MPIntroduksjon

Et eksempel

Jon Nilsen

Kjerne- og energigruppen

Litt om parallisering

- Grunnleggende ideer:
  - Vil ha raskere og større beregninger $\Rightarrow$ parallellisering.
  - Flere prosessorer brukes for å løse et globalt problem.
  - Esensen er å dele beregningen jevnt mellom prosessorene.

Litt mer om parallisering

- Forskjellige paradigmer i parallelprogrammering.
  - Oppgaveparallelisering – deler opp det globale problemet i flere uavhengige oppgaver som sjelden trenger synkronisering. F.eks. Monte Carlo simulering.
  - Dataparallelisering – bruker flere tråder (f.eks. en tråd per prosessor) for å dele opp løkker over arrayer. Eksempler er kompilatorbasert parallelisering og OpenMP direktiver.
  - Message-passing – de involverte prosessorene har uavhengig minne. Brukeren må selv dele opp problemet og fordele bitene.

Message Passing Interface

- MPI er et bibliotek for message passing grensesnittet.
  - Uavhengig av maskinvare.
  - Ikke et språk eller en kompilator-spesifikasjon.
  - Ikke en spesifik implementasjon eller et produkt.
- Portabel og lettbrukelig standard for message passing.

Fundamentale MPI-rutiner

- `int MPI_Init(int *argc, char ***argv)`
  - Initialiserer et MPI-program.
  - Må bli kalt i alle MPI-programmer og før alle andre MPI-rutiner.
  - Utfører de rutiner som er nødvendige før andre MPI-rutiner kan brukes i en implementasjon.
- `int MPI_Finalize(void)`
  - Rydder opp etter MPI-rutinene.
  - Ingen MPI-rutiner kan kalles etter denne (selv ikke `MPI_Init`).
  - Bruker må selv passe på at MPI-avhengige prosesser er ferdige før `MPI_Finalize` kalles.

Fundamentale MPI-rutiner forts.

- `int MPI_Comm_size(MPI_Comm comm, int *size)`
  - Dersom `comm` er `MPI_WORLD_COMM` blir `size` totalt ant. tilgjengelige prosesser.
- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
  - Blir brukt sammen med foregående rutine og gir `rank` $\in \{0, size - 1\}$.
- `MPI_Comm_size` og `MPI_Comm_rank` er nyttige for å holde orden på de forskjellige prosessenes arbeidsoppgaver.

Fundamentale MPI-rutiner forts.

- `int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
  - `buf` er variabelen eller vektoren du ønsker å sende.
  - `count` er antall variable i `buf` (f.eks lengden på vektoren).
  - `datatype` er datatypen i `buf`, f.eks `MPI_INT` eller `MPI_DOUBLE`.
  - `dest` er `rank` til prosessen du vil sende til.
  - `tag` er en merkelapp for å holde orden på alle sendingene i programmet.
  - `comm` er i denne introduksjonen `MPI_COMM_WORLD`.

Fundamentale MPI-rutiner forts.

- `int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm)`
  - Alt må være likt som i den tilsv. `MPI_Send`, bortsett fra `source`.
  - `source` svarer til den prosessen som har sendt informasjonen.
- Med de 6 foregående rutinene kan man lage alle de MPI-programmene man måtte ønske.
- Alle andre rutiner er laget for å gjøre livet enklere for programmereren.

Noen andre MPI-rutiner.

- `int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)`
  - Blokkeringskall som returner når en matchende prosess er ferdig.
  - `source` er som oftest `MPI_ANY_SOURCE`

Noen andre MPI-rutiner forts.

```
int MPI_Reduce(void* sendbuf, void* recvbuf,
    int count, MPI_Datatype datatype,
    MPI_Op op, int root, MPI_Comm comm)
```

- `sendbuf` er det du vil sende.
- `recvbuf` er der du vil ha det.
- `count` er lengden på bufferene.
- `op` er operasjonen du vil gjøre for å sette sammen `sendbuf`ene til `recvbuf`.
- `root` er rangen til den prosessen som skal få all informasjonen.

Noen andre MPI-rutiner forts.

- `double MPI_Wtime(void)`
  - Returner tiden fra en eller annen gang i fortiden.
  - ”En eller annen gang i fortiden” er garantert å ikke endre seg i løpet av prosessens levetid.
  - Tiden som returneres er tid gått på noden som kaller på rutinen.

Noen tidsbetraktninger.

- Speed-up:

$$S(P) = \frac{T(1)}{T(P)}$$

  S er hastighetsøkningen, T er tiden og P er ant. prosesser

- Effektivitet:

$$\eta(P) = \frac{S(P)}{P}$$

  η er effektiviteten.

Noen tidsbetraktninger forts.

