write an incoming delta for calculating the receive address to which the memory pointer is to be set. In another form the exchange configuration state is written with the incoming MUX value which sets the multiplexer input.

For this form of "merge" instructions, the send function uses a send address determined from values stored in one or more registers which is implicit in the instruction. For example, the send address can be determined from the base register and the delta register.

Figure 11 shows a "double width" instruction, referred to as an exchange instruction (EXCH). This instruction initiates a data transmission from an indicated address in the tile memory and sets the incoming exchange configuration state (the multiplexer and/ or the memory pointer for receiving data). The EXCH instruction is unique in that it is immediately followed by an inline 32-bit payload, located at the memory location immediately after the instructions. The EXCH instruction has an opcode field 110 which denotes an exchange instruction EXCH. The payload has a 'coissue' flag 119.

The EXCH instruction includes format field 112 which has a single bit which specifies incoming format datum width (32 bits or 64 bits). The datum width can have implications on the setting of the multiplexer lines, as explained later. An item field 114 defines the number of items which are caused to be sent by the exchange instruction. These items are sent from a sent address calculated using the immediate in field 116, as in the send instruction of Figure 9. The value in this field is added to the value in the base register.

Reference numeral 118 denotes a control field which defines word size for the send datum. The payload includes a switch control field 120 which acts a switch control for the incoming multiplexer, as described above in connection with Figure 9. Numeral 122 denotes a field of the payload defining an incoming delta for calculating the address at which incoming data is to be stored, as described above in connection with the instruction of Figure 10. The 64 bit wide exchange instruction EXCH of Figure 11 can be executed every clock cycle and thus allows simultaneously:

- sending from a particular address
- updating of incoming mux
- updating of incoming address

Thus, any exchange schedule can be encoded in a single instruction. The instructions of Figures 8, 9 and 10 perform similar functions but as they are only 32 bits long can be used to minimize the size of the exchange code in the local memory of each tile. The decision about which instruction to use in any particular context is made at the compiler 70 when constructing the codelets for the local program 72.

There follows a list of key registers and their semantics to support the above instructions. These registers from part of the register file on each tile.

TILE_ID	Holds a unique identifier for that tile
INCOMING_MUX [INCOMING_MUXPAIR]	Holds the Tile ID of the source tile for incoming messages, which acts to select the 'listening' input for the multiplexer associated with the receiving Tile.
INCOMING_DELTA	This holds an auto incrementing value for calculating on address at which incoming data are to be stored: it can be overwritten by an explicit field [e.g. see Figure 10]. It is added to INCOMING_BASE.
INCOMING_BASE	This holds a common base address for updating memory pointer (added to INCOMING_DELTA).
OUTGOING_BASE	This holds a common base address for send instructions
OUTGOING_DELTA	This holds delta for calculating send addresses instructions A 'send' address is outgoing base + outgoing delta.

INCOMING_FORMAT	Identifies 32b or 64b incoming datum.

Note that the INCOMING_DELTA and INCOMING_MUX register form part of the exchange state of tile.

Reference will now be made to Figures 12 and 13 to explain tile pairing which is a feature by which a physical pair of tiles may collaborate in order to make a more effective use of their combined exchange resources. Tile pairing may be used to double a single tile's transmission bandwidth by borrowing a neighbour's transmission bus, or double the received bandwidth for both tiles in a tile pair by sharing a neighbour's received bus and associated incoming multiplexer.

Figure 12 illustrates the logic associated with tiles in a tile pair for performing double width transmission. Double width transmission is achieved by borrowing a neighbour's outgoing exchange resources for the duration of a SEND. The neighbour tile is unable to perform its own data transmission during this time. A SEND instruction is able to perform single or double width data transfer, with the width of the transfer being specified by a value held in a register, or an immediate field. The width can be indicated as 32 bits (one word) in which case the field has a value of 0, or 64 bits (two words) in which case the field has a value of 1. Other logical definitions are possible. The specified width is passed from a register on the chip 4 to a control store 1200 in the Ex Out interface 226 of the tile. Figure 12 shows two such paired tiles, TID00 and TID01. The Ex Out interface 226 has buffers for accommodating the least significant word (LSW) and the most significant word (MSW). In this context, each word is 32 bits. The least significant word is connected directly to an input of a width control multiplexer 1202. The output of the multiplexer is connected to the corresponding cross-wires of the exchange bus 34, the cross-wires corresponding to the output wire for that particular tile. If the transmit width is set at 32 bits, the width control multiplexers 1202 are set to receive inputs from the respective LSW's of the paired tiles, to thereby allow the tiles of the pair to transmit a respective 32 bit word simultaneously.

If one member of the pair wishes to send a 64 bit word, the width control multiplexer 1202 of the neighbouring tile is set to receive the most significant word output from the sending tile

and to pass that to the output of the multiplexer. This will cause the most significant word of the 64 bit output from the sending tile to be placed on the cross wires of the exchange bus associated with the neighbouring tiles (which at this point is inhibited from sending anything). For the sake of clarity, the MUX control line from the width control flag in store 1200 of the sending tile TID00 is shown connected to the control input of the multiplexer 1202 of the neighbouring (non-sending) tile TID01. Similarly, the neighbouring tile TID01 also has a MUX control line connected from its control store 1200 to the input of the width control multiplexer 1202 of its paired tile, although this is not shown in Figure 12 for reasons of clarity.

Reference will now be made to Figure 13 to explain a double width receive using paired tiles. The paired tiles in Figure 13 are labelled TID03 and TID04, although it will readily be understood that this functionality can be used in combination with the double width transmit functionality such that a tile like TID00 could also have the functionality shown on TID03 for example. Double width receive is achieved by sharing a neighbour's incoming exchange resources for the duration of a transfer. When configured for double width receive, each tile within a tile pair can choose to sample or ignore the incoming data. If they both choose to sample, they will see the same incoming data. Double width receive is enabled in collaboration with the neighbour tile via the INCOMING_FORMAT value described earlier which identifies whether the incoming data is 32 bits or 64 bits. The value of the incoming multiplexer 210 of the primary tile of the tile pair must be set to the tile ID of the sending tile. The 'listening input' of the incoming multiplexer 210 of the secondary tile within the tile pair must be set to the tile ID of the other tile within the sending pair. Note that in this case, strictly speaking, the "sending" tile of the sending tile pair (for example TID01) is not actually sending, but has supplied its most significant word to use the exchange resources of tile TID00. Thus, the incoming multiplexers 210 of the tiles of the receiving tile pair must be respectively connected to the cross wires on which the individual words of the double width transmission output of the sending pair are placed.

Note that in some embodiments even if the incoming multiplexers 210 are switched to simultaneously listen to their respective cross wires of the exchange, this does not necessarily mean that the incoming values will be received at the tiles of the receiving tile pair simultaneously, due to the differing latencies of travel between the exchange and individual tiles. There are thus, three possibilities to consider in a receiving pair of tiles.

In a first possibility, the two incoming buses the Exin interface are to be treated independently (neither tile in the tile pair is participating in a double width receive).

According to the second possibility, the local incoming exchange bus is being used to transfer the early component of a double width item (and that component should now be delayed). This implies that the neighbour's bus will be used to transfer the non-early component of the same double width item.

According to the third possibility, the local incoming exchange bus is being used to transfer the non-early component of a double width item. This implies that the neighbour's bus was used to transfer the early component of the same double width item (and therefore the early data component on the neighbour's bus should have been delayed).

Figure 13 shows circuitry 1300 which deals with these scenarios using multiplexers 1302 and 1304. Note that the circuitry 1300 is duplicated on the input of each tile of the receiving tile pair, but is only shown on the input of TID03 for reasons of clarity.

Control of the multiplexer is from the incoming format control which is supplied from a register into an Exin interface 224. If the tile TID03 is to operate in a 32 bit mode, it controls the multiplexer 1302 to pass through 32 bit word at the upper input of the multiplexer in Figure 13 via a pipeline stage 1306 and a control buffer 1308.