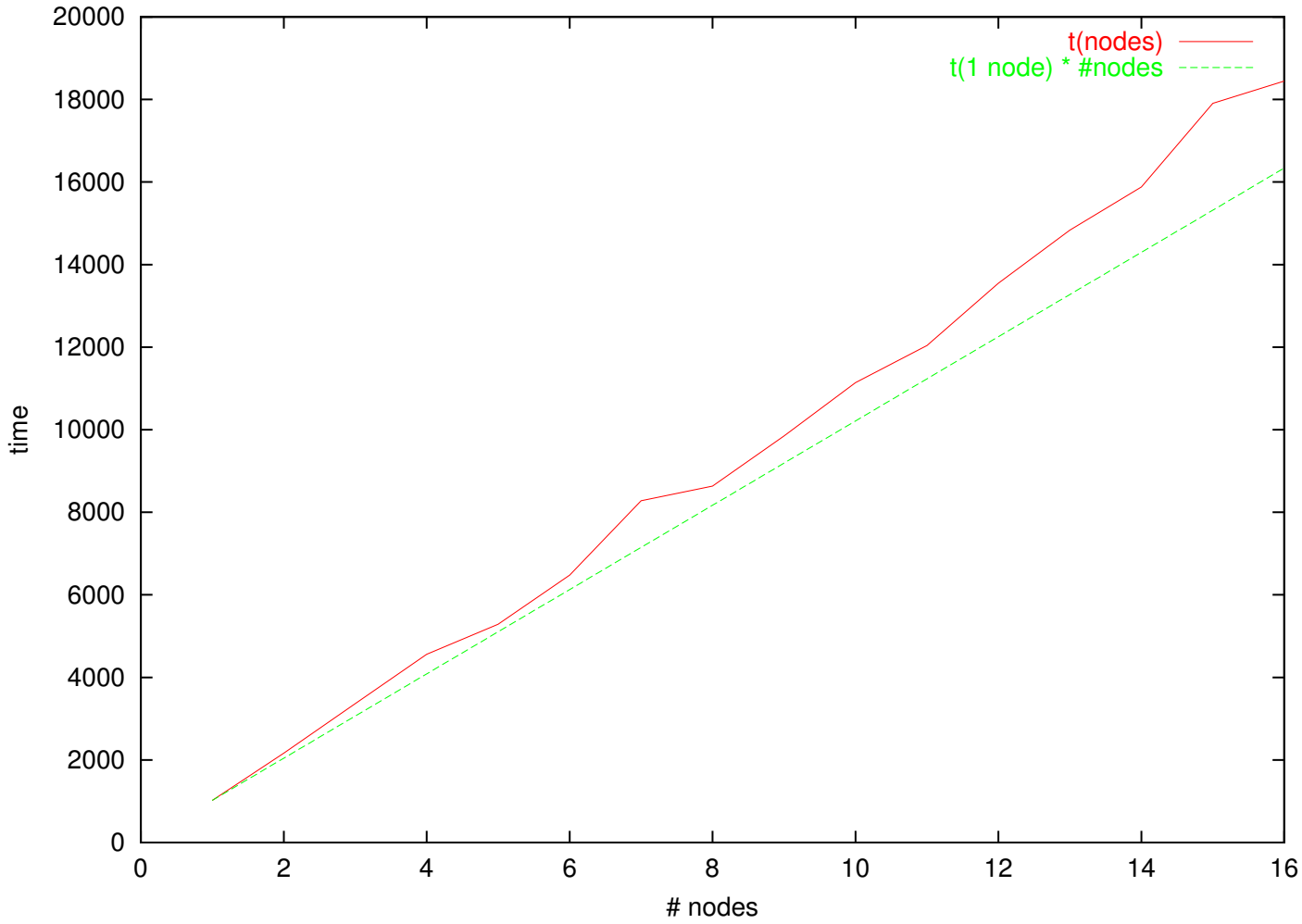
- Kostnadsmodell ved sending av en beskjed med lengde L mellom to prosessorer:

$$t_C(L) = \tau + \beta L$$

  t_C er tiden det tar å sende, τ er latensitet, β er båndbredden.
- Alle prosessene kan ikke gjøre nytte for seg hele tiden.
- Synkronisering og kommunisering tar tid.
- Ekstra beregninger som følge av parallellisering.
- Fører til at

$$T(P) \geq \frac{T(1)}{T(P)} \Rightarrow S(P) \leq P$$

Kjøringer med økende ant. prosesser.



Tid som funksjon av antall prosesser brukt. Grønn linje er ideell tid.

Eksempelet™.

- Variasjonell Monte Carlo $\Rightarrow$ hver kjøring er (eller kan gjøres) uavhengig.
- Dette gjør paralleliseringen veldig enkel.
- Programmet:
  - Ser på flytende He4 med Lennard-Jones potensial og topartikkel-korrelasjon.
  - Består i hovedsak av tre for-løkker inne i hverandre, som kjøres 10, 32 og 100000 ganger. Inne i dette regnes det ut to bølgefunskjoner.
  - Tar 20 minutter.
- Paralleliseringen kjører dette en gang pr. prosessor for å øke nøyaktigheten i energien.

Eksempelet™ forts.

- Programmet er laget etter master-slave modellen (veileder-student relasjonen).
  - null-rank-prosessen deler ut oppgaver til de andre prosessene (og seg selv). Disse gjør beregninger, snakker litt sammen og med master-prosessen og sender til slutt beregningene tilbake master-prosessen.
  - Master-prosessen samler sammen informasjonen (og publiserer).
- I eksempelet er det ingen kommunikasjon under beregningene, bare før og etter.

Mer info om MPI

- http://www.ifi.uio.no/~xingca/MPI-COURSE/