If the receiving tiles are operating as a pair, the multiplexer 1302 is controlled to block its upper input and allow the least significant word from the lower input to be passed through to the pipeline stage 1306. On the next cycle, the most significant word is selected to be passed through the multiplexer 1304 into the control buffer 1308, along with the least significant word which has been clocked through the pipeline stage 1306. The control buffer 1308 can decide whether or not to receive the 64 bit word. Note that according to the logic the 64 bit word will simultaneously be received at the neighbouring tile (TID04). In some circumstances both tiles might want to read the same 64 bit value, but in other circumstances one of the tiles may wish to ignore it.

Note that there may be embodiments where the LSW and MSW of a 64 bit transfer may be simultaneously received at their paired receiving tiles, in which case the relative delay of pipeline stage 1306 would not be required.

There has been described herein a new computer paradigm which is particularly effective in the context of knowledge models for machine learning. An architecture is provided which utilises time determinism as in an exchange phase of a BSP paradigm to efficiently process very large amounts of data. While particular embodiments have been described, other applications and variance of the disclosed techniques may become apparent to a person skilled in the art once given the disclosure hearing. The scope of the present disclosure is not limited by the described embodiments but only by the accompanying claims.

Claims

1. A computer comprising:

a plurality of processing units each having instruction storage holding a local program, an execution unit executing the local program, data storage for holding data; an input interface with a set of input wires, and an output interface with a set of output wires;

a switching fabric connected to each of the processing units by the respective set of output wires and connectable to each of the processing units by the respective input wires via switching circuitry controllable by each processing unit;

a synchronisation module operable to generate a synchronisation signal to control the computer to switch between a compute phase and an exchange phase, wherein the processing units are configured to execute their local programs according to a common clock, the local programs being such that in the exchange phase at least one processing unit executes a send instruction from its local program to transmit at a transmit time a data packet onto its output set of connection wires, the data packet being destined for at least one recipient processing unit but having no destination identifier, and at a predetermined switch time the recipient processing unit executes a switch control instruction from its local program to control its switching circuitry to connect its input set of wires to the switching fabric to receive the data packet at a receive time, the transmit time and, switch time and receive time being governed by the common clock with respect to the synchronisation signal.

2. A computer according to claim 1, wherein the send instruction explicitly defines a send address identifying a location in the data storage from which the data packet is to be sent.

3. A computer according to claim 1, wherein no send address is explicitly defined in the send instruction, and the data packet is transmitted from a send address defined in a register implicitly defined by the send instruction.

4. A computer according to claim 3, wherein the local program comprises a send address update instruction for updating the send address in the implicit register.

5. A computer according to any preceding claim, wherein the transmit time is a known number of clock cycles after the send time at which the instruction is executed.

6. A computer according to any preceding claim, wherein the switching circuitry comprises a multiplexor having an exit set of output wires connected to its processing unit, and multiple sets of input wires connected to the switching fabric, whereby one of the multiple sets of input wires is selected as controlled by the processing unit.

7. A computer according to any preceding claim, wherein the recipient processing unit is configured to receive the data packet and load it into the data storage at a memory location identified by a memory pointer.

8. A computer according to claim 7, wherein the memory pointer is automatically incremented after each data packet has been loaded into the data storage.

9. A computer according to claim 7, wherein the local program at the recipient processing unit includes a memory pointer update instruction which updates the memory pointer.

10. A computer according to any preceding claim, wherein the send instruction identifies a number of data packets to be sent, wherein each data packet is associated with a different transmit time.

11. A computer according to claim 6, wherein one of the sets of input wires is connected to a null input.

12. A computer according to any preceding claim, wherein the recipient processing unit is the same processing unit as the processing unit that executed a send instruction at an earlier time, whereby the same processing unit is configured to send a data packet and receive that data packet at a later time.

13. A computer according to any preceding claim, wherein multiple processing units are configured to execute respective send instructions to transmit respective data packets, and wherein at least some of the data packets are destined for no recipient processing units.

14. A computer according to any preceding claim, wherein at least two of the processing units co-operate in a transmitting pair wherein a first data packet is transmitted from a first processing unit of the pair via its output set of connection wires, and a second data packet is transmitted from the first processing unit of the pair via the output set of connection wires of the second processing unit of the pair to effect a double width transmission.

15. A computer according to any preceding claim, wherein at least two of the processing units operate as a receiving pair wherein each processing unit of the pair controls its switching circuitry to connect its respective input set of wires to the switching fabric to receive respective data packets from respective tiles of a sending pair.

16. A method of computing a function in a computer comprising: a plurality of processing units each having instruction storage holding a local program, an execution unit for executing the local program, data storage for holding data, an input interface with a set of input wires and an output interface with a set of output wires; a switching fabric connected to each of the processing units by the respective sets of output wires and connectable to each of the processing units by their respective input wires via switching circuitry controllable by each processing unit; and a synchronisation module operable to generate a synchronisation signal to control the computer to switch between a compute phase and an exchange phase, the method comprising;

the processing units executing their local programs in the compute phase according to a common clock, wherein in the exchange phase at least one processing unit executes a send instruction from its local program to transmit at a transmit time a data packet onto its output set of connection wires, the data packet being destined for at least one recipient processing unit but having no destination identifier, and

at a predetermined switch time the recipient processing unit executing a switch control instruction from its local program to control the switching circuitry to connect its input set of wires to the switching fabric to receive the data packet at a receive time, the transmit time and switch time and being governed by the common clock with respect to the synchronisation signal.

17. A method according to claim 16, wherein the function is provided in the form of the static graph comprising a plurality of interconnected nodes, each node being implemented by a codelet of the local programs.

18. A method according to claim 17, wherein in the compute phase each codelet processes data to produce a result, wherein some of the results are not required for a subsequent compute phase and are not received by any recipient processing unit.

19. A method according to any of claims 16 to 18, wherein in the exchange phase the data packets are transmitted between processing units via the switching fabric and switching circuitry.

20. A method according to any of claims 16 to 19, wherein each processing unit indicates to the synchronisation module that its own compute phase is complete, and wherein the synchronisation signal is generated by the synchronisation module when all processing units have indicated that their own compute phase is complete, to commence the exchange phase.

21. A method according to claim 17, wherein the graph represents a machine learning function.

22. A method according to any of claims 16 to 21, wherein in the exchange phase data packets are transmitted through the switching fabric in a pipelined fashion through a sequence of temporary stores, each store holding a data packet for one cycle of the common clock.

Intellectual Property Office

Application No: GB1816892.2 **Examiner:** Mr Thomas Davies

Claims searched: 1-22 **Date of search:** 11 April 2019

Patents Act 1977: Search Report under Section 17

Documents considered to be relevant:

Category	Relevant to claims	Identity of document and passage or figure of particular relevance
A	-	US 2014/0006724 A1 (GRAY et al.) See especially paragraphs 42, 89-90.
A	-	US 5754789 A (NOWATZYK et al.) See especially column 8 lines 39-46, column 11 line 29, column 13 lines 35-41.
A	-	US 5434861 A (PRITTY et al.) See especially column 2 lines 19-52.

Categories:

X	Document indicating lack of novelty or inventive step	A	Document indicating technological background and/or state of the art.
Y	Document indicating lack of inventive step if combined with one or more other documents of same category.	P	Document published on or after the declared priority date but before the filing date of this invention.
&	Member of the same patent family	E	Patent document published on or after, but with priority date earlier than, the filing date of this application.

Field of Search:

Search of GB, EP, WO & US patent documents classified in the following areas of the UKC^{X} :

Worldwide search of patent documents classified in the following areas of the IPC

G06F; G06N

The following online and other databases have been used in the preparation of this search report

EPODOC, WPI, INSPEC, Patent Fulltext, XPESP, XPIEE, IP.COM, XPI3E, XPMISC, XPLNCS, XPRD, XPSPRNG, TDB

International Classification:

Subclass	Subgroup	Valid From
G06F	0009/52	01/01/2006
G06F	0009/30	01/01/2018
G06F	0015/173	01/01/2006